Deutsch

JENV V9.0A

Entwicklungs- und Ablaufumgebung

Benutzerhandbuch

Juni 2018

Inhaltsverzeichnis

Benutzerhandbuch

Kritik… Anregungen… Korrekturen…

Die Redaktion ist interessiert an Ihren Kommentaren zu diesem Handbuch. Ihre Rückmeldungen helfen uns, die Dokumentation zu optimieren und auf Ihre Wünsche und Bedürfnisse abzustimmen.

Sie können uns Ihre Kommentare per E-Mail an
manuals@ts.fujitsu.com senden.

Zertifizierte Dokumentation nach DIN EN ISO 9001:2015

Um eine gleichbleibend hohe Qualität und Anwenderfreundlichkeit zu gewährleisten, wurde diese Dokumentation nach den Vorgaben eines Qualitätsmanagementsystems erstellt, welches die Forderungen der DIN EN ISO 9001: 2015 erfüllt.

Copyright und Handelsmarken

1 Einleitung

Diese Dokumentation zum BS2000 Environment for Java™ (JENV) erläutert schwerpunktmäßig den Aufruf von Java-Kommandos, sofern diese von der Original-Beschreibung von Oracle abweichen, sowie Besonderheiten durch den Übergang von ASCII zu EBCDIC und mit dem Java Native Interface (JNI) im Rahmen von JENV V9.0A.
JENV V9.0A ist eine Implementierung der „Java Platform, Standard Edition“ (Java SE™) auf Basis von OpenJDK 9 für BS2000 mit dem vollständigen Namen „BS2000 Environment for Java™ V9.0A“.

Das Produkt enthält eine Ablaufumgebung (JRE), die die relevanten Spezifikationen erfüllt:

- „The Java Language and Virtual Machine Specifications, Java SE 9“
  http://docs.oracle.com/javase/specs/
- die versionsspezifische API Spezifikation
  „Java™ Platform, Standard Edition 9 API Specification“
  http://docs.oracle.com/javase/9/docs/api/

Darüber hinaus enthält das Produkt eine Entwicklungsumgebung (JDK) mit verschiedenen Entwicklungs-Tools. Diese können verwendet werden, um Anwendungen oder Applets zu entwickeln, die konform zur o.g. API-Spezifikation sind.

JENV V9.0A unterstützt alle Features von OpenJDK mit folgenden Ausnahmen:

- Audio-Features
- JDGA (Java Direct Graphic Access)
- Class Data Sharing.

JENV V9.0A enthält zusätzlich Fontdateien aus dem DejaVu Fonts Paket.

Als VM-Technologie kommt ausschließlich die HotSpot-Client-VM zum Einsatz.

Die Demo-Programme aus OpenJDK sind nicht in dem Produkt enthalten.

Gekündigte Java Pakete

Die Packages com.fsc.java.bs2000, com.fsc.java.io und com.fsc.jrio werden in JENV V9.0A letztmalig unterstützt. Ihre Funktionalität wurde seit JENV V7.0A durch die entsprechenden com.fujitsu.ts. Pakete ersetzt.
Die betroffenen Java-Quellen wurden durch eine Verschärfung der Annotation @Deprecated als "deprecated and marked for removal" markiert.

Modulkonzept

Die Pakete com.fujitsu.ts.java.bs2000 und com.fujitsu.ts.java.io sind im Modul java.base enthalten.
Das Paket com.fujitsu.ts.jrio ist im Modul jdk.jrio enthalten.

Optimierte Variante für S- und SQ-Anlagen

Für S- und SQ-Anlagen werden optimierte plattformabhängige Varianten zur Verfügung gestellt. Bei Bedarf kann die /390-Variante von JENV auch auf SQ-Anlagen installiert und verwendet werden.

1.1 Zielsetzung und Zielgruppen des Handbuchs

Die Dokumentation wendet sich an alle, die im BS2000 mit Java™ entwickeln und/oder Java™ in ihrer Systemumgebung anwenden wollen.

1.2 Konzept des Handbuchs

In diesem Handbuch sind ausschließlich die BS2000-Besonderheiten und spezielle BS2000-Teile beschrieben. Die Kenntnis der Original-Beschreibung von Oracle wird vorausgesetzt.

Übergang von ASCII zu EBCDIC

Java ist ein Produkt, das in einer ASCII-Welt (Unix- und Windows-Systeme) entwickelt wurde. Deshalb sind in einem Betriebssystem, dessen Code-Basis EBCDIC ist, einige Besonderheiten bei der Arbeit z.B. mit Codesets, Localized Streams, Print Streams und Standard Streams zu beachten. Diese Besonderheiten sind in der folgenden Dokumentation beschrieben.

JNI unter BS2000

Die Dokumentation beschreibt außerdem die Besonderheiten, die Sie als Benutzer der Java native Schnittstellen (JNI) im BS2000 zu beachten haben, z.B. die Verwendung von Java-Datentypen in C und das dynamische Laden von native Methoden.

Aufbau der Dokumentation

In diesem Handbuch finden Sie

- im Kapitel „Umgebungsvariablen" die Beschreibung der Umgebungsvariablen.
- im Kapitel „Übergang von ASCII zu EBCDIC" die Besonderheiten, die wegen des anderen Codeset des BS2000 (EBCDIC) zu beachten sind.
- im Kapitel „Das Java-Package JRIO" die Schnittstellen und Implementierungen von JRIO.
- im Kapitel „Aufruf der VM von der BS2000-Kommandooberfläche" die Prozeduren *INITIALIZE*, *DELETE* und *START*.
- im Kapitel „JNI unter BS2000" die Besonderheiten, die ein Benutzer der Java native Schnittstellen (JNI) im BS2000 zu beachten hat.
- im Kapitel „JCI - Invocation-API für COBOL" die Besonderheiten, die ein Benutzer des Java-COBOL-Interface (JCI) im BS2000 zu beachten hat.
- im Kapitel „Kommandos für BS2000" die Kommandos *mk_shobj* und *pr_shobj*, die in JENV zusätzlich implementiert sind, und die Kommandos deren Beschreibung von der in „JDK Tools and Utilities“ [11] abweicht.
- im Kapitel „Anhang: Kompatibilität zu Vorgängerversionen und Migration" die Beschreibung der Inkompatibilitäten von JENV V9.0A zu den Vorgängerversionen und der Migration von früheren Versionen zu JENV V9.0A

Readme-Datei

Funktionelle Änderungen der aktuellen Produktversion und Nachträge zu diesem Handbuch entnehmen Sie bitte ggf. der produktspezifischen Readme-Datei.

Readme-Dateien stehen Ihnen online bei dem jeweiligen Produkt zusätzlich zu den Produkthandbüchern unter *http://manuals.ts.fujitsu.com* zur Verfügung. Alternativ finden Sie Readme-Dateien auch auf der Softbook-DVD.

Informationen unter BS2000

Wenn für eine Produktversion eine Readme-Datei existiert, finden Sie im BS2000-System die folgende Datei:

`SYSRME.<product>.<version>.<lang>`

Diese Datei enthält eine kurze Information zur Readme-Datei in deutscher oder englischer Sprache (<lang>=D/E). Die Information können Sie am Bildschirm mit dem Kommando /SHOW-FILE oder mit einem Editor ansehen.

Das Kommando `/SHOW-INSTALLATION-PATH INSTALLATION-UNIT=<product>` zeigt, unter welcher Benutzerkennung die Dateien des Produkts abgelegt sind.

Ergänzende Produkt-Informationen

Aktuelle Informationen, Versions-, Hardware-Abhängigkeiten und Hinweise für Installation und Einsatz einer Produktversion enthält die zugehörige Freigabemitteilung. Solche Freigabemitteilungen finden Sie online unter *http://manuals.ts.fujitsu.com* .

1.3 Darstellungsmittel

Diese Dokumentation verwendet die folgenden Darstellungsmittel.

1.3.1 Beschreibung von Kommandos

Die Beschreibung der Kommandos hält sich - soweit möglich - an ein festes Raster:

Syntax

Darstellung der Kommandosyntax.

Beschreibung

Bedeutung, Funktion und Arbeitsweise des Kommandos. Ggf. wird erklärt, welche Voraussetzungen oder Bedingungen zu beachten sind.

Optionen

Beschreibung der zugehörigen Kommandozeilenoptionen.

Siehe auch

Texte, in denen weitere Hinweise auf das Kommando zu finden sind.

Syntaxdarstellung

Die verwendete Metasyntax hat folgende Bedeutung:

Halbfette Zeichen

Konstanten. Halbfett gedruckte Zeichen müssen genau wie dargestellt eingegeben werden.

Normale Zeichen

Variablen. Diese Zeichenfolgen sind Stellvertreter für konkrete Werte, die Sie eingeben oder auswählen.

Kursive Zeichen

Variable Teile in Optionen, für die Sie konkrete Angabe einsetzen müssen.

[ ]

Optionen. Argumente in eckigen Klammern sind optional und müssen nicht angegeben werden. Die eckigen Klammern dürfen nicht eingegeben werden.

...

Der vorherige Ausdruck kann wiederholt werden.

{ | }

Auswahlmöglichkeit. Wählen Sie genau einen der Ausdrücke aus, die durch senkrechte Striche getrennt sind. Die geschweiften Klammern dürfen nicht eingegebenwerden.

1.3.2 Namen von Dateien, Kommandos und Programmen

Namen von Dateien, Kommandos, Programmen etc. werden im Text in *kursiver* Schrift dargestellt. Kommen dabei Variablen vor, so stehen sie in *<spitzen>* Klammern.

1.3.3 Beschreibung von Arbeitsabläufen

Tätigkeiten sind in einzelne Schritte unterteilt:

- Schritt, der Teil des gesamten Arbeitsablaufs ist. Sie geben hier ein Kommando ein oder führen eine Aktion aus.

1.4 Weitere Informationen und Quellen

Sie finden weitere Informationen zu Java™

- im Kapitel „Literatur"
- unter der Web-Seite mit der URL
  http://www.fujitsu.com/fts/products/computing/servers/bs2000/software/programming/javabs2000.html

1.5 Lizenzrechtliche Bestimmungen

JENV V9.0A ist Open Source Software.
JENV basiert auf einer Portierung von OpenJDK 9.
Alle lizenzrechtlich relevanten Informationen befinden sich in

SYSDOC.JENV.090.OSS

oder im Internet unter

http://docs.ts.fujitsu.com/dl.aspx?id=9149a4ae-06ef-42fd-a971-ecff349bcc66.

2 Umgebungsvariablen

In diesem Kapitel sind die folgenden Umgebungsvariablen beschrieben:

CLASSPATH

Die Umgebungsvariable *CLASSPATH* entspricht in ihrem syntaktischen Aufbau der *PATH*-Umgebungsvariablen und beschreibt die Verzeichnisse und JAR-Archive, in denen nach Benutzer-Klassen gesucht wird.

Bei Benutzung der *java*-Kommandos und der Tools muss der Anwender diese Umgebungsvariable nur so belegen, dass seine eigenen Klassen gefunden werden. Ist die Umgebungsvariable nicht gesetzt, so wird (außer beim *appletviewer*) der Suchpfad für Benutzerklassen auf das aktuelle Verzeichnis gesetzt.

Alternativ kann für die JAVA-Interpreter auch die Option *–classpath* verwendet werden, um den Pfad zu den Benutzer-Klassen zu definieren.

JAVA_HOME

Die Umgebungsvariable *JAVA_HOME* beschreibt den Installationsort der JAVA-Ablaufumgebung. Sie wird nur für Anwender-Programme benötigt, die JAVA über das Invocation-API ansprechen.

Für eine Standardinstallation ist *JAVA_HOME* also auf */opt/java/jdk-9.0.4* zu setzen. Die aktuell gültige Bezeichnung ist der Freigabemitteilung zu entnehmen.

Die Java-Tools benutzen eigene Mechanismen, um ihren Installationsort zu bestimmen. Für die Benutzung des Java-Interpreters und der anderen Java-Tools sollte diese Umgebungsvariable daher nicht gesetzt werden.

JENV_VMTYPE

Für Benutzerprogramme, die das Invocation-API benutzen, existiert keine Schnittstelle, um den VM-Typ für den Ablauf auszuwählen. Mit dieser Umgebungsvariable kann eine spezielle VM für solche Programme angefordert werden. Folgende Werte sind erlaubt:

client

Auswahl der HotSpot™ Client-VM

Wird die Variable nicht gesetzt, so gilt der Standardwert (siehe Abschnitt „Option zur Auswahl des HotSpot ™ VM-Typs" im Abschnitt „java"). Da derzeit nur eine VM-Implementierung verfügbar ist, ist diese Umgebungsvariable noch nicht erforderlich.

Die Java-Tools nutzen diese Umgebungsvariable nicht, sondern werten die entsprechenden Kommandozeilen-Optionen aus.

JENV_SYSHSI

Die Umgebungsvariable *JENV_SYSHSI* stellt die HSI-Variante der VM bei Aufruf des Kommandos *java* ein. Folgende Werte sind erlaubt:

s390

Es wird die S390-Variante von JENV verwendet (falls verfügbar).

x86

Es wird die X86-Variante von JENV verwendet (falls verfügbar).

Setzen Sie die Variable nicht, gilt der Standardwert, wie in Abschnitt „Optionen zur Auswahl der HSI-Variante" im Abschnitt „java" beschrieben. Stellen Sie die HSI-Variante explizit im Kommando *java* ein, hat dieser Wert Vorrang vor der Umgebungsvariablen.

LD_LIBRARY_PATH

Die Umgebungsvariable *LD_LIBRARY_PATH* beschreibt die Verzeichnisse, in denen nach „Shared Objects“ mit native Methoden des Benutzers gesucht wird. Sie entspricht in ihrem syntaktischen Aufbau der Umgebungsvariablen *PATH*.

Für die Suche nach native Methoden der Java-Implementierung werden andere Mechanismen verwendet. Bei Anwender-Programmen, die das Invocation-API benutzen, werden sie z.B. über *JAVA_HOME* gefunden.

3 Übergang von ASCII zu EBCDIC

Das Java SE JDK wurde in einer ASCII-Umgebung (Unix- und Windows-Systeme) entwickelt. Dementsprechend sind wegen des anderen Codeset des BS2000 (EBCDIC) einige Besonderheiten zu beachten, die im folgenden Kapitel beschrieben werden.

3.1 Codesets

Im Gegensatz zu ASCII-basierten Betriebssystemen, wo wegen der partiellen Identität zwischen ASCII und Unicode nicht immer genau zwischen Text- und Binär-Ein-/Ausgabe unterschieden werden muss, ist diese Unterscheidung im BS2000/OSD (und anderen nicht auf ASCII basierenden Betriebssystemen - wie z.B. OS/390) wichtig. Wird das von Java-Programmen nicht berücksichtigt, sind sie nicht portabel und werden auf BS2000 nicht ohne Änderung korrekt laufen.

Java arbeitet intern im Unicode. Für die Kommunikation mit der Außenwelt kann Java beliebige Codes verwenden. Für die Ein-/Ausgabe von Text-Daten wurden die Klassen *InputStreamReader* und *OutputStreamWriter* eingeführt, die entsprechende Code-Umwandlungen vornehmen. Die dabei verwendete Standard-Code-Umwandlung wird durch den Wert der System-Property *file.encoding* bestimmt. Sie ist standardmäßig mit OSD_EBCDIC_DF04_1 festgelegt. Diese Einstellung kann beim Aufruf von Java entweder global über *-Dfile.encoding=XXX* geändert werden oder lokal durch Angabe eines entsprechenden Codesets bei der Instanziierung der Klassen *InputStreamReader* und *OutputStreamWriter*.

Unterstützte Codesets

Folgende Codesets werden im BS2000 zusätzlich unterstützt und stehen demzufolge in anderen Java-Implementierungen **nicht** zur Verfügung:

OSD_EBCDIC_DF04_1

> Standard-Codeset im BS2000. Er entspricht dem Zeichensatz EBCDIC.DF.04-1 mit der Modifikation, dass die EBCDIC-Zeichen x'15' und x'25' vertauscht sind, also x'15' als Newline-Zeichen interpretiert wird. Dies entspricht der aktuellen Praxis im POSIX bzw. in der C-Programmierung im BS2000.
>
> Dieser Zeichensatz ist kompatibel zum ISO-Zeichensatz 8859-1, dem Standard-Zeichensatz in den Unix-Systemen. Kompatibel heißt, er enthält den gleichen Zeichensatz nur in anderer Codierung, ist also 1:1 abbildbar.

OSD_EBCDIC_DF03_IRV

> Zeichensatz EBCDIC.DF.03.IRV (Internationale Referenz Version), ebenfalls mit x'15' als Newline-Zeichen.

OSD_EBCDIC_DF04_15

> entspricht dem Zeichensatz EBCDIC.DF.04-15 mit der Modifikation, dass die EBCDIC-Zeichen x'15' und x'25' vertauscht sind, also x'15' als Newline-Zeichen interpretiert wird. Dies entspricht der aktuellen Praxis im POSIX bzw. in der C-Programmierung im BS2000.
> Dieser Zeichensatz ist voll kompatibel zum ISO-Zeichensatz 8859-15. Kompatibel heißt, er enthält den gleichen Zeichensatz nur in anderer Codierung, ist also 1:1 abbildbar.

Angabe des Codeset

Die Kommandos *javac*, *javadoc*, *appletviewer/* und *native2ascii* unterstützen die Option *-encoding*. Mit dieser Option geben Sie den Zeichensatz für die Dateien an, auf die das Kommando zugreifen soll.

3.2 Localized Streams

Für JENV wurden analog zu OS/390 verschiedene neue Klassen und Methoden für die Localized Streams implementiert und damit einige der ASCII/EBCDIC-Probleme gelöst. Als Anwendungsprogrammierer sollten Sie aber konsequent die von Oracle America Inc. definierten Klassen *InputStreamReader* und *OutputStreamWriter* zur Ein-/Ausgabe von Text verwenden.

Die zu diesem Zweck implementierten neuen Klassen sind:

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

Diese Klassen können nicht instanziiert werden, sie bieten aber eine statische Methode *localize()* an, die einen gegebenen Stream in einen „Localized Stream“ umwandeln, wenn der gegebene Stream auf einer Datei basiert.

Diese Methoden sind:

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

Diese Methoden liefern im BS2000 einen *InputStream* bzw. *OutputStream* zurück, dessen Verhalten gegenüber dem ursprünglichen Stream dergestalt modifiziert ist, dass die gesamte E/A über diesen Stream der Codeset-Konvertierung von bzw. nach dem implementierten Standard-Codeset (Wert der System-Property *file.encoding*) unterliegt. Dies geschieht allerdings nur für Streams, die auf Dateien basieren. Auf andere Streams (z.B. *ByteArray*) haben diese Methoden keine Auswirkung.

Diese so modifizierten Streams verhalten sich also analog zu den Objekten der neuen Klassen *InputStreamReader* und *OutputStreamWriter*, bleiben aber im Gegensatz dazu vom Datentyp *InputStream* bzw. *OutputStream*, können also überall dort verwendet werden, wo nur Objekte dieser Typen zulässig sind.

Vorkehrungen gegen Doppel-Konvertierungen sind getroffen. So kann ein Stream nicht zweimal „Localized“ werden. Beim Aufruf einer *getLocalized...*-Methode für einen bereits lokalisierten Stream, wird dieser einfach wieder zurückgegeben. Auch von einem „Localized Stream“ kann eine Instanz eines *InputStreamReader* bzw. *OutputStreamWriter* gebildet werden, ohne dass deswegen Doppel-Konvertierungen befürchtet werden müssen.

Diese JENV-spezifische Erweiterung kann abgeschaltet werden, indem die System-Property *java.localized.streams* auf den Wert *False* gesetzt wird. Dies kann beim Aufruf von Java über *-Djava.localized.streams=False* erreicht werden.

3.3 Property-Files

Mit den Methoden *store()* und *load()* der Klasse *java.util.Properties* können Property-Files geschrieben und gelesen werden. Handelt es sich bei den angegebenen Streams um Dateien (File-Streams), wird im BS2000 davon ausgegangen, dass diese Dateien im Standard-Codeset (Wert der System-Property *file.encoding*) gelesen bzw. erzeugt werden.

Dies geschieht nicht, wenn die JENV-spezifische Erweiterung für die „Localized Streams“ abgeschaltet wurde (siehe Abschnitt „Localized Streams"). Property-Files werden dann immer im Encoding ISO8859-1 (ASCII-Encoding) geschrieben bzw. erwartet.

Dieses Verhalten ist mit dem auf IBM-Systemen kompatibel.

3.4 Policy-Files

Policy-Files, die von der Standard Policy-Implementierung verwendet werden, müssen im Codeset UTF-8 codiert sein. Das *policytool* verarbeitet und erzeugt entsprechend auch nur UTF-8 codierte Policy-Files. Da der UTF-8 Codeset in den ersten 127 Zeichen mit dem ASCII Codeset übereinstimmt, kann der Anwender eine Datei in diesem Codeset auch erzeugen, indem er die Datei mit dem Editor im üblichen Native Codeset (*OSD_EBCDIC_DF04_1*) erzeugt und dann mit dem Tool *native2ascii* in den ASCII Codeset konvertiert.

> **ACHTUNG!**
>
> *native2ascii* übernimmt beim Erzeugen der neuen Datei nicht die Zugriffsrechte der alten Datei. Diese müssen ggf. noch mit *chmod* geändert werden.

Seit Version JENV V1.4B steht die System-Property *sun.security.policy.utf8* zur Verfügung, mit der auch Policy-Files mit Native Codeset verwendet werden können. *sun.security.policy.utf8* kann mit den Werten *true* oder *false* belegt werden. Mit folgenden Aufruf können also wieder Policy-Files im Native Encoding verwendet werden:

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

Wir empfehlen aber, UTF-8 codierte Policy-Files zu verwenden.

3.5 PrintStream

Die Ausgabe-Streams vom Typ *java.io.PrintStream* sind in der BS2000-Portierung standardmäßig nicht modifiziert, werden aber hier erwähnt, weil sie besondere Schwierigkeiten verursachen können.

Methoden der Klasse java.io.PrintStream

Entsprechend der Java-Spezifikation konvertieren einige Methoden der Klasse *java.io.PrintStream* ihre Ausgaben in den Standard-Codeset (Wert der System-Property *file.encoding*), andere aber nicht. Mit dieser Klasse ist es deshalb besonders leicht, Programme zu schreiben, die in einer ASCII-Welt scheinbar funktionieren, aber im BS2000 nicht das erwartete Ergebnis liefern. Das folgende einfache Beispiel soll das verdeutlichen:

Beispiel

```
...
PrintStream out = new PrintStream(new FileOutputStream("test"));
...
out.print("Das ist ein Text.");
out.write('\n');
...
```

In einem ASCII-basierten System wird der Inhalt der Datei *test* anschließend eine mit Newline abgeschlossene Zeile mit obigem Text sein. Im BS2000 wäre zwar der Text in EBCDIC-Codierung in der Datei zu finden, die Zeile wäre aber nicht mit Newline abgeschlossen, sondern würde als letztes Zeichen ein Schmierzeichen enthalten.

Das zeigt anschaulich, wie wichtig es ist, bei der Neuimplementierung von Java-Code für die Ein-/Ausgabe von Text die neuen Reader- und Writer-Klassen (also *InputStreamReader* und *OutputStreamWriter*) zu verwenden.

Im BS2000 steht ein zusätzlicher Schalter bereit, der das Verhalten von *PrintStream* derart ändert, dass keine Methode mehr eine Konvertierung vornimmt. Dies kann beim Aufruf von Java über *-Djava.localized.print=False* erreicht werden. Mit dieser Einstellung verhält sich die Klasse *PrintStream* nicht mehr spezifikationsgemäß, dies kann aber für existierende Anwendungen trotzdem nützlich sein.

Der Vollständigkeit halber sei nochmals erwähnt, dass die Verwendung von „Localized Streams“ als Basis für PrintStreams oder die „Lokalisierung“ eines PrintStreams nicht zu Mehrfach-Konvertierungen führt. Für derartig behandelte PrintStreams gilt dann aber natürlich, dass **alle** Methoden konvertieren.

Zusammenwirken der Methoden readLine() und println()

Häufig wird angenommen, dass mit *println()* auf einen *PrintStream* geschriebene Daten durch die *readLine()* Methoden einiger *InputStream*-Klassen wieder gelesen werden könnten. Im BS2000 führt diese Annahme zu einem Fehler, da bei der Ausgabe auf einen *PrintStream* eine Konvertierung in den Native Codeset (im BS2000 OSD_EBCDIC_DF04_1) vorgenommen wird, aber keine der *readLine()* Methoden der *InputStream*-Klassen beim Lesen gleiches tut. Verwenden Sie hier als Ersatz die neuen *Reader*- und *Writer*-Klassen oder verwenden Sie ebenfalls „Localized Streams“ bei der Eingabe.

3.6 Standard Streams

Die Klasse *java.lang.System* stellt drei Standard Streams *in*, *out* und *err* zur Verfügung. Analog zur Lösung im OS /390 sind diese Standard Streams in JENV „Localized Streams“. Damit ist im BS2000 eine normale Text- Ein-/Ausgabe über diese Streams möglich.

Dies kann selektiv für jeden der drei Ströme eingestellt werden, indem beim Programmstart die folgenden System-Properties versorgt werden:

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

Die Ströme werden nicht modifiziert, wenn die Erweiterung für „Localized Streams“ abgeschaltet ist (*-Djava.localized.streams=False*). Ein späteres Setzen oder Ändern dieser System-Properties bleibt für die aktuell eingestellten Standard-Ströme ebenfalls wirkungslos.

Folgende Werte können angegeben werden:

Default
: Die Original-Ströme (die beim Programmstart eingestellt sind) werden lokalisiert.
Das ist die Voreinstellung.

Full
: Sowohl die Original-Ströme, als auch die später mit *setIn()* usw. gesetzten Standardströme werden lokalisiert.

None
: Die Standardströme werden nicht modifiziert.

Soll eine Anwendung die Methoden *setIn()*, *setOut()* oder *setErr()* benutzen, um eigene Streams zuzuweisen, so gibt es zwei Möglichkeiten für einen korrekten Ablauf: Entweder es wird dafür gesorgt, dass die Standard-Streams immer „Localized Streams“ (also Text-Streams) sind, oder es wird bei der Verwendung der Standard-Streams auf eine saubere Unterscheidung zwischen Text- und Binär-Ein-/Ausgabe geachtet. Das folgende Beispiel zeigt die Anwendung beider Möglichkeiten.

Die zweite Lösung ist dabei die bessere. So sollte grundsätzlich mit den Standard-Streams gearbeitet werden. Die erste Lösung kann aber notwendig sein, wenn mit bereits existierenden Java-Klassen gearbeitet werden muss, die nicht portabel implementiert wurden.

Beispiel

Der folgende Code (den man ähnlich unter den Demo-Programmen von JavaSoft finden kann) würde zu einer binären Ein-/Ausgabe über diese Ströme führen und damit zu unlesbaren Ausgabedateien oder einer möglichen Fehlinterpretationen der Eingabe, falls eigentlich Text Ein-/Ausgabe gemeint war.

```
...
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();
...
```

Das folgende Programm-Fragment zeigt die erste Lösung: die Standard-Streams sind immer „Localized Streams“ (also Text-Streams). Diese Lösung wäre vom Aufrufer zu realisieren.

```
...
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();
...
```

Das Programm-Fragment der zweiten Lösung könnte so aussehen und wäre vom Nutzer der Standard-Streams zu implementieren: bei der Verwendung der Standard-Streams wird auf eine saubere Unterscheidung zwischen Text- und Binär-Ein-/Ausgabe geachtet.

```
...
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());
}
...
```

3.7 JAR-Archive

JAR-Archive bereiten im Zusammenhang mit der ASCII/EBCDIC Problematik besondere Schwierigkeiten, weil sie auch ein Austausch-Format zwischen verschiedenen Umgebungen (Systemen) darstellen. So können Sie Applets einschließlich aller ihrer Ressourcen in JAR-Archive verpacken und über das Netz von einem Browser laden lassen. Java bietet entsprechende Zugriffsmethoden auf die so verpackten Ressourcen (siehe *java.util.ResourceBundle*).

Zu den typischen Ressourcen gehören dabei häufig auch Property-Files (z.B. mit Fehlermeldungen). Um die Austauschbarkeit zu gewährleisten, müssen Property-Files, die in JAR-Archiven abgelegt werden, daher immer in der Codierung ISO8859-1 vorliegen. Sie müssen also durch den Ersteller eines solchen JAR-Archivs im BS2000 zuvor in diesen Codeset umgewandelt werden.

Wenn der Anwender ein eigenes Manifest-File in das JAR-Archiv einbringt (Option *-m*), gilt:

- Wird das Manifest-File vom *jar*-Kommando selbst generiert, so geschieht das automatisch in der Codierung ISO8859-1.
- Erstellt der Anwender das Manifest-File selbst, so muss es vorher in den Codeset ISO8859-1 umgewandelt werden.

Die Zugriffsmethoden auf diese Ressourcen in JAR-Archiven sind so gestaltet, dass sie auch im BS2000 einen ASCII-Input erwarten.

Zur Unterstützung bei der Code-Umwandlung von Dateien steht das Kommando *native2ascii* zur Verfügung.

3.8 Programm-Argumente

Die Aufruf-Argumente, die an die Methode *main()* eines Java-Programms übergeben werden, werden automatisch einer Konvertierung von EBCDIC nach Unicode unterzogen.

4 Das Java-Package JRIO

Das Package JRIO ist eine Sammlung von Java Klassen zum direkten Umgang mit Dateien, die eine Satz- und /oder Blockstruktur haben, und zur satz- bzw. seiten-orientierten Ein/Ausgabe auf solche Dateien. Darunter fallen insbesondere die BS2000-Dateien des DMS/DVS.

Anders als die normale Java-I/O (Package *java.io*) erlauben diese Schnittstellen auch Operationen, die mit den vorgegebenen (und von uns nicht erweiterbaren) Java-IO-Klassen, nicht auszudrücken sind.

Die Schnittstellen und Implementierungen von JRIO sind im proprietären Package *com.fujitsu.ts.jrio* und weiteren untergeordneten Packages zusammengefasst. Diese werden in anderen Java Implementierungen nicht verfügbar sein. Sie haben aber, soweit technisch sinnvoll und möglich, eine starke Ähnlichkeit zum entsprechenden Package *com.ibm.recordio* von IBM.

4.1 Konzepte

Die Implementierung von JRIO ist auf Erweiterbarkeit angelegt. Die folgenden Abschnitte beschreiben diese Konzepte und ihre Realisierung.

4.1.1 Dateisysteme

Anders als bei den normalen Java-IO-Klassen, werden unter JRIO verschiedene Dateisysteme per Konzept unterstützt (siehe Abschnitt „RecordFile"). In zukünftigen Versionen sollen folgende Dateisysteme unterstützt werden:

- das BS2000-Dateisystem (im folgenden DMS genannt)
- das hierarchische Dateisystem des POSIX (im folgenden UFS genannt)
- das Bibliotheks-Dateisystem des BS2000 (im folgenden LMS genannt)

In dieser Version wird zunächst nur DMS unterstützt.

In jedem der Dateisysteme gilt eine eigene Syntax zur Angabe von Dateinamen. Beim Erzeugen eines *RecordFile*-Objektes wird dieses einmalig und endgültig mit einem Dateisystem assoziiert. Diese Zuordnung kann implizit oder explizit durch Angaben des Benutzers erfolgen und kann dann nicht mehr geändert werden. Diese Zuordnung bestimmt dann die Semantik der meisten Methoden des *RecordFile*-Objektes.

4.1.1.1 Dateinamen im DMS-Dateisystem

Dateinamen im Dateisystem DMS werden entsprechend der Regeln dieses Dateisystems (Handbuch „Einführung in das DVS“ [8]) gebildet. Teilqualifizierte Dateinamen und Wildcard-Angaben werden an keiner der JRIO-Schnittstellen unterstützt, mit der Ausnahme der als Verzeichnis zulässigen Angaben und der Dateibezeichner in Policy-File (siehe auch Abschnitt „Sicherheit").

Als Verzeichnisse im DMS-Dateisystem werden nur die Angabe einer allein stehenden Katalogkennung (Catid), die in Doppelpunkte eingeschlossen ist, einer Benutzerkennung (Userid) mit einleitendem Dollar-Zeichen und abschließendem Punkt oder die Kombination aus beidem angesehen. Die übliche Sonderform zur Angabe der System-Standard-Kennung ist ebenfalls zugelassen. Es gibt also für Verzeichnisse im DMS nur folgende möglichen Angaben:

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

Da das DMS als flaches Dateisystem eigentlich kein Verzeichnis-Konzept kennt, können Verzeichnisse also auch nicht mit den hier bereitgestellten Schnittstellen eingerichtet oder gelöscht werden. Ebenso haben sie keine Eigenschaften, wie Modifikationsdatum oder eine Größe. Lediglich die Methoden zum Auflisten von Verzeichnisinhalten können auf die oben genannten künstlichen Verzeichnisse des DMS sinnvoll angewendet werden.

Wie auch in den DMS-Schnittstellen, werden die so genannten logischen Systemdateien (SYSFILE-Umgebung) nicht unterstützt. Von JRIO werden darüberhinaus keine EAM-Dateien unterstützt.

Normalisierte Pfadnamen

Bei der Erzeugung eines *RecordFile*-Objektes und an anderen Stellen, wo der Benutzer an den JRIO-Schnittstellen einen Datei- oder Pfadnamen angeben kann, erfolgt neben der syntaktischen und semantischen Prüfung dieses Namens auch eine so genannte Normalisierung. Für DMS-Dateien bedeutet das Normalisieren des Namens, dass alle evtl. im angegebenen Namen auftretenden Kleinbuchstaben in Großbuchstaben umgewandelt werden. Außerdem werden Dateinamen, die keine Punkte enthalten, aber mit einem Dollarzeichen ($) beginnen, entsprechend der Konventionen des DMS in einen Namen mit vorangestellter System-Standard-Kennung umgewandelt:

Beispiel

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

Absolute Pfadnamen

Ein Pfadname im DMS wird als absolut angesehen, wenn er mit einer Katalogkennung beginnt. Wird ein absoluter Pfadname erzeugt, bedeutet das also, wenn nicht schon eine Katalogkennung im Namen enthalten ist, dass die Standard-Katalogkennung der evtl. angegebenen Benutzerkennung oder die des Aufrufers hinzugefügt wird (siehe Abschnitt „RecordFile").

Kanonische Pfadnamen

Ein Pfadname im DMS ist kanonisch, wenn er entweder nur aus einer Katalogkennung besteht oder sowohl Katalogkennung als auch Benutzerkennung enthält. Wird ein kanonischer Pfadname erzeugt, bedeutet das also,

dass (falls nicht schon vorhanden) die Standard-Katalogkennung der evtl. angegebenen Benutzerkennung oder die des Aufrufers und/oder die Benutzerkennung des Aufrufers hinzugefügt wird (siehe Abschnitt „RecordFile").

4.1.1.2 Dateinamen im UFS-Dateisystem

Für den syntaktischen Aufbau sowie die semantische Definition gelten die Regeln wie bei der Klasse *java.io.File*. Auch die Begriffe „absoluter Pfadname“, „kanonischer Pfadname“ und „normalisierter Pfadname“ werden an diesen Schnittstellen in gleicher Weise benutzt.

4.1.2 Dateiarten

Folgende Dateiarten werden im DMS-Dateisystem zur Zeit unterstützt:

- SAM-Dateien mit fester und variabler Satzlänge.
- ISAM-Dateien mit fester und variabler Satzlänge.
- PAM-Dateien.

Das UFS-Dateisystem kennt bei Dateien keine Unterscheidung verschiedener Dateiarten, insbesondere gibt es keine definierten Dateiarten mit Satz/Block-Struktur. Lediglich der Inhalt einer Datei und die verarbeitenden Programme bestimmen, was damit geschehen kann bzw. wofür es gedacht ist (siehe Abschnitt „Zugriffsmethoden").

Wie auch bei *java.io.File* werden unter JRIO nur reguläre Dateien und Verzeichnisse unterstützt.

4.1.3 Zugriffsmethoden

Unter einer Zugriffsmethode versteht man üblicherweise eine Menge von Schnittstellen, die den Zugriff auf Daten aus Dateien ermöglichen und dabei eine bestimmte logische Sicht auf diese Daten bieten. Zumeist wird sich diese logische Sicht mehr oder weniger von der physikalischen Ablage in der Datei (dem Datenbehälter) unterscheiden. Hier interessieren jetzt Zugriffsmethoden, die eine satzorientierte Verarbeitung der Daten einer Datei ermöglichen, damit ist die logische Sicht eingeschränkt. Die hier betrachteten Zugriffsmethoden leisten also:

- Definition eines Datensatzes und die Abbildung dieser logischen Sicht der Daten auf eine physikalische Ablageform (Dateiart).
- Definition der Ordnung der Datensätze aus Sicht des Anwenders oder des Programmes, wobei diese Ordnung nicht notwendig etwas mit der physikalischen Ordnung der Daten in der Datei zu tun haben muss.
- Schnittstellen um Datensätze als Ganzes zu lesen und zu schreiben.

Eine besondere Art von Zugriffsmethoden sind elementare Zugriffsmethoden. Diese zeichnen sich dadurch aus, dass das Dateisystem (in dem sie wirken) Kenntnis von diesen Zugriffsmethoden hat und z.B. über Dateiarten bereits eine Zuordnung zwischen Datei und Zugriffsmethode möglich ist. Solche gibt es bekanntermaßen im DMS, wenn auch nicht umkehrbar eindeutig. Andere Dateisysteme (z.B. UFS) kennen nur eine einzige elementare Zugriffsmethode, die dann zumeist die rohe physikalische Sicht auf die Daten anbietet und haben konsequenterweise auch keine inhaltsorientierten Dateiarten.

Darüber hinaus kann es aber noch beliebige weitere Zugriffsmethoden geben, die dann meist unter Benutzung einer der elementaren Zugriffsmethoden implementiert sind und weitere logische Sichten auf die Daten bieten. Diesen Zugriffsmethoden ist ein Problem gemeinsam. Da das Dateisystem keine Kenntnis von ihnen hat, können Sie einer Datei auch nicht ansehen, ob und mit welcher dieser Zugriffsmethoden sie erfolgreich bearbeitet werden kann. Interpretationsfehler werden Sie also erst während der Verarbeitung erkennen, wenn überhaupt.

Im UFS gibt es keine elementare Zugriffsmethode, die eine satzorientierte Verarbeitung anbietet. Man kann sich aber z.B. folgende Zugriffsmethoden vorstellen:

- **TEXT** - Zugriffsmethode, die Textdateien als satzstrukturierte Dateien mit Sätzen variabler und unbegrenzter Länge ansieht. Der physikalische Satztrenner wäre dabei das Newline-Zeichen, das in der logischen Sicht ausgeblendet werden würde.
- **CISAM** - ISAM-ähnliche Zugriffsmethode für Unix-Dateisysteme

Im DMS gibt es mehrere elementare Zugriffsmethoden, von denen folgende direkt in JRIO unterstützt werden:

- **SAM** - sequentielle Zugriffsmethode
- **ISAM** - index-sequentielle Zugriffsmethode
- **UPAM** - blockorientierte Zugriffsmethode

Auch im DMS kennt man Zugriffsmethoden, die auf einer der elementaren Zugriffsmethoden aufsetzen und eine andere logische Sicht liefern. Prominentes Beispiel dafür ist eine auf ISAM aufsetzende Zugriffsmethode, die von etlichen Tools (Editoren, Compiler, ...) benutzt wird, um ISAM-Dateien für normale Texte nutzbar zu machen. Dafür werden ISAM Dateien mit Standard-Schlüsseln benutzt, wobei in der logischen Sicht diese Schlüssel (die ja bei ISAM Bestandteil des Datensatzes sind) ausgeblendet werden und beim Schreiben von Sätzen von der Zugriffsmethode generiert werden.

In den JRIO-Schnittstellen werden Sie den Zugriffsmethoden immer dann begegnen, wenn Sie entscheiden müssen, wie der Zugriff zu einer Datei erfolgen soll.

Derzeit wird in JRIO nur das Dateisystem DMS und darin nur die Zugriffsmethoden SAM, ISAM und UPAM unterstützt. Die Architektur von JRIO erlaubt aber zukünftig die Erweiterung um weitere Dateisysteme und Zugriffsmethoden, ohne dass die Benutzer-Schnittstellen dafür geändert werden müssten.

4.1.4 Zugriffsarten

Ausgangspunkt für das Design der JRIO-Schnittstellen ist eine von Dateisystemen und Zugriffsmethoden unabhängige abstrakte Sicht auf Art und Weise der Datenzugriffe, wie sie auch in der IBM-Implementierung zugrunde gelegt wird.

Aus Sicht der Anwendung kann die Art und Weise des Datenzugriffs dann in folgenden Zugriffsarten klassifiziert werden:

- Sequentieller Zugriff

  Lesender Zugriff auf Sätze/Seiten erfolgt in sequentieller Reihenfolge. Schreibender Zugriff erweitert die Datei am Ende.

- Wahlfreier Zugriff (Random)

  In einer Datei, die mit dieser Zugriffsart verarbeitet wird, kann beliebig auf einzelne Sätze positioniert werden, bevor gelesen oder geschrieben wird.

- Index-sequentieller Zugriff (Keyed)

  In einer Datei, die mit dieser Zugriffsart verarbeitet wird, können durch Angabe von Schlüsseln einzelne Sätze zum Lesen und/oder Schreiben ausgewählt werden.

4.1.5 Shared-Update-Verarbeitung

JRIO erlaubt die simultane, durch Sperren (Locks) synchronisierte Bearbeitung einer Datei durch mehrere Anwendungen (Shared-Update-Verarbeitung), sofern dies vom jeweiligen Dateisystem und der Zugriffsmethode unterstützt wird.

Diese Verarbeitungsart muss von der Anwendung beim Öffnen der Datei explizit eingestellt werden. Sie garantiert, dass die Verarbeitungsschritte (z.B. Schreiben, Löschen oder die Kombination Lesen und Zurückschreiben) durch Sperren abgesichert werden und daher von konkurrierenden Anwendungen nicht gestört werden können. Die Shared-Update-Verarbeitung kann Dateisystem-spezifischen Einschränkungen unterliegen. Sie kann z.B. für bestimmte Dateiarten oder Open-Modi nicht zulässig sein oder bestimmte Aktionen, wie z.B. das Vergrößern oder Verkleinern von Dateien, nicht gestatten.

Der von JRIO bei Shared-Update-Verarbeitung verwendete Sperrmechanismus arbeitet nach den folgenden Kriterien:

- satzorientiert:
  eine Anwendung sperrt *logisch* immer nur Sätze innerhalb einer Datei oder gibt sie frei. Bestimmte Dateisysteme oder Zugriffsmethoden können jedoch *physikalisch* ein größeres Sperrgranulat realisieren. Für die eigene Anwendung ist dies nicht sichtbar, aber konkurrierende Anwendungen können auf eine Sperre laufen, wenn sie auf einen Satz innerhalb des größeren Sperrgranulats zugreifen wollen, obwohl der angeforderte Satz selbst *logisch* nicht gesperrt ist.
- implizit:
  Sätze werden implizit beim Lesen, Schreiben oder Löschen gesperrt. Diese Sperre wird nach Abschluss des Schreib- oder Löschvorgangs auch implizit wieder aufgehoben. Für den Fall, dass ein Satz durch Lesen gesperrt wurde, aber nicht geschrieben werden soll, werden auch Methoden zum expliziten Entsperren angeboten
- deadlock-sicher:
  eine Anwendung kann pro Datei logisch immer nur einen Satz sperren (Deadlock-Sicherheit). Das Setzen einer Sperre für einen Vorgang führt implizit zur Freigabe einer eventuell bestehenden Sperre für einen anderen Satz. Einige Dateisysteme und Zugriffsmethoden können diese Deadlock-Sicherheit auch über Dateigrenzen hinweg realisieren, d.h. pro Anwendung wird nur eine Sperre gestattet- unabhängig von der jeweiligen Datei.

JRIO ermöglicht bei der Shared-Update-Verarbeitung die Steuerung des Verhaltens der Anwendung bei Zugriffskonflikten. Die Anwendung kann die sofortige Übergabe der Kontrolle verlangen (Parameter *NO_WAIT*). Dann wird bei Zugriffskonflikten eine entsprechende Exception (*RecordLockedException*) geworfen, oder sie kann als Thread (Parameter *THREAD_WAIT*) bzw. an der Systemschnittstelle (Parameter *APPLICATION_WAIT*) auf Zuteilung der Sperre warten. Die Wartezeit ist bei beiden Varianten unbegrenzt, d.h. die Anwendung wartet bis zum Erhalt der Sperre oder bis sie beendet wird. Das Warten als Thread bietet den Vorteil, dass andere Threads der Anwendung nicht blockiert sind. Im Extremfall kann es aber dazu führen, dass die Sperre von einer konkurrierenden Anwendung immer gerade in dem Moment gehalten wird, in dem die wartende Anwendung einen neuen Versuch unternimmt, obwohl die Sperre zwischenzeitlich verfügbar war. Nicht alle Dateisysteme bieten alle Varianten des Wartens an. Wenn aber eine Variante angeboten wird, gilt dafür die beschriebene Semantik.

4.1.6 Möglichkeiten und Einschränkungen der Zugriffsarten im DMS

Nicht alle Zugriffsarten sind mit allen Zugriffsmethoden/Dateiarten durchführbar. Die folgende Tabelle gibt eine Übersicht über den Zusammenhang zwischen Zugriffsarten und Zu-griffsmethoden/Dateiarten und die Möglichkeit zur Shared-Update-Verarbeitung im DMS-Dateisystem:

Zugriffsart	Zugriffsmethode SAM	Zugriffsmethode ISAM	Zugriffsmethode UPAM
Sequentiell	Lesen/Schreiben für SAM-Dateien. Physische Satzreihenfolge. Shared-Update-Verarbeitung ist nicht möglich.	Lesen/Schreiben für ISAM-Dateien. Satzreihenfolge durch Primär-Schlüssel bestimmt.Shared-Update-Verarbeitung ist für zum Lesen oder Erweitern geöffnete Dateien möglich.	Lesen/Schreiben für PAM-, SAM- und ISAM-Dateien. Physische Blockreihenfolge. Shared-Update-Verarbeitung ist nur für PAM-Dateien möglich, die zum Lesen geöffnet sind.
Random	Lesen/Schreiben für SAM-Dateien. Beim Überschreiben von Sätzen variabler Länge muss der zu schreibende Satz die gleiche Länge wie der zu überschreibende Satz haben. Shared-Update-Verarbeitung ist nicht möglich.	Nicht möglich.	Lesen/Schreiben für PAM-, SAM- und ISAM-Dateien. Shared-Update-Verarbeitung ist nur für PAM-Dateien möglich, die „INPUT“ oder „INOUT“ geöffnet sind.
Keyed	Nicht möglich.	Lesen/Schreiben für ISAM-Dateien. Shared-Update-Verarbeitung ist für „INPUT“ oder „INOUT“ geöffnete Dateien möglich. Nur die 1. öffnen-de Anwendung darf „OUTIN“ eröffnen.	Nicht möglich.

Tabelle 1: Übersicht über den Zusammenhang zwischen Zugriffsarten und Zugriffsmethoden/Dateiarten im DMS-Dateisystem

4.1.7 Treiber

JRIO hat ein dynamisches Treiber-Konzept, das sowohl die Dateisystem-Implementierungen, als auch die Implementierungen der verschiedenen Zugriffsmethoden von den Benutzerschnittstellen des JRIO entkoppelt. Neue Dateisystem-Treiber oder ZugriffsmethodenTreiber können hinzugefügt werden, ohne dass Benutzerschnittstellen dafür geändert werden müssten.

Beim Start einer Anwendung wird dynamisch ermittelt, welche Treiber für Dateisyteme und Zugriffsmethoden verfügbar sind. Diese werden dann bei Bedarf dynamisch geladen. Die dazugehörigen Schnittstellen (insbesondere das Treiber-API) sowie die Konfigurationsmechanismen sollen aber derzeit nicht für Benutzer zugänglich gemacht werden und werden daher hier auch nicht beschrieben.

4.1.8 Sicherheit

JRIO erlaubt ein feingranulares Sicherheitskonzept analog zu *java.io*.

Eine Anwendung, die JRIO benutzt und unter einem Security Manager läuft, wird z.B. mit folgendem Kommando gestartet:

```
java -Djava.security.manager <Anwendungsname>
```

Alle Zugriffe auf Dateien und Verzeichnisse der unterstützten Dateisysteme werden durch den Security Manager zunächst abgelehnt. Lediglich der Zugriff auf Dateien aus dem UFS-Verzeichnis, das die geladene Klasse enthält, wird gewährt.

UFS-Dateien unterliegen bei der Behandlung durch den Security Manager auch für JRIO den gleichen Mechanismen, die durch *java.io* geboten werden. Im Folgenden werden daher die Besonderheiten des DMS-Dateisystems beschrieben.

Um einer Anwendung auch unter dem Security Manager den Zugriff auf bestimmte Dateien und Verzeichnisse des DMS-Dateisystems zu erlauben, sind entsprechende „Berechtigungen“ (*Permissions*) in einem Policy-File zu erteilen. Der Mechanismus zur Auswahl des gültigen Policy-File unterscheidet sich nicht von dem in Java üblichen Verfahren, insbesondere kann das PolicyFile auch direkt spezifiziert werden:

```
java -Djava.security.manager
    -Djava.security.policy= <Policy-File> <Anwendung>
```

JRIO bietet zwei neue *Permissions* an, die im Policy-File erteilt werden können:

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

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

> Die Einträge im Policy-File können Sie mit dem *policytool*, aber auch mit einem normalen Editor vornehmen. In diesem Handbuch werden die Einträge so dargestellt, wie sie ein Editor anzeigt.
>
> Beachten Sie, dass das Policy-File in UTF8-Codierung vorliegen muss.

File-Permission

Die *com.fujitsu.ts.jrio.DMS.FilePermission* steuert den Zugriff auf Dateien und Verzeichnisse. Die Syntax eines Eintrags im Policy-File ist wie folgt:

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

Der *Dateibezeichner* ist dabei entweder ein gültiger BS2000-Verzeichnisname oder ein gültiger BS2000-Dateiname mit oder ohne Katalogkennung und/oder Benutzerkennung, d.h. eine Katalogkennung (in der Form ":catid:"), eine Benutzerkennung (in der Form "$userid.") oder eine Kombination davon (in der Form ":catid:$userid."). Das letzte Zeichen des Dateinamens kann ein „*“ sein . Dann bezieht sich die Zugriffserlaubnis auf alle Dateien, deren Namen mit dem String vor dem „*“ beginnen. In diesem Fall muss der Namensteil vor dem „*“ lediglich zu einem gültigen Dateinamen vervollständigt werden können. Für Katalog- und Benutzerkennung können auch „:*:“ bzw. „$*.“

angegeben werden, wodurch Sie die Erlaubnis für alle Katalogkennungen bzw. alle Benutzerkennungen erteilen. Die Abkürzung „$.“ für die System-Standard-Kennung ist erlaubt, aber nicht die Abkürzung „$file“ für „$.file“. Siehe Abschnitt „Dateinamen im DMS-Dateisystem".

Wenn keine Benutzerkennung explizit angegeben ist, bezieht sich die Erlaubnis auf Dateien unter der Benutzerkennung des Aufrufers (die der Anwendung daher nicht namentlich bekannt sein muss). Wenn keine Katalogkennung angegeben ist, bezieht sich die Erlaubnis auf Dateien der Standard-Katalogkennung der entsprechenden Benutzerkennung. Der String *<<ALL FILES>>* erlaubt den Zugriff auf alle Dateien und Verzeichnisse. In der mitgelieferten JAVADOC-Dokumentation für die Klasse *com.fujitsu.ts.jrio.DMS.FilePermission* finden Sie weitere Einzelheiten.

Die *Aktionsliste* ist eine komma-separierte Liste der für die Datei erlaubten Aktionen *read*, *write* und *delete*. Wenn die Erlaubnis für die Aktion in dieser Datei bzw. diesem Verzeichnis nicht erteilt ist, wird der Versuch eines entsprechenden Zugriffs mit einer *SecurityException* abgewiesen. Dies gilt auch für Auskunftsfunktionen, z.B. *list()* oder *listFiles()*, die eine Leseberechtigung für das zugrunde liegende Verzeichnis erfordern.

SystemInfo-Permission

Über die *com.fujitsu.ts.java.bs2000.SystemInfoPermission* wird innerhalb von JRIO gesteuert, welche Informationen über das DMS-Dateisystem sich die Anwendung verschaffen darf. Die Syntax hierzu lautet:

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

Dabei ist *Name* ein Wert aus der Menge *HomePubset*, *UserName*, *UserPubset*, *DefaultUserName*, *DefaultUserPubset* und *ForeignUserPubset* oder der String *<<ALL INFO>>*, mit dem die Erlaubnis für alle genannten Daten erteilt wird. Wenn die *Permission* erteilt wird, darf die Anwendung die entsprechenden Katalog- und Benutzerkennungen über die Schnittstellen *getCanonicalPath()*, *getCanonicalFile()*, *getAbsolutePath()* und *getAbsoluteFile()* der Klasse *RecordFile* ermitteln. Anderenfalls wird der Versuch mit einer *SecurityException* abgewiesen. Die mit *User...* beginnenden Namen beziehen sich auf die Kennung des Aufrufers, die mit *Default...* beginnenden Namen auf die System-Standard-Kennung und die mit *Foreign...* beginnenden Namen auf alle fremden Benutzerkennungen. Die Erlaubnis bezieht sich nur auf die Schnittstellen, die die entsprechenden Namen im Rahmen der Dateinamen-Vervollständigung offen legen, jedoch nicht auf den eigentlichen Zugriff auf Dateien aus diesen Katalog- oder Benutzerkennungen.

Beispiel

Einer Anwendung wird der Zugriff zu der Datei *HUGO* unter der Kennung des Aufrufers erlaubt, ohne dass die Anwendung die Erlaubnis hat, zu ermitteln, wie die Kennung des Aufrufers heißt:

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

Mit dieser Einstellung ist das Öffnen, das Lesen und das Schreiben der Datei erlaubt. Die Vervollständigung des Dateinamens z.B. mit *getCanonicalPath(...)* ist aber nicht gestattet.

4.2 API-Übersicht

Die öffentlichen Klassen, die die JRIO-Schnittstellen ausmachen, sind:

Klasse	Verwendung
Record	Repräsentiert einen Datensatz/-block
BufferOverflowException	Diese Exception wird immer dann ausgelöst, wenn beim Lesen von Sätzen das vom Benutzer bereitgestellte Record Objekt zu klein ist, um die Daten aufzunehmen.
RecordLockedException	Diese Exception wird bei Shared-Update-Verarbeitung ausgelöst, wenn auf einen Satz zugegriffen wird, der von einer anderen Anwendung gesperrt ist und der Anwender spezifiziert hat, dass auf die Zuteilung der Sperre nicht gewartet werden soll.
RecordNotLockedException	Diese Exception wird bei Shared-Update-Verarbeitung ausgelöst, wenn auf einen Satz mit einer Methode zugegriffen wird, die das vorherige Sperren des Satzes voraussetzt, diese Sperre aber nicht (mehr) besteht.
RecordFile	Repräsentiert eine Datei mit Satz-/Block-Struktur (siehe *java.io.File*).
RecordFileFilter	Interface für die Implementierung benutzereigener Klassen, die als Filter in der Methode *listFiles()* der Klasse *RecordFile* genutzt werden können (siehe *java.io.FileFilter*).
RecordFilenameFilter	Interface für die Implementierung benutzereigener Klassen, die als Filter in der Methode *list()* der Klasse *RecordFile* genutzt werden können (siehe *java.io.FilenameFilter*).
AccessParameter	Repräsentiert die allgemeinen Parameter, die für den Zugriff auf eine Datei mit Satz-/Block-Struktur unter Benutzung einer bestimmten Zugriffsmethode benötigt werden.
DMS/AccessParameterSAM	Repräsentiert eine Auswahl der Parameter, die für einen Zugriff auf eine Datei (insbesondere Erzeugung) mit der Zugriffsmethode SAM im DMS benötigt werden.
DMS/AccessParameterISAM	Repräsentiert eine Auswahl der Parameter, die für einen Zugriff auf eine Datei (insbesondere Erzeugung) mit der Zugriffsmethode ISAM im DMS benötigt werden.
DMS/AccessParameterUPAM	Repräsentiert eine Auswahl der Parameter, die für einen Zugriff auf eine Datei (insbesondere Erzeugung) mit der Zugriffsmethode UPAM im DMS benötigt werden
DMS/FilePermission	Erlaubt die feingranulare Zuteilung von Zugriffsrechten für Dateien und Verzeichnisse im DMS-Dateisystem. Die Klasse wird dabei üblicherweise nur im Rahmen von Einträgen im Policy-File verwendet.
InputRecordStream	Abstrakte Basisklasse für *FileInputRecordStream* und *ArrayInputRecordStream* sowie benutzerimplementierte Input-Klassen (siehe *java.io.InputStream*).

ArrayInputRecordStream	Klasse zum sequentiellen Lesen von Sätzen aus einem Array von Sätzen (siehe *java.io.ByteArrayInputStream*)
FileInputRecordStream	Repräsentiert eine für sequentielles Lesen geöffnete Datei mit Satz-/Block-Struktur (siehe *java.io.FileInputStream*)
OutputRecordStream	Abstrakte Basisklasse für *FileOutputRecordStream* und *ArrayOutputRecordStream* sowie benutzerimplementierte Output-Klassen (siehe *java.io.OutputStream*)
ArrayOutputRecordStream	Klasse zum sequentiellen Schreiben von Sätzen in ein Array von Sätzen (siehe *java.io.ByteArrayOutputStream*)
FileOutputRecordStream	Repräsentiert eine für sequentielles Schreiben geöffnete Datei mit Satz-/Block-Struktur (siehe *java.io.FileOutputStream*)
RandomAccessRecordFile	Repräsentiert eine für wahlfreien Zugriff geöffnete Datei mit Satz-/Block-Struktur (siehe *java.io.RandomAccessFile*)
KeyedAccessRecordFile	Repräsentiert eine für schlüsselorientierten Zugriff geöffnete Datei mit Satz-/Block-Struktur
KeyDescriptor	Beschreibt einen Satzschlüssel einer index-sequentiellen Datei
DMS /PrimaryKeyDescriptorISAM	Beschreibt den Primärschlüssel einer ISAM-Datei
DMS /SecondaryKeyDescriptorISAM	Beschreibt einen Sekundärschlüssel einer ISAM-Datei
KeyValue	Repräsentiert den konkreten Wert eines Satzschlüssels

Tabelle 2: Öffentliche Klassen, die JRIO-Schnittstellen ausmachen

Die folgenden Abschnitte beschreiben die wesentlichen der genannten Klassen des Paketes JRIO mit ihren wichtigsten und gebräuchlichsten Methoden und Feldern. Eine vollständige Schnittstellenbeschreibung befindet sich in der mitgelieferten *javadoc*-Dokumentation (siehe im Installationsverzeichnis unter *doc/jrio*).

4.2.1 Record

Ein *Record*-Objekt repräsentiert einen logischen Satz einer Datei und besteht aus einem Satzpuffer, der den eigentlichen Datensatz enthält und der getrennt verwalteten Länge der Daten innerhalb des Satzpuffers.

Die Klasse *Record* bietet Methoden an, um auf die Daten im Satzpuffer und deren Länge zuzugreifen, diese zu setzen oder auch zu verändern. Es werden keine Methoden für den Zugriff auf numerische Datenfelder angeboten, diese kann der Anwender auf Basis der angebotenen Methoden selbst implementieren.

Ein *Record*-Objekt wird typischerweise verwendet, um die Daten der satz- oder seitenweisen Zugriffsoperationen in Dateien abzulegen bzw. aus Dateien wiederzugewinnen. Es ist serialisierbar und kann deshalb für entfernte Methoden-Aufrufe (RMI) verwendet werden. Außerdem wird das *Cloneable* Interface implementiert.

Positionen innerhalb eines Satzes werden mit Position 0 beginnend gezählt (das erste Datenbyte eines Satzes hat also die Position 0 usw.). Ein logischer Datensatz einer Datei enthält nur die Nutzinformationen, während der physikalisch in der Datei abgelegte Datensatz zusätzliche Meta-Informationen (z.B. Satzlänge) enthalten kann. Daher kann sich die Nummerierung von Satzpositionen z.B. an den DMS-Makro-Schnittstellen des BS2000 (liefern den physikalischen Satz) von der an den JRIO-Schnittstellen (liefern den logischen Satz) unterscheiden.

4.2.1.1 Konstruktoren

Beim Erzeugen eines *Record*-Objektes kann entweder ein leerer Satzpuffer einer gewünschten Größe angelegt, oder ein vom Benutzer bereitgestellter Puffer verwendet werden. Enthält dieser Puffer bereits Daten, so kann die Länge der Daten mit übergeben werden.

Typischerweise sollten Sie die Größe des Satzpuffers so wählen, dass auch der längste erwartete Datensatz darin Platz findet. Das *Record*-Objekt kann dann bei der Ein-/Ausgabe immer wieder verwendet werden, wenn der alte Inhalt nicht mehr benötigt wird, statt immer wieder neue Instanzen zu erzeugen.

Zusätzlich steht ein *Copy*-Konstruktor zur Verfügung, der aus den Daten eines anderen Records ein neues *Record*-Objekt erzeugt. Eine Eins-zu-Eins-Kopie eines *Record*-Objektes kann mit der *clone()*-Methode erzeugt werden.

4.2.1.2 Allgemeine Methoden

Die Methode *getBuffer()* liefert den Satzpuffer des Satzes. Diesen können Sie verwenden, um mit Hilfe anderer Klassen und Methoden den Inhalt zu verarbeiten bzw. bereitzustellen. Beachten Sie, dass Manipulationen auf diesem Satzpuffer das Objekt verändern, aus dem der Satzpuffer stammt, da es sich nicht um eine Kopie der Daten handelt. Die aktuelle Länge der Daten innerhalb des Satzpuffers kann mit der Methode *getDataLength()* ermittelt werden.

Ein Satzpuffer kann mit den Methoden *setBuffer()* ersetzt werden. Enthält der übergebene Puffer des Benutzers bereits Daten, so kann die Datenlänge mit übergeben werden.

Mit der Methode *setDataLength()* kann der Benutzer selbst den Füllstand des Satzpuffers festlegen. Es wird nicht geprüft, ob die im Satzpuffer befindlichen Daten sinnvoll sind.

4.2.1.3 Methoden zum Extrahieren der Daten eines Satzes

Mit der Methode *getByteData()* werden die kompletten Daten eines Satzes binär (als Bytes) geliefert.

Die Methoden der Familie *getStringData()* liefern die kompletten Daten eines Satzes als Text (String) interpretiert.

Wenn kein Encoding für die Umwandlung der Daten in Text vom Benutzer angegeben wurde, so wird das systemabhängige Standard-Encoding (im BS2000 ist der Standardwert *OSD_EBCDIC_DF04_1*) verwendet.

4.2.1.4 Methoden zum Extrahieren der Datenfelder eines Satzes

Mit den verschiedenen Methoden der Familie *getByteField()* werden die Daten eines spezifizierten Datenfeldes (definiert durch Position und Länge innerhalb des Datensatzes) binär (als Bytes) geliefert.

Die Methoden der Familie *getStringField()* liefern die Daten eines spezifizierten Datenfeldes als Text (String) interpretiert.

Wenn kein Encoding für die Umwandlung der Daten in Text vom Benutzer angegeben wurde, so wird das systemabhängige Standard-Encoding (im BS2000 ist der Standardwert *OSD_EBCDIC_DF04_1*) verwendet.

Die Methode *getKeyField()* liefert anhand einer Schlüsselbeschreibung den Inhalt eines Schlüsselfeldes als Schlüsselwert.

4.2.1.5 Methoden zum Füllen eines Satzes mit Daten

Die Methoden *setByteData()* füllen einen Satz komplett mit binären Daten (Bytes), der alte Inhalt geht dabei verloren. Die Datenlänge des Satzes entspricht anschließend genau der Länge der eingefüllten Daten.

Die Methoden der Familie *setStringData()* füllen einen Satz komplett mit Text-daten (String). Die Datenlänge des Satzes entspricht anschließend genau der Länge der eingefüllten Daten.

Wenn kein Encoding für die Umwandlung von Text in Daten vom Benutzer angegeben wurde, so wird das systemabhängige Standard-Encoding (im BS2000 ist der Standardwert *OSD_EBCDIC_DF04_1*) verwendet.

4.2.1.6 Methoden zum Füllen von Datenfeldern eines Satzes

Die verschiedenen Methoden der Familie *setByteField()* füllen binäre Daten (Bytes) in ein spezifiziertes Datenfeld (definiert durch Position und Länge) eines Satzes.

Diese Methoden aktualisieren die Datenlänge, wenn beim Füllen der Datenfelder der Satz verlängert wurde. Sind die Daten kürzer als das ausgewählte Datenfeld, so kann optional der Rest mit einem Füllbyte ergänzt werden. Sind die Daten länger als das ausgewählte Datenfeld, so werden die Daten nur in der Länge des Datenfeldes in den Satzpuffer übernommen.

Die Methoden der Familie *setStringField()* füllen Textdaten (String) in ein spezifiziertes Datenfeld. Wenn kein Encoding für die Umwandlung von Text in Daten vom Benutzer angegeben wurde, so wird das systemabhängige Standard-Encoding (im BS2000 ist der Standardwert *OSD_EBCDIC_DF04_1*) verwendet.

Diese Methoden aktualisieren die Datenlänge, wenn beim Füllen der Datenfelder der Satz verlängert wurde. Sind die Daten kürzer als das ausgewählte Datenfeld, so kann optional der Rest mit Leerzeichen ergänzt werden. Sind die Daten länger als das ausgewählte Datenfeld, so werden die Daten nur in der Länge des Datenfeldes in den Satzpuffer übernommen.

Die Methode *setKeyField()* füllt ein Schlüsselfeld eines Satzes mit einem konkreten Schlüsselwert.

4.2.2 RecordFile

Die Klasse *RecordFile* spielt für die satzorientierte Ein-/Ausgabe etwa die gleiche Rolle, wie die Klasse *java.io.File* für die normale Java-I/O. Sie beschreibt die Objekte des/der zugrunde liegenden Dateisystems/-systeme, also normalerweise Dateien und Verzeichnisse.

Anders als die Klasse *java.io.File* werden durch die Klasse *RecordFile* tatsächlich verschiedene Dateisysteme unterstützt und nicht nur eines. Daher besteht ein *RecordFile*-Objekt stets aus einem Pfadnamen (Datei- oder Verzeichnisname) und einem assoziierten Dateisystem (DMS, UFS, ...).

Es kann also insbesondere in verschiedenen Dateisystemen gleichnamige Objekte geben. Man kann sich im BS2000 durchaus vorstellen, dass es eine Datei namens *HALLO* sowohl im UFS (Posix-Dateisystem), im DMS (BS2000-Dateisystem) als auch im LMS (als Element einer PLAM-Bibliothek) geben kann. Dieser Ansatz spiegelt also die tatsächlichen Gegebenheiten des BS2000 besser wider, als der monolithische Ansatz von *java.io.File* (siehe Abschnitt „Dateisysteme").

Die Klasse *RecordFile* stellt (wie auch *java.io.File*) Methoden und Felder bereit, um Analysen und Transformationen auf dem Pfadnamen vorzunehmen. Diese sind für jedes unterstützte Dateisystem möglicherweise anders definiert. Für diese Operationen ist es meist unerheblich, ob es im Dateisystem tatsächlich eine Datei oder ein Verzeichnis mit dem vorliegenden Namen gibt, denn es wird zumeist nicht auf das zugrunde liegende Dateisystem zugegriffen.

Darüber hinaus stellt die Klasse *RecordFile* Methoden zur Verfügung, um auf die Eigenschaften existierender Dateien und Verzeichnisse zuzugreifen und diese möglicherweise zu verändern.

Außerdem werden durch die Klasse *RecordFile* Methoden angeboten, um typische Dateisystem-Operationen auszuführen. Dazu gehören das Umbenennen und Löschen von existierenden Dateien und Verzeichnissen, das Neuanlegen von noch nicht existierenden Dateien und Verzeichnissen sowie das Auflisten von Verzeichnis-Inhalten.

Alle Methoden, die tatsächlich auf das Dateisystem zugreifen, unterliegen den Restriktionen des aktiven Security Managers und lösen entsprechende Ausnahmen aus, wenn der Zugriff auf das Dateisystem eingeschränkt ist (siehe Abschnitt „Sicherheit").

In den folgenden Abschnitten wird auf Besonderheiten des UFS Dateisystems meist nicht verwiesen. Dann gilt für das UFS Dateisystem immer das gleiche, was bei *java.io.File* für das Unix-Dateisystem gilt.

4.2.2.1 Prinzipieller Aufbau eines Dateinamens

Im allgemeinen besteht ein Pfadname in jedem unterstützten Dateisystem aus einem eventuell vorhandenen Dateisystempräfix und keinem Namensteil oder einer Folge von einem oder mehreren Namensteilen, die durch Trennzeichen getrennt sein können. Jeder Namensteil in einem Pfadnamen, außer dem letzten, bezeichnet ein Verzeichnis. Der letzte Namensteil kann entweder ein Verzeichnis oder eine Datei bezeichnen. Der leere Pfadname hat kein Präfix und eine leere Folge von Namensteilen. Ob ein leerer Pfadname erlaubt ist und wie die Semantik des leeren Pfadnamens ist, ist abhängig vom Dateisystem.

Das oder die Dateisystempräfixe sind Dateisystem-spezifisch definiert. Es ist garantiert, dass alle von *listRoots()* gelieferten Root-Verzeichnisse zulässige Dateisystempräfixe sind. Im DMS-Dateisystem wird jede Katalogkennung als ein Dateisystempräfix in diesem Sinne interpretiert, ungeachtet der Existenz dieser Katalogkennung im Dateisystem. Im UFS-Dateisystem ist die Wurzel „/“ das einzige Dateisystempräfix.

Wenn mehrstufige Namen in einem Dateisystem erlaubt sind, wird meist (aber nicht immer) ein Trennzeichen definiert sein, mit dem die Namensteile getrennt werden (z.B. „/“ in UFS). Ob und wie viele Namensteile erlaubt sind, und wie sie getrennt werden, definiert aber letztlich das konkrete Dateisystem.

Für Pfadnamen im DMS-Dateisystem gibt es kein definiertes Trennzeichen. Außer dem Dateisystempräfix (die Katalogkennung mit Doppelpunkten ':' am Anfang und Ende) kann der Pfadname noch maximal zwei Namensteile enthalten: eine Benutzerkennung (mit Dollar '$' am Anfang und Punkt '.' am Ende) und/oder einen Dateinamen. Auch wenn beide Teile im Pfadnamen enthalten sind, gibt es kein zusätzliches Trennzeichen zwischen den beiden.

Für Pfadnamen im UFS-Dateisystem werden die gleichen Namensregeln benutzt wie bei *java.io.File*.

4.2.2.2 Konstruktoren

Ein *RecordFile*-Objekt wird aus einem gegebenen Pfadnamen und einer Dateisystem-Spezifikation gebildet. Dabei wird geprüft, ob der gegebene Pfadname den syntaktischen Vorschriften des angegebenen Dateisystems genügt und eine so genannte Normalisierung des Pfadnamens durchgeführt. Was das im einzelnen bedeutet wird für jedes unterstützte Dateisystem getrennt festgelegt (siehe Abschnitt „Dateisysteme"). Dieser normalisierte Pfadname ist dann der Name dieses Objektes und Basis aller Operationen darauf.

Es gibt Konstruktor-Varianten, welche die Angaben des Pfadnamens in unterschiedlicher Form erlauben, entweder einfach als einzelner String oder getrennt als zwei Strings, die den Verzeichnisanteil und den Dateinamensanteil des Pfadnamen bilden oder aber als *RecordFile*-Objekt für den Verzeichnisanteil und String für den Dateinamensanteil. Im letzteren Fall entfällt die Dateisystem-Spezifikation, weil das *RecordFile*-Objekt für den Verzeichnisanteil diese schon implizit enthält.

4.2.2.3 Felder

Das Trennzeichen *separatorChar* bzw. *separator* (in String Form) ist Dateisystem-spezifisch definiert. Normalerweise wird dieses Trennzeichen verwendet, um verschiedene Namensbestandteile innerhalb eines Pfadnamens zu trennen.

Besonderheiten des DMS-Dateisystems

Das DMS-Dateisystem kennt kein Trennzeichen in diesem Sinne. Deswegen wird für *separatorChar* das Null-Zeichen und für *separator* ein leerer String verwendet. Das erfordert aber Vorsicht bei der Benutzung, da das Null-Zeichen ein definiertes Zeichen auch innerhalb von Zeichenketten ist.

Das Trennzeichen *pathSeparatorChar* bzw. *pathSeparator* (in String Form) ist auch Dateisystem-spezifisch definiert. Dieses Trennzeichen wird verwendet, um bei der Spezifikation von Pfadnamens-Listen die einzelnen Pfadnamen voneinander zu trennen.

Besonderheiten des DMS-Dateisystems

Das Trennzeichen für Pfadnamens-Listen ist das Komma „,“.

Anders als in *java.io.File* sind die Trennzeichen keine statischen Felder, da hier mehrere Dateisysteme unterstützt werden. Die Trennzeichen werden bei der Instanziierung eines *RecordFile*-Objektes von dem darunter liegenden Datei-system initialisiert.

4.2.2.4 Allgemeine Methoden

Die Methode *getPath()* liefert den Pfadnamen dieses RecordFile-Objektes. Die Methode *getFileSystem()* liefert den Namen des mit dem Pfadnamen assoziierten Dateisystems. Für das DMS Dateisystem wird der String „DMS“ und für das UFS Dateisystem (wird momentan von JRIO nicht unterstützt) wird der String „UFS“ geliefert.

4.2.2.5 Methoden zur Analyse und Transformation von Pfadnamen

Die Methode *getName()* liefert den letzten Namensteil des Pfadnamens des *RecordFile*-Objektes. Das Ergebnis wird durch Streichen eines eventuell vorhandenen Dateisystempräfixes und jedes Namensteils, außer des Letzten, gebildet. Besteht der Pfadname lediglich aus einem Namensteil und handelt sich es dabei nicht um ein Dateisystempräfix, so wird der Pfadname des Objektes geliefert. Wenn der Pfadname leer ist oder nur aus dem Dateisystempräfix besteht, wird ein Leerstring geliefert.

Besonderheiten des DMS-Dateisystems

Das Dateisystempräfix ist die Katalogkennung.

Beispiel

Name	Ergebnis
: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:	Leerstring ""
$USER.	$USER.

Die Methode *getParent()* liefert den Elternteil dieses Pfadnamens als String oder den Rückgabewert null, wenn der Pfadname keinen Elternteil besitzt. Der Elternteil eines Pfadnamens besteht aus dem Dateisystempräfix, falls vorhanden und aus jedem Namensbestandteil in der Namensfolge des Pfadnamens, außer dem Letzten. Wenn die Namensfolge leer ist, dann hat der Pfadname keinen Elternteil.

Besonderheiten des DMS-Dateisystems

Das Dateisystempräfix ist die Katalogkennung.

Beispiel

Name	Ergebnis
: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

Die Methode *getParentFile()* liefert wie die Methode *getParent()* den Elternteil dieses Pfadnamens, aber als ein *RecordFile*-Objekt. Wenn der Pfadnamen kein Elternteil hat, wird null geliefert.

Die Methode *isAbsolute()* liefert true, wenn der Pfadname dieses *RecordFile*-Objektes ein absoluter Pfadname ist. Was ein absoluter Pfadname ist, ist Dateisystem-spezifisch festgelegt (siehe Abschnitt „Dateisysteme").

Beispiel

Name	Ergebnis
:catid:$userid.	true
:catid:$.	true
$userid.	false
$.	false
:catid:	true
$.HALLO	false
$USER.HALLO	false
:JAVA:$.HALLO.JAVA	true

Die Methode *getAbsolutePath()* gibt die absolute Form des Pfadnamens dieses *RecordFile*-Objektes als String zurück. Wurde das *RecordFile*-Objekt mit Hilfe eines absoluten Pfadnamens konstruiert, so wird einfach dieser Name geliefert. Wenn das nicht der Fall ist, wird Dateisystem-spezifisch der Name ergänzt (siehe Abschnitt „Dateisysteme").

Besonderheiten des DMS-Dateisystems

Es ist im DMS unter Umständen nicht möglich, zu einem syntaktisch korrekten Pfadnamen den absoluten Pfadnamen zu bilden. So ist z.B. ein aus 42 Zeichen bestehender Dateiname mit einer 5-stelligen Benutzerkennung syntaktisch korrekt. Wird er aber mit einer aus 4 Zeichen bestehenden Katalogkennung ergänzt, entsteht ein syntaktisch falscher (zu langer) Pfadname.

Die Methode *getAbsoluteFile()* liefert die absolute Form des Pfadnamens dieses *RecordFile*-Objektes als *RecordFile* -Objekt.

Die Methode *getCanonicalPath()* gibt die kanonische Form des Pfadnamens dieses *RecordFile*-Objektes als String zurück. Ein kanonischer Pfadname ist so-wohl absolut als auch eindeutig. Die genaue Definition der kanonischen Form ist abhängig vom Dateisystem (siehe Abschnitt „Dateisysteme").

Besonderheiten des DMS-Dateisystems

> Es ist im DMS unter Umständen nicht möglich, zu einem syntaktisch korrekten Pfadnamen den kanonischen Pfadnamen zu bilden. So ist z.B. ein aus 42 Zeichen bestehender Dateiname mit einer 4-stelligen Katalogkennung syntaktisch korrekt. Wird er aber mit einer aus 5 oder mehr Zeichen bestehenden Benutzerkennung ergänzt, entsteht ein syntaktisch falscher (zu langer) Pfadname.

Die Methode *getCanonicalFile()* liefert die kanonische Form des Pfadnamens dieses *RecordFile*-Objektes als *RecordFile*-Objekt.

Die Vervollständigung eines Dateinamens durch die o.g. Methoden gibt der Anwendung Einblick in die Struktur des Dateisystems und unterliegt daher der Überwachung durch einen aktiven Security Manager. Unter Umständen wird diese Vervollständigung mit einer entsprechenden *Exception* abgewiesen (siehe Abschnitt „Sicherheit").

Die Methode *compareTo()* vergleicht zwei Pfadnamen lexikographisch. Wenn die zwei Pfadnamen unterschiedlichen Dateisystemen angehören, werden zuerst die Dateisystemnamen verglichen.

Die Methode *equals()* vergleicht zwei Pfadnamen. Sie liefert nur true, wenn es sich bei dem angegebenen Objekt um ein *RecordFile*-Objekt handelt, das dem gleichen Dateisystem zugeordnet ist, und wenn die Pfadnamen beider Objekte (im Sinne von *compareTo()*) gleich sind. Die Gleichheit wird auf Basis der Pfadnamen und nicht der Datei oder des Verzeichnisses im zugrunde liegenden Dateisystem geprüft, d.h. wenn verschiedene Namen die gleiche existierende Datei bezeichnen, wird trotzdem false geliefert.

Die Methode *hashcode()* berechnet einen Hashcode aus den Zeichen des Pfad-namens und des Dateisystemnamens. Zwei *RecordFile*-Objekte mit den gleichen Pfadnamen und dem gleichen Dateisystem haben auch den gleichen Hashcode. Jedoch zwei *RecordFile*-Objekte mit den gleichen Hashcodes müssen nicht notwendigerweise den gleichen Pfadnamen haben.

4.2.2.6 Methoden zur Abfrage von Datei- und Verzeichniseigenschaften

Die Methode *exists()* prüft, ob die Datei oder das Verzeichnis im Dateisystem existiert. Viele der angebotenen Methoden sind nur sinnvoll anwendbar, wenn die Datei oder das Verzeichnis existiert und sichtbar ist (z.B. sind Dateien in fremden Benutzerkennungen nicht immer für den Aufrufer sichtbar).

Besonderheiten des DMS-Dateisystems

Eine Datei wird als existierend angesehen, wenn sie schon einmal geöffnet war, das bedeutet, dass ein Katalogeintrag für die Existenz einer Datei nicht genügt. Ein Verzeichnis existiert, wenn die angegebene Katalogkennung und/oder Benutzerkennung im Dateisystem zugreifbar ist.

Die Methode *canRead()* prüft, ob die Datei oder das Verzeichnis für dieses Objekt existiert (im Sinne von *exists()*) und für den Aufrufer lesbar ist.

Besonderheiten des DMS-Dateisystems

Für existierende Verzeichnisse wird immer true geliefert.

Die Methode *canWrite()* prüft, ob die Datei oder das Verzeichnis für dieses Objekt existiert (im Sinne von *exists()*) und für den Aufrufer schreibbar ist.

Besonderheiten des DMS-Dateisystems

Liefert für ein Verzeichnis, das nur aus der Katalogkennung besteht, immer false, wenn der Aufrufer nicht privilegiert ist, ebenso für fremde Benutzerkennungen.

Die Methode *isDirectory()* liefert true, wenn es sich um ein existierendes Verzeichnis im zugehörigen Dateisystem handelt.

Die Methode *isFile()* liefert true, wenn es sich um eine normale Datei im zuge-hörigen Dateisystem handelt. Eine Datei ist normal, wenn sie kein Verzeichnis ist und zusätzlich Dateisystem-spezifische Kriterien erfüllt (z.B. Gerätedateien im UFS sind keine normale Dateien). Jede Datei, die von einer Java-Anwendung erzeugt wurde und die kein Verzeichnis ist, ist garantiert eine normale Datei.

Mit der Methode *isHidden()* kann festgestellt werden, ob die Datei bzw. das Verzeichnis im Dateisystem versteckt ist. Was genau versteckt bedeutet, ist Dateisystem-spezifisch definiert.

Besonderheiten des DMS-Dateisystems

Temporäre Dateien im DMS-Sinne werden immer als versteckt angesehen.

Die Methode *lastModified()* liefert den Zeitpunkt der letzten Änderung der Datei oder des Verzeichnisses, wenn das Dateisystem das unterstützt.

Besonderheiten des DMS-Dateisystems

Verzeichnisse haben kein eigenes Modifikationsdatum, deshalb wird immer 0 geliefert.

Die Methode *length()* liefert die Größe einer Datei oder eines Verzeichnisses. Es ist Dateisystem-spezifisch, wie die Größe eines Verzeichnisses definiert ist.

Besonderheiten des DMS-Dateisystems

Für Dateien wird die Anzahl der benutzen (nicht der reservierten) PAM-Seiten der Datei multipliziert mit 2048 und für Verzeichnisse wird immer der Wert 0 geliefert.

Die Methode *getAccessParameter()* liefert die Parameter für den Zugriff auf diese Datei mit der angegebenen Zugriffsmethode. Das gelieferte *AccessParameter*-Objekt kann z.B. benutzt werden, um eine neue Datei mit den gleichen Parametern zu erzeugen und wird intern für den Dateizugriff benutzt.

Die statische Methode *getDefaultAccessParameter()* liefert die Standard-Parameter für die gegebene Zugriffsmethode in dem gegebenen Dateisystem. Diese Parameter kann der Anwender dann evtl. noch verändern und damit neue Dateien erzeugen.

Die Methode *getPreferredAccessMethod()* liefert den Namen der bevorzugten Zugriffsmethode für eine existierende Datei im zugehörigen Dateisystem. Es wird nicht garantiert, dass die Datei mit dieser Zugriffsmethode erzeugt wurde, insbesondere, wenn die Zugriffsmethoden in diesem Dateisystem nur eine logische Sicht auf den Inhalt der Datei und keine abfragbare Dateieigenschaft sind.

Die Methode *getAllowedAccessMethods()* liefert eine Liste der Namen der erlaubten Zugriffsmethoden für eine existierende Datei im zugehörigen Dateisystem. Es wird nicht garantiert, dass die Datei mit einer dieser Zugriffsmethoden erzeugt wurde, insbesondere, wenn die Zugriffsmethoden in diesem Datei-system nur eine logische Sicht auf den Dateiinhalt und keine abfragbare Datei-Eigenschaften sind.

Die statische Methode *getAllAccessMethods()* liefert eine Liste der Namen aller unterstützten Zugriffsmethoden des angegebenen Dateisystems.

4.2.2.7 Methoden zur Änderung von Datei- und Verzeichnis-Eigenschaften

Die Methode *setLastModified()* setzt das Modifikationsdatum auf den angegebenen Wert, wenn das Dateisystem dies unterstützt.

Besonderheiten des DMS-Dateisystems

Das Modifikationsdatum kann nicht gesetzt werden.

Die Methode *setReadOnly()* verändert die Eigenschaften der Datei so, dass nur Lese-Operationen erlaubt sind.

Besonderheiten des DMS-Dateisystems

Für Dateien wird die Dateieigenschaft *ACCESS* auf den Wert *READ* gesetzt. Für Verzeichnisse und temporäre Dateien im DMS-Sinne kann diese Eigenschaft nicht gesetzt werden.

4.2.2.8 Methoden zum Erzeugen von Dateien und Verzeichnissen

Die Methoden *createNewFile()* erzeugen eine neue Datei mit dem Pfadnamen des *RecordFile*-Objektes unter Verwendung der angegebenen Zugriffsparameter oder der StandardParameter der angegebenen Zugriffsmethode.

Besonderheiten des DMS-Dateisystems

Beim Erzeugen einer Datei wird diese exklusiv geöffnet und wieder geschlossen. Eine eventuell in den Zugriffsparametern gesetzte Shared-Update-Option wird ignoriert. Die Shared-Update-Option wird erst wirksam, wenn die Datei zur Verarbeitung geöffnet wird.

Die statischen Methoden *createTempFile()* bieten die Möglichkeit temporäre Dateien unter Verwendung der angegebenen Zugriffsparameter anzulegen. Eine temporäre Datei hat einen generierten Namen, der in dem Dateisystem garantiert noch nicht existiert. Die Datei wird entweder in dem angegebenen Verzeichnis, oder wenn kein Verzeichnis angegeben wurde, in einem Dateisystem-spezifischen Verzeichnis, angelegt. Die Datei wird nicht automatisch gelöscht, sondern der Benutzer muss die Methode *deleteOnExit()* rufen, wenn die Datei beim Beenden der Anwendung gelöscht werden soll. Der Name wird aus den Präfix- und Suffixangaben des Benutzers sowie einem Dateisystem-spezifisch generierten String gebildet.

Besonderheiten des DMS-Dateisystems

Es werden keine temporären Dateien im üblichen DMS-Sinne erzeugt, sondern immer permanente Dateien, die der Benutzer auch selbst wieder löschen muss. Mit dem Dateisystem-spezifischen Verzeichnis ist immer die Standard-Katalogkennung des aufrufenden Benutzers gemeint. Der Verzeichnisparameter kann im DMS-Dateisystem benutzt werden, um die temporäre Datei auf einer anderen Katalogkennung als der Standard-Katalogkennung zu erzeugen.

Beim Erzeugen einer temporären Datei wird diese exklusiv geöffnet und wieder geschlossen. Eine eventuell in den Zugriffsparametern gesetzte Shared-Update-Option wird ignoriert. Die Shared-Update-Option wird erst wirksam, wenn die Datei zur Verarbeitung geöffnet wird.

Die Methode *mkdir()* legt ein Verzeichnis mit dem Namen dieses Objektes in dem zugehörigen Dateisystem an.

Besonderheiten des DMS-Dateisystems

Es können keine Verzeichnisse angelegt werden.

Die Methode *mkdirs()* legt ein Verzeichnis inklusive aller notwendigen Vaterverzeichnisse in dem zugehörigen Dateisystem an.

Besonderheiten des DMS-Dateisystems

Es können keine Verzeichnisse angelegt werden.

4.2.2.9 Methoden zum Löschen und Umbenennen von Dateien und Verzeichnissen

Die Methode *renameTo()* benennt diese Datei bzw. dieses Verzeichnis in den an-gegebenen Pfadnamen um. Die Zieldatei darf noch nicht existieren. Eine Umbenennung ist nur innerhalb eines Dateisystems möglich.

> Das Record-File Objekt selbst bleibt dem alten Namen zugeordnet, repräsentiert anschließend also eventuell eine nicht existierende Datei.

Besonderheiten des DMS-Dateisystems

Das Umbenennen von Verzeichnissen ist nicht möglich. Das Umbenennen von Dateien kann nur mit der gleichen Katalog- und Benutzerkennung durchgeführt werden.

Die Methode *delete()* löscht diese Datei bzw. dieses Verzeichnis im zugehörigen Dateisystem. Wenn der Pfadname ein Verzeichnis bezeichnet, kann dieses nur gelöscht werden, wenn es leer ist.

Besonderheiten des DMS-Dateisystems

Verzeichnisse können nicht gelöscht werden.

Die Methode *deleteOnExit()* sorgt dafür, das diese Datei bzw. dieses Verzeichnis im zugehörigen Dateisystem beim Beenden der Anwendung gelöscht wird. Löschungen werden nur bei normaler Beendigung der Anwendung durchgeführt.

Besonderheiten des DMS-Dateisystems

Verzeichnisse können nicht gelöscht werden.

4.2.2.10 Methoden zum Auflisten von Verzeichnissen

Die Methoden *list()* liefern eine Liste aller oder ausgewählter Namen von Dateien und Verzeichnissen in dem Verzeichnis, das dieses *RecordFile*-Objekt repräsentiert. Die Datei- bzw. Verzeichnisnamen werden ohne Vaterverzeichnisse geliefert. Die Ordnung der gelieferten Namen ist undefiniert. Eine Auswahl der gelieferten Dateien und Verzeichnisse kann durch einen Filter vorgenommen werden (siehe *RecordFilenameFilter* in Kapitel "API-Übersicht").

Name	Ergebnisse
:JAVA:	nur die eigene Benutzerkennung, z.B. $USER., für nicht privilegierte Benutzer oder alle Benutzer-Kennungen im Pubset *JAVA* für einen privilegierten Benutzer
:JAVA: HALLO. JAVA	leer, da kein Verzeichnis angegeben wurde
$USER.	sichtbare Dateien der Benutzerkennung *USER* in deren Standard-Katalogkennung
$.	sichtbare Dateien der Standard-System-Kennung
:JAVA:$.	sichtbare Dateien der Standard-System-Kennung auf dem Pubset *JAVA*

Tabelle 3: Beispiel für das DMS-Dateisystem

Die Methoden *listFiles()* liefern ein Array von *RecordFile*-Objekten mit Pfad-namen von Dateien bzw. Verzeichnissen in dem Verzeichnis, das vom Pfadnamen dieses *RecordFile*-Objektes repräsentiert wird. Die entstehenden Pfadnamen werden aus dem Verzeichnis selbst und den ermittelten Datei- bzw. Verzeichnisnamen zusammengesetzt. Die Ordnung der Namen ist undefiniert. Eine Auswahl der gelieferten Dateien und Verzeichnisse kann durch den Einsatz von Filtern vorgenommen werden (siehe *RecordFilenameFilter* und *RecordFileFilter* "API-Übersicht").

Die Methoden *list()* und *listFiles()* können einer Anwendung Kenntnis über Dateinamen verschaffen, die nicht zu ihrem eigentlichen Bereich gehören. Diese Methoden unterliegen daher der Überwachung durch einen aktiven *Security Manager*. Die Methoden sind daher nur erlaubt, wenn die Anwendung Leseberechtigung für das entsprechende Verzeichnis hat (siehe Abschnitt „Sicherheit").

Die statische Methode *listRoots()* liefert eine Liste aller Dateisystempräfixe („Wurzeln“) für das angegebene Dateisystem. Es ist garantiert, dass der kanonische Pfadname einer tatsächlich physikalisch existierenden Datei mit einem der von *listRoots()* gelieferten Präfixe beginnt.

Die Methode *listRoots()* verschafft einer Anwendung Kenntnis über die Struktur des Dateisystems. Wenn ein Security Manager aktiv ist, werden daher nur die Wurzeln angezeigt, für die eine Leseberechtigung erteilt wurde. Soll die Anwendung alle Wurzeln ermitteln können, müssen Sie ihr die Leseberechtigung für *<<ALL FILES>>* erteilen (siehe Abschnitt „Sicherheit").

Besonderheiten des DMS-Dateisystems

Es wird eine Liste aller zugreifbaren Katalogkennungen geliefert.

4.2.3 AccessParameter

Die Klasse *AccessParameter* und die davon abgeleiteten Klassen beschreiben alle Parameter, die für den Zugriff auf satzorientierten Dateien notwendig sind (bzw. unterstützt werden) und enthalten wenigstens die verwendete Zugriffsmethode, das zugehörigen Dateisystem sowie die Standard-Parameter Satzformat und Satzlänge.

Die zugriffsmethodenspezifischen Implementierungen dieser Klasse können zusätzliche Parameter definieren und bieten dann Methoden zum Setzen und Abfragen der Werte dieser Parameter an. Die folgenden Unterabschnitte beschreiben die allgemeinen Methoden, die jede Implementierung zur Verfügung stellen muss sowie die spezifischen Methoden zu den derzeit unterstützten Zugriffsmethoden.

Objekte der zugriffsmethodenspezifischen Implementierungen der Klasse *AccessParameter* können benutzt werden, um neue Dateien mit der jeweiligen Zugriffsmethode im entsprechenden Dateisystem anzulegen. Intern werden solche Objekte auch für andere Zugriffe auf Dateien (z.B. Öffnen) verwendet. Sie können nur in dem Dateisystem zur Erzeugung von Dateien verwendet werden, aus dem sie stammen.

Objekte dieser abstrakten Klasse können vom Anwender nicht erzeugt werden. Die Klasse *RecordFile* bietet aber die Methoden *getAccessParameter()* und *getDefaultAccessParameter()*, über die Sie sich Objekte der Zugriffsmethoden-spezifischen Implementierung dieser Klasse liefern lassen können.

Unzulässige Werte bei den einzelnen Parametern werden zumeist nicht beim Eintragen der Werte in das *Parameter* -Objekt, sondern erst bei dessen Verwendung entdeckt.

4.2.3.1 Allgemeine Parameter-Methoden und Konstanten

Die Methode *getFileSystem()* liefert den Namen des zugehörigen Dateisystems.

Von der Methode *getAccessMethod()* wird der Name der Zugriffsmethode geliefert, zu der dieses *Parameter*-Objekt gehört.

Darüber hinaus muss jede Implementierung die Methoden *getRecordFormat()*, *setRecordFormat()*, *getRecordLength()* und *setRecordLength()* bereitstellen. Auf deren Beschreibung wird hier verzichtet, weil sie in den folgenden Abschnitten in ihren spezifischen Ausprägungen erläutert werden.

Die Konstanten *RECORD_FORMAT_UNKNOWN*, *RECORD_FORMAT_FIXED* und *RECORD_FORMAT_VARIABLE* werden als Argumente beim Aufruf der Methode *setRecordFormat()* verwendet. Ihre Bedeutung wird in den folgenden Abschnitten in ihren spezifischen Zugriffsmethoden erläutert.

Die Konstanten *NO_WAIT, THREAD_WAIT* und *APPLICATION_WAIT* werden im Rahmen der Shared-Update-Verarbeitung als Argumente beim Aufruf der Methode *setWaitMode()* verwendet. Ihre Bedeutung wird ebenfalls in den folgenden Abschnitten in ihren spezifischen Zugriffsmethoden erläutert.

4.2.3.2 Parameter für SAM im DMS

Die Klasse *AccessParameterSAM* im Package *com.fujitsu.ts.jrio.DMS* stellt eine Reihe zusätzlicher Methoden zum Setzen und Abfragen weiterer Parameter, die für diese Zugriffsmethode spezifisch sind, bereit.

Objekte dieser abstrakten Klasse können vom Anwender nicht erzeugt werden. Die Klasse *RecordFile* bietet aber die Methoden *getAccessParameter()* und *getDefaultAccessParameter()*, über die der Anwender Objekte der Implementierung dieser Klasse erhalten kann.

Unzulässige Werte bei den einzelnen Parametern werden zumeist nicht beim Eintragen der Werte in das Parameter-Objekt, sondern erst bei dessen Verwendung entdeckt.

Die Methode *getRecordFormat()* liefert das Satzformat, das in diesem Parameter-Objekt gespeichert ist. Die Methode *setRecordFormat()* setzt das Satzformat in diesem *Parameter*-Objekt. Die Angaben *RECORD_FORMAT_FIXED* und *RECORD_FORMAT_VARIABLE* sind bei SAM möglich. Dieser Parameter entspricht der Angabe *RECFORM* im DMS.

Die Methode *getRecordLength()* liefert die Satzlänge, die in diesem Parameter-Objekt gespeichert ist. Die Methode *setRecordLength()* setzt die Satzlänge in diesem *Parameter*-Objekt. Dieser Parameter entspricht der Angabe *RECSIZE* im DMS. Im Zusammenhang mit festem Satzformat, definiert dieser Parameter die genaue Länge jedes Satzes einer Datei. Bei variablem Satzformat definiert er die maximale Satzlänge jedes Satzes. Dann ist die Längenangabe 0 erlaubt und bedeutet unbegrenzte Satzlänge. Die DMS/SAM spezifischen Einschränkungen und Abhängigkeiten zu anderen Parametern (Satzformat, Blocklänge) gelten für JRIO natürlich genauso, wie an anderen DMS-Schnittstellen.

Die Methode *getBlockSize()* liefert die Blocklänge, die in diesem Parameter-Objekt gespeichert ist. Die Methode *setBlockSize()* setzt die logische Blocklänge (als Anzahl PAM-Blöcke) in diesem Parameter-Objekt. Dieser Parameter entspricht der Angabe *BLKSIZE=(STD,n)* im DMS. Die Abhängigkeiten zur Satzlänge gelten für JRIO natürlich genauso, wie an anderen DMS-Schnittstellen.

Die Methode *getBlockControl()* liefert das Blockformat, das in diesem Parameter-Objekt gespeichert ist. Dieser Parameter entspricht der Angabe BLKCTRL im DMS. Die Methode *setBlockControl()* setzt das Blockformat in diesem Parameter-Objekt. Die Angaben *BLOCK_CONTROL_BY_PUBSET, BLOCK_CONTROL_DATA, BLOCK_CONTROL_NO, BLOCK_CONTROL_PAMKEY, BLOCK_CONTROL_DATA_2K und BLOCK_CONTROL_DATA_4K* sind möglich. Dieser Parameter wird nur beim Erzeugen neuer Dateien berücksichtigt.

Die Methode *getPrimarySpaceAllocation()* liefert den in diesem Parameter-Objekt gespeicherten Wert für die primäre Speicherplatzzuordnung einer Datei. Die Methode *setPrimarySpaceAllocation()* setzt den Wert für die primäre Speicherplatzzuordnung einer Datei in diesem Parameter-Objekt. Dieser Parameter entspricht dem ersten Teil der Angabe *SPACE* im DMS.

Die Methode *getSecondarySpaceAllocation()* liefert den in diesem Parameter-Objekt gespeicherten Wert für die sekundäre Speicherplatzzuordnung einer Datei. Die Methode *setSecondarySpaceAllocation()* setzt den Wert für die sekundäre Speicherplatzzuordnung einer Datei in diesem *Parameter*-Objekt. Dieser Parameter entspricht dem zweiten Teil der Angabe *SPACE* im DMS.

Die Zugriffsmethode SAM erlaubt das simultane Öffnen einer Datei durch mehrere Anwendungen nur bei lesendem Zugriff. Daher ist die Shared-Update-Verarbeitung für SAM-Dateien nicht möglich.

4.2.3.3 Parameter-Methoden für ISAM im DMS

Die Klasse *AccessParameterISAM* im Package *com.fujitsu.ts.jrio.DMS* stellt eine Reihe zusätzlicher Methoden zum Setzen und Abfragen weiterer Parameter, die für diese Zugriffsmethode spezifisch sind, bereit.

Objekte dieser abstrakten Klasse können vom Anwender nicht erzeugt werden. Die Klasse *RecordFile* bietet aber die Methoden *getAccessParameter()* und *getDefaultAccessParameter()*, über die der Anwender Objekte der Implementierung dieser Klasse erhalten kann.

Unzulässige Werte bei den einzelnen Parametern werden zumeist nicht beim Eintragen der Werte in das Parameter-Objekt, sondern erst bei dessen Verwendung entdeckt.

Die Methode *getRecordFormat()* liefert das Satzformat, das in diesem Parameter-Objekt gespeichert ist. Die Methode *setRecordFormat()* setzt das Satzformat in diesem Parameter-Objekt. Die Angaben *RECORD_FORMAT_FIXED* und *RECORD_FORMAT_VARIABLE* sind bei ISAM möglich. Dieser Parameter entspricht der Angabe *RECFORM* im DMS.

Die Methode *getRecordLength()* liefert die Satzlänge, die in diesem Parameter-Objekt gespeichert ist. Die Methode *setRecordLength()* setzt die Satzlänge in diesem Parameter-Objekt. Dieser Parameter entspricht der Angabe *RECSIZE* im DMS. Im Zusammenhang mit festem Satzformat, definiert dieser Parameter die genaue Länge jedes Satzes einer Datei. Bei variablem Satzformat definiert er die maximale Satzlänge jedes Satzes. Dann ist die Längenangabe 0 erlaubt und bedeutet unbegrenzte Satzlänge. Die DMS/ISAM spezifischen Einschränkungen und Abhängigkeiten zu anderen Parametern (Satzformat, Blocklänge) gelten für JRIO natürlich genauso, wie an anderen DMS-Schnittstellen.

Die Methode *getBlockSize()* liefert die Blocklänge, die in diesem Parameter-Objekt gespeichert ist. Die Methode *setBlockSize()* setzt die logische Blocklänge (als Anzahl PAM-Blöcke) in diesem Parameter-Objekt. Dieser Parameter entspricht der Angabe BLKSIZE=(STD,n) im DMS. Die Abhängigkeiten zur Satzlänge gelten für JRIO natürlich genauso, wie an anderen DMS-Schnittstellen.

Die Methode *getBlockControl()* liefert das Blockformat, das in diesem Parameter-Objekt gespeichert ist. Dieser Parameter entspricht der Angabe BLKCTRL im DMS. Die Methode *setBlockControl()* setzt das Blockformat in diesem Parameter-Objekt. Die Angaben *BLOCK_CONTROL_BY_PUBSET*, *BLOCK_CONTROL_DATA*, *BLOCK_CONTROL_NO*, *BLOCK_CONTROL_PAMKEY*, *BLOCK_CONTROL_DATA_2K* und *BLOCK_CONTROL_DATA_4K* sind möglich. Dieser Parameter wird nur beim Erzeugen neuer Dateien berücksichtigt.

Die Methode *getSharedUpdate()* liefert *true* bzw. *false*, je nachdem ob mit diesem ParameterObjekt die gleichzeitige Bearbeitung einer Datei durch mehrere Anwendungen (Shared-Update-Verarbeitung) erlaubt ist (erlaubt werden soll) bzw. verboten ist (verboten werden soll). Die Methode *setSharedUpdate()* legt fest, ob mit diesem Parameter-Objekt für eine Datei die Shared-Update-Verarbeitung erlaubt sein soll (*setSharedUpdate(true)*) oder nicht (*setSharedUpdate(false)*). Der Parameter wird nur beim Öffnen einer Datei berücksichtigt. Dieser Parameter entspricht der Angabe *SHARUPD* im DMS.

Die Methode *getWaitMode()* liefert die in diesem Parameter-Objekt gespeicherte Einstellung für das Verhalten der Anwendung bei Zugriffskonflikten während der Shared-Update-Verarbeitung für eine mit diesem Parameter-Objekt geöffnete Datei.

Die Methode *setWaitMode()* steuert das Verhalten der Anwendung bei Zugriffskonflikten während der Shared-Update-Verarbeitung für eine Datei. Die Angaben *NO_WAIT*, *THREAD_WAIT* und *APPLICATION_WAIT* sind möglich. *NO_WAIT* bewirkt, dass die Anwendung nicht auf Zuteilung der Sperre wartet und bei einem Konflikt eine *RecordLockedException* ausgelöst wird.

THREAD_WAIT bewirkt, dass der Thread in einen Wartezustand versetzt wird. Nach Ablauf einer (intern festgelegten kurzen) Wartezeit wird solange erneut versucht, die Sperre zu erhalten, bis dies gelingt oder die Anwendung beendet wird. *APPLICATION_WAIT* bewirkt, dass die gesamte Anwendung an der Systemschnittstelle auf die Zuteilung der Sperre wartet. Die Wartezeit an der Systemschnittstelle ist vom Betriebssystem auf ca. 12 h begrenzt. Nach Ablauf dieser Zeit wird nach einer (intern festgelegten kurzen) Wartezeit der Systemaufruf solange erneut aufgesetzt, bis die Sperre erhalten oder die Anwendung beendet wird. Dieser Parameter hat im DMS keine direkte Entsprechung, da das Warteverhalten bei ISAM-Shared-Update nur über den *EXLST*-Mechanismus gesteuert werden kann.

Die Methode *getPrimarySpaceAllocation()* liefert den in diesem Parameter-Objekt gespeicherten Wert für die primäre Speicherplatzzuordnung einer Datei. Die Methode *setPrimarySpaceAllocation()* setzt den Wert für die primäre Speicherplatzzuordnung einer Datei in diesem Parameter-Objekt. Dieser Parameter entspricht dem ersten Teil der Angabe *SPACE* im DMS.

Die Methode *getSecondarySpaceAllocation()* liefert den in diesem Parameter-Objekt gespeicherten Wert für die sekundäre Speicherplatzzuordnung einer Datei. Die Methode *setSecondarySpaceAllocation()* setzt den Wert für die sekundäre Speicherplatzzuordnung einer Datei in diesem Parameter-Objekt. Dieser Parameter entspricht dem zweiten Teil der Angabe *SPACE* im DMS.

Die Methode *getPrimaryKeyPosition()* liefert den in diesem Parameter-Objekt gespeicherten Wert für die Schlüsselposition einer ISAM-Datei. Die Methode *setPrimaryKeyPosition()* setzt den Wert für die Schlüsselposition einer ISAM-Datei in diesem Parameter-Objekt. Dieser Parameter entspricht der Angabe *KEYPOS* im DMS mit dem Unterschied, dass die Nummerierung der Positionen in JRIO von der anderer DMS-Schnittstellen abweicht (siehe Abschnitt „Record").

Die Methode *getPrimaryKeyLength()* liefert den in diesem Parameter-Objekt gespeicherten Wert für die Schlüssellänge einer ISAM-Datei. Die Methode *setPrimaryKeyLength()* setzt den Wert für die Schlüssellänge einer ISAM-Datei in diesem Parameter-Objekt. Dieser Parameter entspricht der Angabe *KEYLEN* im DMS.

Die Methode *getDuplicateKeyIndicator()* liefert den in diesem Parameter-Objekt gespeicherten Wert für die Erlaubnis des mehrfachen Auftretens gleicher Schlüsselwerte in einer ISAM-Datei. Die Methode *setDuplicateKeyIndicator()* setzt den Wert für die Erlaubnis des mehrfachen Auftretens gleicher Schlüsselwerte in einer ISAM-Datei in diesem ParameterObjekt. Dieser Parameter entspricht der Angabe *DUPEKY* im DMS.

4.2.3.4 Parameter-Methoden für UPAM im DMS

Die Klasse *AccessParameterUPAM* im Package *com.fujitsu.ts.jrio.DMS* stellt eine Reihe zusätzlicher Methoden zum Setzen und Abfragen weiterer Parameter, die für diese Zugriffsmethode spezifisch sind, bereit.

Objekte dieser abstrakten Klasse können vom Anwender nicht erzeugt werden. Die Klasse *RecordFile* bietet aber die Methoden *getAccessParameter()* und *getDefaultAccessParameter()*, über die der Anwender Objekte der Implementierung dieser Klasse erhalten kann.

Unzulässige Werte bei den einzelnen Parametern werden zumeist nicht beim Eintragen der Werte in das Parameter-Objekt, sondern erst bei dessen Verwendung entdeckt.

Die Methode *getRecordFormat()* liefert das Satzformat, das in diesem Parameter-Objekt gespeichert ist. Die Methode *setRecordFormat()* setzt das Satzformat in diesem Parameter-Objekt. Nur die Angabe *RECORD_FORMAT_FIXED* ist bei UPAM möglich.

Die Methode *getRecordLength()* liefert die Satzlänge, die in diesem Parameter-Objekt gespeichert ist. Die Methode *setRecordLength()* setzt die Satzlänge in diesem Parameter-Objekt. Für UPAM ist die Satzlänge immer identisch zur logischen Blocklänge in Bytes. Es sind also nur Werte zulässig, die ein Vielfaches von 2048 sind.

Die Methode *getBlockControl()* liefert das Blockformat, das in diesem Parameter-Objekt gespeichert ist. Dieser Parameter entspricht der Angabe BLKCTRL im DMS. Die Methode *setBlockControl()* setzt das Blockformat in diesem Parameter-Objekt. Die Angaben *BLOCK_CONTROL_BY_PUBSET, BLOCK_CONTROL_DATA, BLOCK_CONTROL_NO, BLOCK_CONTROL_PAMKEY, BLOCK_CONTROL_DATA_2K und BLOCK_CONTROL_DATA_4K* sind möglich. Dieser Parameter wird nur beim Erzeugen neuer Dateien berücksichtigt.

Die Methode *getPrimarySpaceAllocation()* liefert den in diesem Parameter-Objekt gespeicherten Wert für die primäre Speicherplatzzuordnung einer Datei. Die Methode *setPrimarySpaceAllocation()* setzt den Wert für die primäre Speicherplatzzuordnung einer Datei in diesem Parameter-Objekt. Dieser Parameter entspricht dem ersten Teil der Angabe *SPACE* im DMS.

Die Methode *getSecondarySpaceAllocation()* liefert den in diesem Parameter-Objekt gespeicherten Wert für die sekundäre Speicherplatzzuordnung einer Datei. Die Methode *setSecondarySpaceAllocation()* setzt den Wert für die sekundäre Speicherplatzzuordnung einer Datei in diesem Parameter-Objekt. Dieser Parameter entspricht dem zweiten Teil der Angabe *SPACE* im DMS.

Die Methode *getSharedUpdate()* liefert *true* bzw. *false*, je nachdem ob mit diesem ParameterObjekt die gleichzeitige Bearbeitung einer Datei durch mehrere Anwendungen (Shared-Update-Verarbeitung) erlaubt ist (erlaubt werden soll) bzw. verboten ist (verboten werden soll). Die Methode *setSharedUpdate()* legt fest, ob mit diesem Parameter-Objekt für eine Datei die Shared-Update-Verarbeitung erlaubt sein soll (*setSharedUpdate(true)*) oder nicht (*setSharedUpdate(false)*). Der Parameter wird nur beim Öffnen einer Datei berücksichtigt. Dieser Parameter entspricht der Angabe *SHARUPD* im DMS.

Die Methode *getWaitMode()* liefert die in diesem Parameter-Objekt gespeicherte Einstellung für das Verhalten der Anwendung bei Zugriffskonflikten während der Shared-Update-Verarbeitung für eine mit diesem Parameter-Objekt geöffnete Datei. Die Methode *setWaitMode()* steuert das Verhalten der Anwendung bei Zugriffskonflikten während der Shared-Update-Verarbeitung für eine Datei. Die Angaben *NO_WAIT*, *THREAD_WAIT* und *APPLICATION_WAIT* sind möglich. *NO_WAIT* bewirkt, dass die Anwendung nicht auf Zuteilung der Sperre wartet und im Konfliktfall eine *RecordLockedException* ausgelöst wird.

THREAD_WAIT bewirkt, dass der Thread in einen Wartezustand versetzt wird. Nach Ablauf einer (intern festgelegten kurzen) Wartezeit wird solange erneut versucht, die Sperre zu erhalten, bis dies gelingt oder die Anwendung beendet wird. *APPLICATION_WAIT* bewirkt, dass die gesamte Anwendung an der Systemschnittstelle

auf Zuteilung der Sperre wartet. Die Wartezeit an der Systemschnittstelle ist vom Betriebssystem auf ca. 30 min begrenzt. Nach Ablauf dieser Zeit wird nach einer (intern festgelegten kurzen) Wartezeit der Systemaufruf solange erneut aufgesetzt, bis die Sperre erhalten oder die Anwendung beendet wird. Dieser Parameter hat im DMS keine direkte Entsprechung, da das Warteverhalten bei *UPAM-Shared-Update* nur über den *PAMTOUT*-Wert gesteuert werden kann.

4.2.4 Sequentielle Datenbearbeitung

Für die sequentielle Verarbeitung von Dateien oder anderer Medien, die Datensätze enthalten, stehen getrennte Schnittstellengruppen für die Eingabe und die Ausgabe zur Verfügung. Die Struktur, Benennung und Funktionalität dieser Schnittstellen orientiert sich dabei an den aus dem normalen Package *java.io* bekannten Klassen zur sequentiellen Ein-/Ausgabe.

4.2.4.1 InputRecordStream

Die abstrakte Klasse *InputRecordStream* ist die Basisklasse für alle Implementierungen von Klassen, die das sequentielle Lesen von Datensätzen erlauben. Mit dem JRIO-API werden zwei Implementierungen dieser abstrakten Klasse zur Verfügung gestellt, die Klasse *FileInputRecordStream* für das sequentielle Lesen aus einer Datei und die Klasse *ArrayInputRecordStream* für das sequentielle Lesen aus einem Array von *Record*-Objekten.

Die abstrakte Klasse schreibt die Implementierung von Methoden zum sequentiellen Lesen und Überspringen von Sätzen ebenso wie das Schließen der Datei vor sowie eine Methodengruppe zur elementaren Rückpositionierung (mark/reset), die aber von Implementierungen nicht notwendig unterstützt werden muss.

Die Methoden der abstrakten Klasse werden hier nicht weiter beschrieben, sondern bei den einzelnen Implementierungen in ihrer konkreten Ausprägung erläutert. Die API-Dokumentation enthält diese Beschreibung für Anwender, die eigene Implementierungen realisieren wollen.

4.2.4.2 FileInputRecordStream

Ein *FileInputRecordStream*-Objekt repräsentiert eine für den sequentiellen Lese-Zugriff geöffnete Datei. Die Datei wird beim Erzeugen des Objektes (siehe Abschnitt „Öffnen und Schließen einer Datei") implizit geöffnet.

Die *FileInputRecordStream*-Klasse bietet Methoden zum Lesen und Überspringen von Sätzen sowie zum Schließen der Datei. Die Methodengruppe zur Positionierung ist zwar vorhanden, stellt aber keine Funktionalität zur Verfügung.

Die Datei, die geöffnet werden soll, muss im zugrunde liegenden Dateisystem schon existieren. Für die Erzeugung einer Datei muss die Methode *createNewFile()* der Klasse *RecordFile* benutzt werden.

Für eine Datei, die für sequentiellen Lese-Zugriff geöffnet wurde, ist immer eine aktuelle Dateiposition definiert, an der die nächste Lese-Operation stattfindet. Die aktuelle Dateiposition wird durch die Nummer des Satzes, gemäß der Anordnung der Sätze in dieser Datei, definiert, wobei die Sätze einer Datei mit Null beginnend nummeriert werden. Nach dem Öffnen der Datei ist die aktuelle Dateiposition der Dateianfang.

Öffnen und Schließen einer Datei

Bei der Konstruktion eines *FileInputRecordStream*-Objektes wird die als *RecordFile*-Objekt angegebene Datei mit der angegebenen Zugriffsmethode oder den angegebenen Zugriffsparametern im Lese-Modus geöffnet. Die Datei muss im darunter liegenden Dateisystem bereits existieren und die Zugriffsmethode muss zu diesem Dateisystem gehören und muss für diese Datei erlaubt sein. Der Benutzer muss die zum Lesen erforderlichen Zugriffsrechte auf die Datei besitzen. Falls ein Security Manager aktiv ist und dessen Einschränkungen für diese Datei ein Lesen nicht erlauben, so wird eine *Exception* ausgelöst (siehe Abschnitt „Sicherheit").

Werden zum Öffnen Zugriffsparameter angegeben, so werden diese beim Öffnen berücksichtigt, soweit nicht die Datei-Parameter Vorrang haben. Nach dem Öffnen werden sie mit den entsprechenden Werten der geöffneten Datei aktualisiert.

Die Methode *close()* schließt die Datei. Danach sind keine E/A-Operationen über dieses *FileInputRecordStream*-Objekt mehr möglich.

Besonderheiten des DMS-Dateisystems

> Die Shared-Update-Verarbeitung (siehe Abschnitt „Shared-Update-Verarbeitung" sowie Abschnitt „AccessParameter") eines *FileInputRecordStream* ist für die Zugriffsmethoden ISAM und UPAM möglich. Mit UPAM können allerdings nur PAM-Dateien mit der Shared-Update-Verarbeitung geöffnet werden. Da die Datei nur zum Lesen geöffnet wird, erfolgen alle Zugriffe ohne Sperre. Es können also keine Zugriffskonflikte auftreten. Es muss jedoch damit gerechnet werden, dass der Inhalt des Satzes zwischenzeitlich von einer anderen Anwendung geändert wurde.

Methoden zum Lesen von Sätzen

Die *read*-Methode wird in zwei Ausprägungen angeboten, in einer Variante, bei der der gelesene Satz in einem neu erzeugten *Record*-Objekt als Ergebnis bereitgestellt wird und einer Variante, bei der ein vom Aufrufer als Argument übergebenes *Record*-Objekt mit den Daten des gelesenen Satzes gefüllt wird.

Wird ein Satzpuffer neu angelegt, so hat er genau die Größe der gelesenen Daten. Stellt der Aufrufer das *Record*-Objekt bereit, muss er dafür sorgen, dass der Satzpuffer groß genug ist, um die Daten des zu lesenden Satzes aufzunehmen. Wenn der angegebene Satzpuffer zu klein ist, um die kompletten Daten aufzunehmen, wird eine Ausnahme ausgelöst und es werden keine Daten übertragen.

Die Methoden *read()* lesen den Satz an der aktuellen Dateiposition. Die aktuelle Dateiposition wird anschließend um eins erhöht, d.h. es wird automatisch auf den nächsten Satz positioniert.

Mit der Methode *skip()* ist es möglich, die angegebene Anzahl Sätze in der Datei zu überspringen. Dabei kann es vorkommen, dass nicht exakt die angegebene Anzahl an Sätzen übersprungen werden kann (z.B. wenn nicht mehr genügend Sätze in der Datei vorhanden sind). Der Rückgabewert von *skip()* gibt die tatsächliche Anzahl der übersprungenen Sätze an.

Die Methode *available()* liefert die Anzahl der Sätze, die ohne Blockieren mindestens gelesen werden können. Aber auch das Ergebnis Null, das häufig geliefert wird, wenn es nicht oder nur aufwendig feststellbar ist, ob ein Leseversuch zu einem Wartezustand (des Threads) führt, erlaubt nicht den Umkehrschluss, dass der nächste Aufruf von *read()* oder *skip()* tatsächlich zu solch einem Wartezustand führt.

Methoden zur Positionierung

Die Methode *markSupported()* gibt Auskunft darüber, ob das Markieren bzw. Repositionieren für diese Datei unterstützt wird. Analog zur Klasse *java.io.FileInputStream* wird für Objekte dieser Klasse derzeit grundsätzlich diese Positionierung nicht unterstützt, d.h. diese Methode liefert immer false.

Die Methode *mark()* ist zwar vorhanden, hat aber keine Funktion.

Ein Aufruf von *reset()* führt zu einer Ausnahme, da diese Funktionalität derzeit nicht unterstützt wird.

4.2.4.3 ArrayInputRecordStream

Ein *ArrayInputRecordStream*-Objekt repräsentiert ein für den sequentiellen Lese-Zugriff geöffnetes Array von *Record*-Objekten. Das Öffnen erfolgt implizit beim Erzeugen des Objektes (siehe „Öffnen und Schließen"), hat aber hier keine weitere Bedeutung, wie etwa bei Dateien.

Die *ArrayInputRecordStream*-Klasse bietet Methoden zum Lesen und Überspringen von Sätzen. Die Methodengruppe zur Positionierung ist ebenfalls vollständig unterstützt.

Innerhalb des Array aus dem gelesen wird, ist immer eine aktuelle Leseposition definiert, an der die nächste Lese-Operation stattfindet. Die aktuelle Leseposition wird durch die Nummer des Satzes in dem Array definiert, wobei die Sätze mit Null beginnend nummeriert werden. Nach dem Öffnen ist die aktuelle Leseposition Null.

Öffnen und Schließen

Bei der Konstruktion eines *ArrayInputRecordStream*-Objektes stellt der Aufrufer das Array mit Datensätzen bereit, aus dem später gelesen werden soll. Dabei wird dieses Array direkt verwendet und nicht etwa kopiert, d.h. etwaige Manipulationen an diesem Array oder den darin enthaltenen Datensätzen wirken sich direkt auf das *ArrayInputRecordStream*-Objekt aus. Mit einer zweiten Variante des Konstruktors kann der Anwender einen Teil eines Arrays mit Datensätzen (definiert durch Offset und Länge) für die Eingabe bereitstellen.

Die Methode *close()* ist zwar vorhanden, hat aber für diese Klasse keine Funktion.

Methoden zum Lesen von Sätzen

Die *read*-Methode wird in zwei Ausprägungen angeboten. Einmal eine Variante, bei der der gelesene Satz in einem neu erzeugten *Record*-Objekt als Ergebnis bereitgestellt wird und einer Variante, bei der ein vom Aufrufer als Argument übergebenes *Record*-Objekt mit den Daten des gelesenen Satzes gefüllt wird.

Wird ein Satzpuffer neu angelegt, so hat er genau die Größe der gelesenen Daten. Stellt der Aufrufer das *Record*-Objekt bereit, muss er dafür sorgen, dass der Satzpuffer groß genug ist, um die Daten des zu lesenden Satzes aufzunehmen. Wenn der angegebene Satzpuffer zu klein ist, um die kompletten Daten aufzunehmen, wird eine Ausnahme ausgelöst und es werden keine Daten übertragen.

Die Methoden *read()* lesen den Satz an der aktuellen Leseposition. Die aktuelle Leseposition wird anschließend um eins erhöht, d.h. es wird automatisch auf den nächsten Satz im Array positioniert.

Mit der Methode *skip()* ist es möglich, die angegebene Anzahl Sätze im Array zu überspringen. Dabei kann es vorkommen, dass nicht exakt die angegebene Anzahl an Sätzen übersprungen werden kann, weil das Array nicht mehr so viele Datensätze enthält. Der Rückgabewert von *skip()* gibt die tatsächliche Anzahl der übersprungenen Sätze an.

Die Methode *available()* liefert die Anzahl der Sätze, die noch gelesen werden können, bevor das Ende des Arrays erreicht ist. Das Lesen aus einem Array von Datensätzen führt niemals zu Wartezuständen.

Methoden zur Positionierung

Die Methode *markSupported()* gibt Auskunft darüber, ob das Markieren bzw. Repositionieren für diesen Datenstrom unterstützt wird. In dieser Klasse liefert diese Methode immer true.

Die Methode *mark()* merkt sich die aktuelle Leseposition, um später an diese Stelle zurückpositionieren zu können. Das für *mark()* vorgesehene Argument wird bei dieser Implementierung ignoriert und sollte immer als 0 angegeben werden.

Ein Aufruf von *reset()* repositioniert auf eine zuvor mit *mark()* gemerkte Lese-position.

4.2.4.4 OutputRecordStream

Die abstrakte Klasse *OutputRecordStream* ist die Basisklasse für alle Implementierungen von Klassen, die das sequentielle Schreiben von Datensätzen erlauben. Mit dem JRIO-API werden zwei Implementierungen dieser abstrakten Klasse zur Verfügung gestellt, die Klasse *FileOutputRecordStream* für das sequentielle Schreiben in eine Datei und die Klasse *ArrayOutputRecordStream* für das sequentielle Schreiben in ein Array von *Record*-Objekten.

Diese abstrakte Klasse schreibt die Implementierung von Methoden zum sequentiellen Schreiben von Sätzen ebenso wie das Schließen der Datei vor.

Die Methoden dieser abstrakten Klasse werden hier nicht weiter beschrieben, sondern bei den einzelnen Implementierungen in ihrer konkreten Ausprägung erläutert. Die API-Dokumentation enthält diese Beschreibung für Anwender, die eigene Implementierungen realisieren wollen.

4.2.4.5 FileOutputRecordStream

Ein *FileOutputRecordStream*-Objekt repräsentiert eine für den sequentiellen Schreib-Zugriff geöffnete Datei. Die Datei wird beim Erzeugen des Objektes implizit geöffnet (siehe Abschnitt „Öffnen und Schließen einer Datei“ unten).

Die *FileOutputRecordStream*-Klasse bietet Methoden zum Schreiben von Sätzen sowie zum Schließen der Datei. In einer Datei, die für sequentiellen Schreib-Zugriff geöffnet wurde, werden Datensätze immer am Ende der Datei hinzugefügt.

Die Datei, die geöffnet werden soll, muss im zugrunde liegenden Dateisystem schon existieren. Für die Erzeugung einer Datei muss die Methode *createNewFile()* der Klasse *RecordFile* benutzt werden.

Öffnen und Schließen einer Datei

Bei der Konstruktion eines *FileOutputRecordStream*-Objektes wird die als *RecordFile*-Objekt angegebene Datei mit der angegebenen Zugriffsmethode oder den angegebenen Zugriffsparametern im Schreib-Modus geöffnet. Dabei kann gewählt werden, ob ein evtl. vorhandener Inhalt der Datei beim Öffnen gelöscht wird oder nicht.

Die Datei muss im darunter liegenden Dateisystem bereits existieren und die Zugriffsmethode muss zu diesem Dateisystem gehören und muss für diese Datei erlaubt sein. Der Benutzer muss die zum Schreiben erforderlichen Zugriffsrechte auf die Datei besitzen. Falls ein Security Manager aktiv ist und dessen Einschränkungen für diese Datei ein Schreiben nicht erlauben, so wird eine *Exception* ausgelöst (siehe Abschnitt „Sicherheit").

Werden zum Öffnen Zugriffsparameter angegeben, so werden diese beim Öffnen berücksichtigt, soweit nicht die Datei-Parameter Vorrang haben. Nach dem Öffnen werden sie mit den entsprechenden Werten der geöffneten Datei aktualisiert.

Die Methode *close()* schließt die Datei. Danach sind keine E/A-Operationen über dieses *FileOutputRecordStream*-Objekt mehr möglich.

Besonderheiten des DMS-Dateisystems

> Die Shared-Update-Verarbeitung (siehe Abschnitt „Shared-Update-Verarbeitung" sowie Abschnitt „AccessParameter") eines *FileOutputRecordStream* ist für die Zugriffsmethoden ISAM und nur dann möglich, wenn eine existierende Datei zum Erweitern geöffnet wird. Andere Anwendungen können die Datei dann allerdings nicht ebenfalls als *FileOutputRecordStream* öffnen. Generell wird von einer Shared-Update-Verarbeitung im Zusammenhang mit *FileOutputRecordStream* abgeraten. In Ausnahmefällen kann das gleichzeitige Eröffnen als *FileInputRecordStream* sinnvoll sein.

Methoden zum Schreiben von Sätzen

Die Methode *write()* schreibt einen Satz hinter den letzten vorhandenen Satz der Datei. Bei einer Shared-Update-Verarbeitung wird implizit eine Sperre angefordert, was bei Zugriffskonflikten je nach der mittels *setWaitMode()* eingestellten Option zu einer *RecordLockedException* bzw. zum Warten des Threads oder der gesamten Anwendung führen kann. Nach Abschluss des Schreibvorgangs wird die Sperre aufgehoben.

Die Methode *flush()* sorgt dafür, dass tatsächlich alle mit *write()* geschriebenen Sätze in die Datei ausgegeben werden, auch wenn die zugrunde liegende Zugriffsmethode eine Pufferung der Ausgaben vorsieht. Bei einer Shared-Update-Verarbeitung wird eine bestehende Sperre für diese Datei aufgehoben.

4.2.4.6 ArrayOutputRecordStream

Ein *ArrayOutputRecordStream*-Objekt repräsentiert ein für den sequentiellen Schreib-Zugriff geöffnetes Array von *Record*-Objekten. Das Array wird beim Öffnen implizit angelegt (siehe Abschnitt „Öffnen und Schließen“ unten) und wächst mit den hineingeschriebenen Daten.

Die *ArrayOutputRecordStream*-Klasse bietet Methoden zum Schreiben von Sätzen. Beim Schreiben werden Datensätze immer am Ende des Arrays hinzugefügt. Außerdem bietet diese Klasse noch Methoden, um den kompletten Inhalt des Datenstroms abzuholen, ihn zu löschen oder die Größe abzufragen.

Öffnen und Schließen

Bei der Konstruktion eines *ArrayOutputRecordStream*-Objektes wird ein Array intern bereitgestellt, in das später Datensätze geschrieben werden sollen. Der Aufrufer kann dabei angeben, wie viele Datensätze das Array zunächst aufnehmen soll. Tut er das nicht, wird eine Standardgröße angenommen. Reicht diese Größe dann aber nicht aus, um die geschriebenen Datensätze aufzunehmen, wird das Array automatisch intern vergrößert.

Die Methode *close()* ist zwar vorhanden, hat aber für diese Klasse keine Funktion.

Methoden zum Schreiben von Sätzen

Die Methode *write()* fügt einen Satz hinter den letzten vorhandenen Satz des Arrays hinzu.

Die Methode *flush()* ist zwar vorhanden, hat aber für diese Klasse keine Funktion.

Methoden für den Zugriff auf den Inhalt eines Datenstroms

Die Methode *size()* liefert die Anzahl der Datensätze, die sich im Array befinden.

Mit der Methode *reset()* kann der komplette Inhalt des Arrays gelöscht werden. Das Array selbst bleibt dabei in seiner gewachsenen Größe erhalten und wird bei weiteren *write()* Aufrufen wieder gefüllt.

Die Methode *toRecordArray()* liefert den kompletten momentanen Inhalt des Datenstroms als Array von *Record*-Objekten. Das gelieferte Array ist, im Gegensatz zu dem intern verwendeten, genauso groß, wie zur Aufnahme der Daten erforderlich. Die einzelnen Datensätze werden dabei aber nicht kopiert, so dass Manipulationen an den Satzinhalten Auswirkungen auf den Inhalt des Datenstroms haben.

Die Methode *writeTo()* schreibt den kompletten momentanen Inhalt des Daten-stroms in einen gegebenen anderen Datenstrom. Dafür ist jeder Datenstrom geeignet, dessen Implementierung von der abstrakten Klasse *OutputRecordStream* abgeleitet wurde.

4.2.5 Wahlfreie Datenbearbeitung (RandomAccessRecordFile)

Ein *RandomAccessRecordFile*-Objekt repräsentiert eine für wahlfreien Zugriff geöffnete Datei. Die Datei wird beim Erzeugen des Objektes (siehe Abschnitt „Öffnen und Schließen einer Datei") implizit geöffnet.

Die *RandomAccessRecordFile*-Klasse bietet Methoden zum Lesen und Schreiben von Sätzen bzw. zum Kürzen und Erweitern dieser Datei. Zusätzlich gibt es Methoden zum Positionieren und zum Schließen der Datei.

Die Datei, die geöffnet werden soll, muss im zugrunde liegenden Dateisystem schon existieren. Für die Erzeugung einer Datei muss die Methode *createNewFile()* der Klasse *RecordFile* benutzt werden.

Für eine Datei, die für wahlfreien Zugriff geöffnet wurde, ist immer eine aktuelle Dateiposition definiert, an der die nächste Lese- oder Schreiboperation stattfindet. Die aktuelle Dateiposition wird durch die Nummer des Satzes, gemäß der Anordnung der Sätze in dieser Datei, definiert, wobei die Sätze mit Null beginnend nummeriert werden. Die aktuelle Dateiposition nach dem Öffnen ist der Dateianfang.

Beim Öffnen einer Datei für den wahlfreien Zugriff kann die spezifische Zugriffsrichtung eingeschränkt werden und es kann das Löschen des Inhalts einer existierenden Datei angefordert werden.

Die folgenden Open-Modi werden mit dieser Klasse erlaubt:

- **INPUT**
  Nach dem Öffnen der Datei sind nur Lese-Operationen erlaubt.
- **OUTIN**
  Nach dem Öffnen sind sowohl Schreib- als auch Lese-Operationen erlaubt. Der vollständige Dateiinhalt wird beim Öffnen der Datei gelöscht.
- **INOUT**
  Nach dem Öffnen sind sowohl Lese- als auch Schreib-Operationen erlaubt. Beim Öffnen der Datei bleibt der Dateiinhalt unverändert.

Nach dem Schließen der Datei sollte das *RandomAccessRecordFile*-Objekt nicht mehr benutzt werden.

4.2.5.1 Öffnen und Schließen einer Datei

Bei der Konstruktion eines *RandomAccessRecordFile*-Objektes wird die als *RecordFile*-Objekt angegebene Datei mit der angegebenen Zugriffsmethode oder den angegebenen Zugriffsparametern im spezifizierten Modus geöffnet.

Die Datei muss im darunter liegenden Dateisystem bereits existieren und die Zugriffsmethode muss zu diesem Dateisystem gehören und muss für diese Datei erlaubt sein. Der Benutzer muss die für den angegebenen Open-Modus erforderlichen Zugriffsrechte auf die Datei besitzen. Falls ein Security Manager aktiv ist und dessen Einschränkungen für diese Datei mit dem angegebenen Open-Modus kollidieren, so wird eine *Exception* ausgelöst (siehe Abschnitt „Sicherheit").

Werden zum Öffnen Zugriffsparameter angegeben, so werden diese beim Öffnen berücksichtigt, soweit nicht die Datei-Parameter Vorrang haben. Nach dem Öffnen werden sie mit den entsprechenden Werten der geöffneten Datei aktualisiert.

Die Methode *close()* schließt die Datei. Danach sind keine E/A-Operationen über dieses *RandomAccessRecordFile*-Objekt mehr möglich.

Besonderheiten des DMS-Dateisystems

Die Shared-Update-Verarbeitung (siehe Abschnitt „Shared-Update-Verarbeitung" sowie Abschnitt „AccessParameter") eines *RandomAccessRecordFile* ist für die Zugriffsmethode UPAM nur für PAM-Dateien in den OpenModi */NPUT* und */NOUT* möglich. Wenn die Datei im Open-Modus */NPUT* eröffnet wurde, erfolgen alle Zugriffe ohne Sperre. Es können also keine Zugriffskonflikte auftreten. Es muss jedoch damit gerechnet werden, dass der Inhalt des Satzes zwischenzeitlich von einer anderen Anwendung geändert wurde. Beim Open-Modus */NOUT* erfolgen Lese- und Schreibzugriffe mit impliziter Sperre. Bei Zugriffskonflikten kann das je nach mittels *setWaitMode()* eingestellten Option zu einer *RecordLockedException* bzw. zum Warten des Threads oder der gesamten Anwendung führen. Sperren werden beim Schreiben des gesperrten Satzes implizit wieder aufgehoben, können aber auch explizit mit *flush()* aufgehoben werden. Einzelheiten entnehmen Sie der jeweiligen Schnittstellenbeschreibung in der mitgelieferten JAVADOC-Dokumentation.

4.2.5.2 Methoden zum Lesen von Sätzen

Die *read*-Methode wird in zwei Ausprägungen angeboten. Einmal eine Variante, bei der der gelesene Satz in einem neu erzeugten *Record*-Objekt als Ergebnis bereitgestellt wird und einer Variante, bei der ein vom Aufrufer als Argument übergebenes *Record*-Objekt mit den Daten des gelesenen Satzes gefüllt wird.

Wird ein Satzpuffer neu angelegt, so hat er genau die Größe der gelesenen Daten. Stellt der Aufrufer das *Record*-Objekt bereit, muss er dafür sorgen, dass der Satzpuffer groß genug ist, um die Daten des zu lesenden Satzes aufzunehmen. Wenn der angegebene Satzpuffer zu klein ist, um die kompletten Daten aufzunehmen, wird eine Ausnahme ausgelöst und es werden keine Daten übertragen.

Die Methoden *read()* lesen den Satz an der aktuellen Dateiposition. Die aktuelle Dateiposition wird anschließend um eins erhöht, d.h. es wird automatisch auf den nächsten Satz positioniert.

4.2.5.3 Methoden zum Schreiben von Sätzen

Die Methode *write()* schreibt einen Satz an der aktuellen Dateiposition in die Datei. Ein bereits existierender Satz wird überschrieben, aber nur, wenn die für die Zugriffsmethode geltenden Einschränkungen (z.B. gleiche Satzlänge) erfüllt werden. Wenn die aktuelle Dateiposition das Ende der Datei (oder dahinter) ist, so wird die Datei erweitert. Nach dem Schreiben ist die aktuelle Dateiposition der Satz hinter dem geschriebenen Satz bzw. das Dateiende.

Bei der Shared-Update-Verarbeitung ist die Änderung existierender Sätze gegenüber konkurrierenden Anwendungen nur dann sicher, wenn zwischen Lesen und Zurückschreiben die beim Lesen implizit gesetzte Sperre für diesen Satz nicht aufgehoben wurde - insbesondere also zwischenzeitlich kein anderer Satz gelesen oder geschrieben wurde. Bei der Shared-Update-Verarbeitung sollten Sie daher eine entsprechende Aktionsfolge einhalten, eine Überprüfung findet aber nicht statt.

Die Methode *flush()* sorgt dafür, dass tatsächlich alle mit *write()* geschriebenen Sätze in die Datei ausgegeben werden, auch wenn die zugrunde liegende Zugriffsmethode eine Pufferung der Ausgaben vorsieht. Bei der Shared-Update-Verarbeitung wird darüber hinaus garantiert, dass eine von dieser Anwendung gehaltene Sperre für diese Datei aufgehoben wird.

Besonderheiten des DMS-Dateisystems

Bei der Shared-Update-Verarbeitung im DMS-Dateisystem kann die Information über das jeweils aktuelle Dateiende nicht zwischen den beteiligten Anwendungen synchronisiert werden. Ein gleichzeitiges Erweitern von *RandomAccessRecordFiles* durch mehrere Anwendungen ist daher nicht zu empfehlen.

4.2.5.4 Methoden zur Positionierung und Größenänderung

Die Methode *getCurrentRecordNumber()* liefert die aktuelle Dateiposition als Satznummer.

Die Methode *setCurrentRecordNumber()* setzt die aktuelle Position der Datei auf den Satz mit der angegebenen Nummer. Die speziellen Konstanten *POS_FIRST* und *POS_LAST* können benutzt werden, um auf den Dateianfang bzw. das Dateiende zu positionieren.

Die Methode *getRecordCount()* liefert die Anzahl der Sätze dieser Datei. Das ist gleichzeitig die Position des Dateiendes.

Die Methode *setRecordCount()* verändert die Größe der Datei auf die Anzahl der angegebenen Sätze. Wenn die angegebene Anzahl der Sätze kleiner ist, als die aktuelle Anzahl der Sätze dieser Datei, so wird die Datei gekürzt, so dass sie nur noch so viele Sätze enthält, wie angegeben. Wenn dabei die aktuelle Dateiposition größer war, als die neue Dateigröße, so wird die aktuelle Dateiposition auf das neue Ende der Datei gesetzt. Wenn die angegebene Anzahl der Sätze größer ist als die aktuelle Anzahl von Sätzen dieser Datei, so kann die Datei vergrößert werden. Eine Zugriffsmethode kann solch eine Vergrößerung der Datei, z.B. für Dateien mit variablem Satzformat, verweigern. Wird die Operation durchgeführt, so ist der Inhalt der neu eingefügten Sätze undefiniert.

Besonderheiten des DMS-Dateisystems

Bei der Shared-Update-Verarbeitung im DMS-Dateisystem ist ein Verkürzen der Datei nicht möglich. Dies führt zu einer *IOException*.

Bei der Shared-Update-Verarbeitung im DMS-Dateisystem kann die Information über das jeweils aktuelle Dateiende nicht zwischen den beteiligten Anwendungen synchronisiert werden. Ein gleichzeitiges Erweitern von *RandomAccessRecordFiles* durch mehrere Anwendungen ist daher nicht zu empfehlen. Unabhängig davon werden die Sperren so gesetzt, als ob die Datei sukzessive durch Schreiben einzelner Sätze erweitert würde.

4.2.6 Index-sequentielle Datenbearbeitung

Für die index-sequentielle Dateibearbeitung spielen Schlüssel eine ganz zentrale Rolle.

Schlüssel definieren die Ordnung der Sätze innerhalb einer index-sequentiellen Datei. Ein Schlüssel ist immer Bestandteil eines Satzes und ist durch das Schlüsselfeld (Position und Länge) innerhalb jedes Satzes einer index-sequentiellen Datei definiert. Der Inhalt eines Schlüsselfeldes ist der Schlüsselwert. Außerdem kann ein Schlüssel einen Namen haben, wenn das z.B. in einer Implementierung zur Unterscheidung verschiedener Schlüssel notwendig ist.

Es wird unterschieden zwischen primären und sekundären Schlüsseln. Jede index-sequentielle Datei besitzt immer genau einen primären Schlüssel und kann einen oder mehrere sekundäre Schlüssel haben. Sekundäre Schlüssel müssen dabei immer einen eindeutigen Namen besitzen. Hat eine Datei mehrere Schlüssel, so definiert jeder dieser Schlüssel möglicherweise eine andere Ordnung.

Für einen Schlüssel können gleiche Schlüsselwerte in verschiedenen Sätzen erlaubt sein.

Besonderheiten der BS2000 Zugriffsmethode ISAM

Bei ISAM-Dateien ist immer ein unbenannter primärer Schlüssel definiert. Nur für NK-ISAM-Dateien können mehrere sekundäre Schlüssel (max. 30) zusätzlich definiert sein. Die sekundären Schlüssel müssen immer einen eindeutigen Namen (max. 8-stellig) haben. Die bei ISAM gültigen Einschränkungen (z.B. bezüglich Schlüssellänge) müssen natürlich auch bei Nutzung der JRIO-Schnittstellen beachtet werden. Gleiche sekundäre Schlüssel in verschiedenen Sätzen können nur erlaubt werden, wenn für den primären Schlüssel keine gleichen Schlüsselwerte in verschiedenen Sätzen erlaubt wurden, und wenn für alle anderen sekundären Schlüssel bereits gleiche Schlüsselwerte in verschiedenen Sätzen erlaubt wurden.

Die Möglichkeiten der Markierung (Wertmarkierung und logischer Markierung), die ISAM bietet, werden von JRIO nicht unterstützt.

Beachten Sie, dass an den ISAM DMS-Makroschnittstellen Positionen innerhalb eines Satzes, also insbesondere auch Schlüsselpositionen, anders nummeriert sein können, als in den JRIO-Schnittstellen (siehe Abschnitt „Record").

4.2.6.1 KeyDescriptor

Die Klasse *KeyDescriptor* beschreibt die Position, Länge sowie weitere Eigenschaften eines bestimmten Schlüsselfeldes innerhalb eines Satzes einer index-sequentiellen Datei (Schlüsselbeschreibung). Sie stellt Methoden für den Zugriff auf diese Schlüsseleigenschaften einer index-sequentiellen Datei bereit.

Ein *KeyDescriptor*-Objekt dient zum Erzeugen oder Extrahieren eines konkreten Schlüsselwertes. Für ISAM wurden entsprechende Implementierungen dieser abstrakten Klasse bereitgestellt. Sie können also derartige *KeyDescriptor*-Objekte selbst erzeugen oder aber über die Methoden der Klasse *KeyedAccessRecordFile* bereitstellen lassen.

Arbeiten Sie auf einer ISAM-Datei mit selbst erzeugten Schlüsselbeschreibungen, müssen Sie selbstverständlich sorgfältig darauf achten, dass diese zu den in der Datei definierten Schlüsseln passen.

Ein *KeyDescriptor*-Objekt ist serialisierbar und kann deshalb für entfernte Methoden-Aufrufe (RMI) verwendet werden.

Methoden

Die Methode *getPosition()* liefert die Position des Schlüsselfeldes in einem Satz.

Die Methode *getLength()* liefert die Länge des Schlüsselfeldes.

Die Methode *getName()* liefert den Namen eines benannten Schlüssels, bzw. null bei unbenannten Schlüsseln. Bei sekundären Schlüsseln wird also immer der eindeutige Name geliefert, für den primären Schlüssel hängt es von der Implementierung ab, ob ein Name geliefert wird.

Mit der Methode *hasDuplicates()* wird geprüft, ob für diesen Schlüssel gleiche Schlüsselwerte in verschiedenen Sätzen erlaubt sind.

Ob es sich bei diesem Schlüssel um einen primären oder sekundären Schlüssel handelt, wird mit den Methoden *isPrimary()* oder *isSecondary()* geprüft.

PrimaryKeyDescriptorISAM

Die Klasse *PrimaryKeyDescriptorISAM* im Package *com.fujitsu.ts.jrio.DMS* ist eine Implementierung der abstrakten Klasse *KeyDescriptor* und repräsentiert den primären Schlüssel einer ISAM-Datei. Die Klasse bietet nur genau die Methoden, die die abstrakte Klasse vorschreibt sowie Konstruktoren zur Erzeugung von Schlüsselbeschreibungen. Folgende ISAM-spezifischen Besonderheiten gelten:

- Die Schlüsselposition muss ein Wert zwischen 0 und 32767 sein. D.h. aber nicht, dass diese Werte grundsätzlich sinnvoll sind. Die tatsächlich für die I/O verwendbaren Werte hängen noch von anderen Einflussfaktoren (Blockgröße, Satzformat, Schlüssellänge) ab, die aber vom Konstruktor nicht geprüft werden können.
- Die Länge des Schlüssels muss ein Wert zwischen 1 und 255 sein.
- Der primäre ISAM-Schlüssel hat keinen Namen, daher liefert die Methode *getName()* immer null.

SecodaryKeyDescriptorISAM

Die Klasse *SecondaryKeyDescriptorISAM* im Package *com.fujitsu.ts.jrio.DMS* ist eine Implementierung der abstrakten Klasse *KeyDescriptor* und repräsentiert einen sekundären Schlüssel einer ISAM-Datei. Die Klasse bietet nur genau die Methoden, die die abstrakte Klasse vorschreibt sowie Konstruktoren zur Erzeugung von Schlüsselbeschreibungen. Folgende ISAM-spezifischen Besonderheiten gelten:

- Die Schlüsselposition muss ein Wert zwischen 0 und 32767 sein. D.h. aber nicht, dass diese Werte grundsätzlich sinnvoll sind. Die tatsächlich für die I/O verwendbaren Werte hängen noch von anderen Einflussfaktoren (Blockgröße, Satzformat, Schlüssellänge) ab, die aber vom Konstruktor nicht geprüft werden können.
- Die Länge des Schlüssels muss ein Wert zwischen 1 und 127 sein.
- Ein sekundärer ISAM-Schlüssel muss einen eindeutigen Namen von maximal 8 Zeichen Länge haben, der den DMS-Vorschriften genügt. Groß-/Kleinschreibung wird bei diesen Namen ignoriert und der Name wird von *getName()* immer in Großbuchstaben geliefert.

4.2.6.2 KeyValue

Die Klasse *KeyValue* beschreibt einen konkreten Schlüsselwert. Mit jedem Schlüsselwert ist eine Schlüsselbeschreibung assoziiert. Die Klasse bietet Methoden zum Manipulieren des Schlüsselwertes und zum Abfragen der Eigenschaften der assoziierten Schlüsselbeschreibung.

Ein *KeyValue*-Objekt kann dazu verwendet werden, um mit diesem Schlüssel einen Satz einer index-sequentiellen Datei zu selektieren.

Ein *KeyValue* Objekt ist serialisierbar und kann deshalb für entfernte Methoden-Aufrufe (RMI) verwendet werden.

Konstruktoren

Beim Erzeugen eines *KeyValue*-Objektes wird der Schlüsselwert mit den Daten des Benutzers gefüllt. Diese können als Byte-Array oder String angegeben werden. Gibt der Benutzer keine Daten an oder sind die angegebenen Daten kürzer als der Schlüssel, so wird der komplette Schlüsselwert mit Null-Bytes bzw. Leerzeichen gefüllt. Wenn die Daten länger als der Schlüssel sind, so werden nur so viele Daten übernommen, wie in den Schlüssel passen.

Gibt der Benutzer die Daten als String an, aber kein Encoding für die Umwandlung von Text in Daten, so wird das systemabhängige Standard-Encoding (im BS2000 ist der Standardwert *OSD_EBCDIC_DF04_1*) verwendet.

Methoden zum Manipulieren des Schlüsselwertes

Die Methoden *setValue()* füllen den Schlüssel mit den angegebenen Daten des Benutzers. Gibt der Benutzer keine Daten an, so wird der komplette Schlüsselwert mit Null-Bytes gefüllt. Wenn die Daten kürzer als der Schlüssel sind, so wird der Rest mit einem Füll-Byte aufgefüllt. Das Füll-Byte kann vom Benutzer mitgegeben werden, ansonsten wird ein NullByte verwendet. Wenn die Daten länger als der Schlüssel sind, so werden nur so viele Daten übernommen, wie in den Schlüssel passen.

Die Methoden *setStringValue()* füllen den Schlüssel mit den konvertierten Daten des angegebenen String. Gibt der Benutzer keine Daten an, so wird der komplette Schlüsselwert mit Leerzeichen gefüllt. Wenn die Daten kürzer als der Schlüssel sind, so wird der Rest mit Leerzeichen aufgefüllt. Wenn die Daten länger als der Schlüssel sind, so werden nur so viele Daten übernommen, wie in den Schlüssel passen.

Wenn kein Encoding für die Umwandlung von Text in Daten vom Benutzer angegeben wurde, so wird das systemabhängige Standard-Encoding (im BS2000 ist der Standardwert *OSD_EBCDIC_DF04_1*) verwendet.

Mit den Methoden *getValue()* wird der Schlüsselwert dieses Schlüssels geliefert. Der Schlüsselwert wird entweder in einen vom Benutzer bereitgestellten Puffer übertragen oder als Kopie des Wertes geliefert. Da der Schlüsselwert also in jedem Fall kopiert wird, heißt das, dass Manipulationen auf dem gelieferten Ergebnis keinen Einfluss auf das Objekt haben, aus dem der Wert stammt. Soll der Wert im Objekt geändert werden, muss anschließend die Methode *setValue()* benutzt werden.

Mit den Methoden *getStringValue()* wird der Schlüsselwert dieses Schlüssels umgewandelt in einen String geliefert. Wenn kein Encoding für die Umwandlung von Daten in Text vom Benutzer angegeben wurde, so wird das systemabhängige Standard-Encoding (im BS2000 ist der Standardwert *OSD_EBCDIC_DF04_1*) verwendet.

Methoden zum Ermitteln der Schlüsseleigenschaften

Die Methode *getPosition()* liefert die Position des Schlüsselfeldes in einem Satz.

Die Methode *getLength()* liefert die Länge des Schlüsselfeldes.

Die Methode *getKeyDescriptor()* liefert die mit dem Schlüsselwert assoziierte Schlüsselbeschreibung.

4.2.6.3 KeyedAccessRecordFile

Ein *KeyedAccessRecordFile*-Objekt repräsentiert eine für index-sequentiellen Zugriff geöffnete Datei. Die Datei wird beim Erzeugen des Objektes (siehe Abschnitt „Öffnen und Schließen einer Datei") implizit geöffnet.

Die *KeyedAccessRecordFile*-Klasse bietet Methoden zum Lesen, Schreiben und Löschen von Sätzen dieser Datei. Zusätzlich gibt es Methoden für die Handhabung von Schlüsseln und zum Schließen der Datei.

Die Datei, die geöffnet werden soll, muss im zugrunde liegenden Dateisystem schon existieren. Für die Erzeugung einer Datei muss die Methode *createNewFile()* der Klasse *RecordFile* benutzt werden.

Beim Öffnen einer Datei für den index-sequentiellen Zugriff kann die spezifische Zugriffsrichtung eingeschränkt werden und es kann das Löschen des Datei-inhalts einer existierenden Datei angefordert werden.

Die folgenden Open-Modi werden mit dieser Klasse erlaubt:

- **INPUT**
  Nach dem Öffnen der Datei sind nur Lese-Operationen erlaubt.
- **OUTIN**
  Nach dem Öffnen sind sowohl Schreib- als auch Lese-Operationen erlaubt. Der vollständige Dateiinhalt wird beim Öffnen der Datei gelöscht.
- **INOUT**
  Nach dem Öffnen sind sowohl Lese- als auch Schreib-Operationen erlaubt. Beim Öffnen der Datei bleibt der Dateiinhalt unverändert.

Nach dem Schließen der Datei sollte das *KeyedAccessRecordFile*-Objekt nicht mehr benutzt werden.

Öffnen und Schließen einer Datei

Bei der Konstruktion eines *KeyedAccessRecordFile*-Objektes wird die als *RecordFile*-Objekt angegebene Datei mit der angegebenen Zugriffsmethode oder den angegebenen Zugriffsparametern im spezifizierten Modus geöffnet.

Die Datei muss im darunter liegenden Dateisystem bereits existieren und die Zugriffsmethode muss zu diesem Dateisystem gehören und muss für diese Datei erlaubt sein. Der Benutzer muss die für den angegebenen Open-Modus erforderlichen Zugriffsrechte auf die Datei besitzen. Falls ein Security Manager aktiv ist und dessen Einschränkungen für diese Datei mit dem angegebenen Open-Modus kollidieren, so wird eine *Exception* ausgelöst (siehe Abschnitt „Sicherheit").

Werden zum Öffnen Zugriffsparameter angegeben, so werden diese beim Öffnen berücksichtigt, soweit nicht die Datei-Parameter Vorrang haben. Nach dem Öffnen werden sie mit den entsprechenden Werten der geöffneten Datei aktualisiert.

Die Methode *close()* schließt die index-sequentielle Datei. Danach sind keine E/A-Operationen über dieses *KeyedAccessRecordFile*-Objekt mehr möglich.

Besonderheiten des DMS-Dateisystems

Die Shared-Update-Verarbeitung (siehe Abschnitt „Shared-Update-Verarbeitung" sowie Abschnitt „AccessParameter") eines *KeyedAccessRecordFile* ist für alle Open-Modi (*INPUT*, *INOUT*, *OUTIN*) möglich. Der Open-Modus *OUTIN* ist allerdings nur für die Anwendung erlaubt, die die Datei als erste öffnet. Wenn die Datei im Open-Modus *INPUT* eröffnet wurde, erfolgen alle Zugriffe ohne Sperre. Es können also keine Zugriffskonflikte auftreten. Es muss jedoch damit gerechnet werden, dass der Inhalt des Satzes zwischenzeitlich von einer anderen Anwendung geändert wurde. Bei den anderen Open-Modi erfolgen Lese- und Schreibzugriffe mit impliziter Sperre, was bei Zugriffskonflikten je nach mittels *setWaitMode()* eingestellten Option zu einer *RecordLockedException* bzw. zum Warten des Threads oder der gesamten Anwendung

führen kann. Sperren werden nach dem Schreiben oder Löschen des gesperrten Satzes implizit wieder aufgehoben. Sie können aber auch explizit mit *unlock()* aufgehoben werden. Beim Lesen eines Satzes wird eine noch bestehende Sperre für einen anderen Satz ebenfalls aufgehoben. Einzelheiten entnehmen Sie der jeweiligen Schnittstellenbeschreibung in der mitgelieferten JAVADOC-Dokumentation.

Methoden zum Lesen von Sätzen

Alle *read*-Methoden werden in zwei Ausprägungen angeboten. Einmal eine Variante, bei der der gelesene Satz in einem neu erzeugten *Record*-Objekt als Ergebnis bereitgestellt wird und einer Variante, bei der ein vom Aufrufer als Argument übergebenes *Record*-Objekt mit den Daten des gelesenen Satzes gefüllt wird.

Wird ein Satzpuffer neu angelegt, so hat er genau die Größe der gelesenen Daten. Stellt der Aufrufer das *Record*-Objekt bereit, muss er dafür sorgen, dass der Satzpuffer groß genug ist, um die Daten des zu lesenden Satzes aufzunehmen. Wenn der angegebene Satzpuffer zu klein ist, um die kompletten Daten aufzunehmen, wird eine Ausnahme ausgelöst und es werden keine Daten übertragen.

Bei den *read*-Methoden, bei denen ein *KeyValue*- bzw. *KeyDescriptor*-Objekt angegeben werden kann, werden solche Argumente nur akzeptiert, wenn sie zu dieser Datei passen (siehe Methoden *getPrimaryKeydescriptor()* und *getSecondaryKeydescriptor()*).

Die Methoden *read()* lesen den Satz, der durch den angegebenen Schlüsselwert ausgewählt wird. Gibt es in der Datei mehrere Sätze mit dem gleichen Schlüsselwert, wird der erste geliefert. Als Schlüsselwert kann sowohl ein Wert des primären als auch ein Wert eines sekundären Schlüssels angegeben werden.

Die Methoden *readNext()* lesen den jeweils nächsten Satz gemäß der Ordnung, die durch das gegebene Argument bestimmt wird. Es gibt drei Varianten dieser Methoden:

- Wird kein Ordnungsargument angegeben, so wird der nächste Satz gelesen, der durch die Ordnung des primären Schlüssels definiert ist. Wenn diese Methode als erste Operation nach dem Öffnen der Datei gerufen wird, so wird der Satz mit dem kleinsten verfügbaren primären Schlüsselwert gelesen, ansonsten liest diese Operation den nachfolgenden Satz des zuletzt gelesenen Satzes, falls der zuletzt gelesene Satz ebenfalls über den primären Schlüssel gelesen wurde (ansonsten ist das Verhalten zugriffsmethodenspezifisch). Dies ist eine Methode, um Sätze sequentiell zu lesen, die den gleichen Schlüsselwert enthalten.
- Wird als Ordnungsargument eine Schlüsselbeschreibung angegeben, so wird der nächste Satz gelesen, der durch die Ordnung des primären bzw. sekundären Schlüssels der gegebenen Schlüsselbeschreibung definiert ist. Wenn diese Methode sofort nach dem Öffnen der Datei gerufen wird, wird der Satz mit dem kleinsten verfügbaren Schlüsselwert, gemäß der gegebenen Schlüsselbeschreibung, gelesen, ansonsten liest diese Operation den nachfolgenden Satz des zuletzt gelesenen Satzes, falls der zuletzt gelesene Satz über die gleiche Schlüsselbeschreibung gelesen wurde (ansonsten ist das Verhalten zugriffsmethodenspezifisch). Dies ist eine Methode, um Sätze sequentiell zu lesen, die den gleichen Schlüsselwert enthalten.
- Wird als Ordnungsargument ein Schlüsselwert angegeben, so wird der Satz mit dem nächst größeren Schlüssel, gemäß der Ordnung der zugehörigen Schlüsselbeschreibung, gelesen.

Die Methoden *readPrevious()* lesen analog den Methoden *readNext()*, jedoch den jeweils vorherigen Satz statt des nachfolgenden Satzes.

Methoden zum Schreiben und Löschen von Sätzen

Beim Schreiben von Sätzen in eine index-sequentielle Datei ist die Position des geschriebenen Satzes durch die im Satz enthaltenen Schlüsselfelder bestimmt.

Die Methode *write()* schreibt einen Satz in die Datei. Wenn es schon einen Satz mit gleichem primären Schlüsselwert gibt und keine doppelten Schlüssel für den primären Schlüssel erlaubt sind, so wird der existierende Satz ersetzt. Wenn doppelte Schlüssel erlaubt sind und der Satz schon existiert, wird der Satz nach dem letzten Satz mit dem gleichen primären Schlüsselwert angefügt.

Die Methode *writeNew()* schreibt einen Satz in die Datei, aber nur, wenn noch kein Satz in der Datei mit dem gleichen primären Schlüssel existiert.

Die Methode *writeBack()* überschreibt einen direkt vorher gelesenen Satz in der Datei. Es dürfen zwischen der Lese- und Schreib-Operation keine Änderung am primären Schlüsselfeld des Satzes durchgeführt werden. Bei der *Shared-Update-Verarbeitung* wird ein existierender Satz nur dann überschrieben, wenn die für ihn beim Lesen gesetzte Sperre noch besteht. Andernfalls wird eine *RecordNotLockedException* ausgelöst.

Die Methode *delete()* löscht den Satz, der durch den angegebenen Schlüsselwert ausgewählt wird. Gibt es in der Datei mehrere Sätze mit dem gleichen Schlüsselwert, wird der erste gelöscht. Als Schlüsselwert kann sowohl ein Wert des primären als auch ein Wert eines sekundären Schlüssels angegeben werden.

Methoden zum unbedingten Aufheben einer Sperre

Die Methode *unlock()* dient zum expliziten Aufheben einer durch eine Leseoperation implizit gesetzten Sperre bei der Shared-Update-Verarbeitung.

Methoden zum Ermitteln von Schlüsselbeschreibungen

Die Methode *getPrimaryKeyDescriptor()* liefert die Schlüsselbeschreibung für den primären Schlüssel dieser Datei.

Die Methode *getSecondaryKeyDescriptor()* liefert die Schlüsselbeschreibung für den sekundären Schlüssel mit dem angegebenen Namen.

Die Methode *getKeyDescriptorNames()* liefert eine Liste der Namen aller sekundären Schlüssel dieser Datei.

Methoden zum Erzeugen und Löschen von sekundären Schlüsseln

Die Methoden *createSecondaryKey()* erzeugen für diese index-sequentielle Datei einen neuen sekundären Schlüssel mit den angegebenen Parametern. Es gibt zwei Parameter-Varianten, die eine Variante ist, dass alle Felder des *KeyDescriptor*-Objektes (Name, Schlüsselposition, -länge und die Angabe, ob für diesen Schlüssel gleiche Schlüsselwerte in verschiedenen Sätzen erlaubt sind) einzeln angegeben werden und die zweite Variante ist, dass ein *KeyDescriptor*-Objekt für einen sekundären Schlüssel angegeben wird. Mit der zweiten Variante können Sie z.B. die Attribute eines sekundären Schlüssels einer anderen Datei benutzen, um einen entsprechenden sekundären Schlüssel in dieser Datei zu erzeugen

Die Methode *deleteSecondaryKey()* löscht den angegebenen sekundären Schlüssel dieser index-sequentiellen Datei.

Die Methoden *createSecondaryKey()* und *deleteSecondaryKey()* erfordern den exklusiven Zugriff auf die Datei und sind daher bei der Shared-Update-Verarbeitung nicht erlaubt. Sie führen zur *IOException*.

4.3 Implementierungs-Spezifika

Die in den API-Beschreibungen als implementierungs-spezifisch gekennzeichneten Eigenschaften werden in diesem Abschnitt festgelegt.

4.3.1 Dateisystem-spezifische Festlegungen

Sowohl in den API-Beschreibungen in diesem Dokument, als auch in den eigentlichen API-Spezifikationen, wird an einigen Stellen spezifiziert, dass eine Dateisystem-Implementierung besondere Festlegungen treffen kann.

Diese Festlegungen werden in der folgenden Tabelle für die in dieser Version unterstützen Dateisysteme vorgenommen. Dabei wird das UFS-Dateisystem nur zur Veranschaulichung mit aufgeführt, obwohl es derzeit noch nicht unterstützt wird.

Spezifikum	**Dateisystem DMS**	**Dateisystem UFS**
Name, wie er an den JRIO-Schnittstellen zu verwenden ist	„DMS“	„UFS“
Zugriffsmethoden	ISAM, SAM, UPAM	derzeit keine
Dateisystempräfixe	Katalogkennungen (*":catid:"*)	Wurzel-Verzeichnis '/'
Normalisierung	Kleinbuchstaben werden in Großbuchstaben und Pfadnamen *$<name>* werden nach *$.<name>* umgewandelt	. und ..-Verzeichnisse werden aufgelöst und doppelte Schrägstriche '//' vereinfacht; ein '/' am Ende des Pfadnamens wird gelöscht
absoluter Pfadname	Ergänzung des Pfadnamens mit der Katalogkennung	Ergänzung des aktuellen Verzeichnisses bei relativen Pfadnamen
kanonischer Pfadname	entweder nur Katalogkennung oder der Dateiname ergänzt um Katalog- und Benutzerkennung ggf. mit Auflösung der Standard-System-Kennung	Umwandlung wie absolute Pfadnamen und Auflösung aller symbolischen Links
leerer Pfadname	Standard-Katalogkennung des Benutzers	Wurzel-Verzeichnis '/'
normale Datei	alle Dateien sind normale Dateien	reguläre Dateien (z.B. keine Gerätedateien)
versteckte Dateien und Verzeichnisse	temporäre Dateien im DMS-Sinne	alle Dateien und Verzeichnisse deren Name mit Punkt '.' beginnen
Größe einer Datei mit der Methode *length()*	Anzahl der benutzten PAM-Seiten * 2048 (last page pointer).	Größe in bytes
Größe eines Verzeichnisses mit der Methode *length()*	immer 0	Größe in bytes
Dateiname	siehe Handbuch „Einführung in das DVS“ [8]	siehe Handbuch „POSIX, Grundlagen für Anwender und Systemverwalter“ [1]
	nicht definiert	Schrägstrich '/' bzw. "/"

Trennzeichen zwischen Pfadnamensteilen *separatorChar* bzw. *separator*		
Trennzeichen zwischen Pfadnamen *pathSeparatorChar* bzw. *pathSeparator*	Komma ',' bzw. ","	Doppelpunkt ':' bzw. ":"
Standard-Verzeichnis beim Anlegen einer temporären Datei mit der Methode *createTempFile()*	Standard-Katalogkennung des Aufrufers	Standard-Verzeichnis, das der System-Property *java.io.tmpdir* zugewiesen ist
generierter Namensteil einer temporären Datei (zwischen Suffix- und Präfixangaben)	String der Länge 7	String der Länge 7
Shared-Update-Verarbeitung	wird (mit Einschränkungen) unterstützt	wird nicht unterstützt

Tabelle 4: Dateisystem-spezifische Festlegungen

4.3.2 Zugriffsmethoden-spezifische Festlegungen

Sowohl in den API-Beschreibungen in diesem Dokument, als auch in den eigentlichen API-Spezifikationen, wird an einigen Stellen spezifiziert, dass eine Zugriffsmethode besondere Festlegungen treffen kann.

Diese Festlegungen werden in der folgenden Tabelle für die in dieser Version unterstützen Zugriffsmethoden des DMS vorgenommen.

Spezifika	**Zugriffsmethode SAM**	**Zugriffsmethode ISAM**	**Zugriffsmethode UPAM**
Name, wie er an den JRIO-Schnittstellen zu verwenden ist	„SAM“	„ISAM“	„UPAM“
Zulässige Satzformate	Satzformat mit variabler und fester Satzlänge	Satzformat mit variabler und fester Satzlänge	Satzformat mit fester Satzlänge
Maximale Satzlänge (in Abhängigkeit vom Satzformat (fest, variabel), logischem Blockformat (NO, KEY, DATA) und Blocksize (1 <= BS <= 16) - die Methode *setRecordLength* der Klassen *AccessParameter*... akzeptiert auch größere Werte, weil Blocksize oder Satzformat später geändert werden können)	fest, KEY BS * 2048 fest, NO, DATA BS * 2048 - 16 variabel, KEY: BS * 2048 -4 variabel, NO, DATA BS * 2048 - 20	fest: BS * 2048 variabel: BS * 2048 - 4 Bei voller Ausnutzung entstehen ggf. Über-laufblöcke	fest, NO, KEY BS * 2048 fest, DATA BS * 2048 (die ersten 12 Bytes enthalten Metadaten!)
Zulässige Werte für *setRecordLength*() der Klassen *AccessParameter*...	0 bis 32768 (0 bedeutet: variabel, nur durch Blocksize beschränkt)	0 bis 32768 (0 bedeutet: variabel, nur durch Blocksize beschränkt)	0 bis 32768 (0 bedeutet: Pubsetstandard) Werte != n*2048 (n=0,,16) sind nicht erlaubt
Zulässige Werte für die Methoden *setBlocksize*, *setPrimarySpaceAllocation*, *setSecondarySpaceAllocation*, *setPrimaryKeyPosition*, *setSecondaryKeyPosition* der Klassen *AccessParameter*...	Siehe API-Dokumentation des Interface *AccessParameter SAM*	Siehe API-Dokumentation des Interface *AccessParameter ISAM*	Siehe API-Dokumentation des Interface *AccessParameterUPAM*
Methode *markSupported()* Für Markieren und Repositionieren beim sequentiellen Lesen der Klasse *FileInputRecordStream*	immer false	immer false	immer false
			keine Funktion

Gepufferte Ausgabe in den Ausgabestrom schreiben mit der Methode *flush()* der Klasse *FileOutputRecordStream*	der Schreib-Puffer wird geleert	der Schreib-Puffer wird geleert	
Zulässige Werte beim Erzeugen von sekundären Schlüsseln mit der Methode *createSecondaryKey()* der Klasse *KeyedAccessRecordFile*	nicht unterstützt	Ja max. 30 Sekundärschlüssel, mit max. je 127 Byte Länge. keyPos <= 32495	nicht unterstützt
Name von sekundären Schlüsseln (Methode *createSecondaryKey()* der Klasse *KeyedAccessRecordFile*)	nicht unterstützt	8-stellig, nach DMS-Regeln, Kleinbuchstaben werden ggf. in Großbuchstaben umgesetzt.	nicht unterstützt
Setzen der Dateiposition mit der Methode *setCurrentRecordNumber()* der Klasse *RandomAccessRecordFile* hinter den letzten Satz (Wert von *get-RecordCount()*) - bzw. Schreiben an einer solchen Position	ja - ggf. werden Leersätze (bei variablem Satzformat) oder Sätze undefinierten Inhalts (bei festem Satzformat) eingefügt	nein	ja - ggf. werden Sätze undefinierten Inhalts eingefügt
Überschreiben von Sätzen mit der Methode *write()*	gleiche Satzlänge für Sätze variabler Länge		ohne Einschränkung möglich
Überschreiben von Sätzen mit der Methode *writeBack()*	-	der Primärschlüssel darf nicht verändert werden	-
Reihenfolge beim sequentiellen Lesen mit der Klasse *KeyedAccessRecordFile*	-	Schreib- oder Löschoperationen verändern die Dateiposition und sollen daher nicht zwischen sequentiellen Leseoperationen verwendet werden. Für die Reihenfolge bei sequentiellen Leseoperationen bzgl. unterschiedlicher Schlüssel gilt das ISAM-Verhalten (siehe „Einführung in das DVS“ [8])	-
Shared-Update-Verarbeitung: allgemein	nicht möglich	Möglich als:	

		FileInputStream, *FileOutputStream* *KeyedAccessRecordFile*	Nur für PAM-Dateien als *FileInputStream* oder als *RandomAccessRecordFile* möglich
Shared-Update-Verarbeitung: Open-Modi	-	*INPUT*, *INOUT* oder *OUTIN* erlaubt, *FileOutputStream* nur zum Erweitern einer Datei, *OUTIN* nur für die erste öffnende Anwendung	*INPUT* oder *INOUT* erlaubt
Shared-Update-Verarbeitung: Sperrgranulat	-	Bei *NK-ISAM* erfolgt die Sperre auf Schlüsselebene (Primärschlüssel), bei *K-ISAM* auf Blockebene	Die Sperre erfolgt auf Blockebene
Shared-Update-Verarbei-tung: weitere Besonderheiten	-	Sperren gelten für die gesamte Anwendung (nicht nur für eine Datei)	Vergrößern oder Verkleinern einer Datei ist nicht möglich

Tabelle 5: Zugriffsmethoden-spezifische Festlegung

4.3.3 Standardwerte der DMS-Zugriffsmethoden

Die folgende Tabelle gibt eine Übersicht über die Standardwerte der Zugriffsmethoden im DMS-Dateisystem eines *AccessParameter*-Objektes, das mit der Methode *getDefaultAccessParameter()* erzeugt wurde. Die Übersicht ist nach den für das Auslesen benutzten Methoden gegliedert.

Methode	**Zugriffsmethode SAM**	**Zugriffsmethode ISAM**	**Zugriffsmethode UPAM**
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

Tabelle 6: Standardwerte der DMS-Zugriffsmethoden

Der Wert 0 bei *getRecordLength()* bezeichnet bei Zugriffsmethoden SAM und ISAM den Wert „variabel - nur durch Blocksize begrenzt“, bei UPAM den „pubsetspezifischen Standard“.

Die Werte 0 bei *getBlockSize()*, 0 bei *getPrimarySpaceAllocation()* und -1 bei *getSecondarySpaceAllocation()* bezeichnen den „pubsetspezifischen Default“. Wenn die Datei angelegt ist, werden hier die aktuellen Werte eingesetzt.

4.4 Einschränkungen

Folgende explizite Einschränkungen werden für DMS unter JRIO festgelegt:

- Banddateien und Privatplatten werden nicht unterstützt.
- EAM und logische Systemdateien werden nicht unterstützt.
- Nicht alle Dateiparameter sind über JRIO manipulierbar oder einstellbar. Nur die in den API-Beschreibungen explizit genannten Parameter werden berücksichtigt. Insbesondere bei der Erzeugung neuer Dateien führt das zu Einschränkungen, wenn ganz spezielle Attribute verwendet werden sollen. Mit der Klasse *AccessParameter* und den zugehörigen Implementierungen der Zugriffsmethoden werden aber die gebräuchlichsten Parameter der einzelnen Zugriffsmethoden bereits unterstützt.
- Nur die ausgewiesenen Zugriffsmethoden und damit verbundene Dateien werden unterstützt.
- Shared-Update und Locking werden in dieser Version nicht unterstützt.
- Rückwärts-Lesen wird nur bei index-sequentiellem, aber nicht bei sequentiellem oder wahlfreiem Zugriff unterstützt.
- Bei ISAM wird die logische Wertmarkierung nicht unterstützt. ISAM-Dateien, die solche enthalten, können nicht bearbeitet werden. ISAM-Pools werden ebenfalls nicht unterstützt.
- Das undefinierte Satzformat wird nicht unterstützt. Dateien mit undefiniertem Satzformat können nicht bearbeitet werden.
- Dateigenerationsgruppen werden nicht unterstützt.

4.5 Beispiele

Die in den folgenden Abschnitten aufgeführten Beispiele sollen die verschiedenen Zugriffsarten und den allgemeinen Umgang mit den JRIO-Schnittstellen an jeweils einem oder zwei (mehr oder weniger typischen) Problemen demonstrieren.

Alle hier aufgeführten Beispiele stellen komplette Programme dar, die so auch ausführbar sind. Die Quelltexte aller Beispielprogramme werden mit dem Produkt ausgeliefert und finden sich im Unterverzeichnis *demo/jrio* des Installationsverzeichnisses. Die Programme sollten mit ihrer Inline-Dokumentation im Wesentlichen selbsterklärend sein.

4.5.1 Sequentielle Datenverarbeitung

Zur Demonstration der sequentiellen Dateibearbeitung wird ein einfaches Kopierprogramm für SAM-Dateien herangezogen. Das Programm erwartet zwei Parameter, den Namen der Datei, die zu kopieren ist, und den Dateinamen der Kopie. Existiert die Zieldatei schon, wird sie gelöscht (Vorsicht!) und neu angelegt.

Das Beispiel ist so angelegt, dass man keine Kenntnisse über die Dateiattribute, wie Satzformat oder Satzlänge, der zu kopierenden Datei benötigt. Die Fehlerbehandlung ist im Beispiel nicht besonders komfortabel ausgelegt, damit der dafür notwendige umfangreiche Code nicht von der eigentlichen Schnittstellennutzung ablenkt.

Das Programm befindet sich in der Datei *CopySAM.java*:

```
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();
   }
}
```

Das Programm kann mit dem Java-Compiler *javac* übersetzt und anschließend zum Ablauf gebracht werden. Es sind dabei keine besonderen Angaben notwendig, um die JRIO-Schnittstellen verfügbar zu haben.

4.5.2 Wahlfreie Datenbearbeitung

Zur Demonstration der wahlfreien Dateibearbeitung werden zwei Beispiele herangezogen. Das erste Programm löst die Aufgabe, einen oder mehrere Sätze aus einer SAM-Datei zu löschen. Das Programm erwartet dazu als Parameter den Namen einer existierenden Datei und die Nummer oder den Nummernbereich der Datensätze, die gelöscht werden sollen. Beachten Sie, dass auch hier Datensätze mit Null beginnend durchnummeriert werden.

Das Beispiel ist so angelegt, dass man keine Kenntnisse über die Dateiattribute, wie Satzformat oder Satzlänge, der zu bearbeitenden Datei benötigt. Die Fehlerbehandlung ist im Beispiel nicht besonders komfortabel ausgelegt, damit der dafür notwendige umfangreiche Code nicht von der eigentlichen Schnittstellennutzung ablenkt.

Das Programm befindet sich in der Datei *DeleteRecordsSAM.java*:

```
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();
    }
}
```

Das Programm kann mit dem Java-Compiler *javac* übersetzt und anschließend zum Ablauf gebracht werden. Es sind dabei keine besonderen Angaben notwendig, um die JRIO-Schnittstellen verfügbar zu haben.

Das zweite Programm gibt einen zufällig ausgewählten „Spruch des Tages“ aus einer Datei (SAM) mit Sprüchen aus. Das Programm erwartet dazu als Parameter den Namen der Datei mit den Sprüchen. Existiert diese noch nicht, wird sie mit einem Grundbestand an Sprüchen neu angelegt.

Das Beispiel ist ebenfalls so angelegt, dass man keine Kenntnisse über die Dateiattribute, wie Satzformat oder Satzlänge, der Datei benötigt. Die Fehlerbehandlung ist im Beispiel nicht besonders komfortabel ausgelegt, damit der dafür notwendige umfangreiche Code nicht von der eigentlichen Schnittstellennutzung ablenkt.

Das Programm befindet sich in der Datei *SloganOfTheDay.java*:

```
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();
    }
}
```

Das Programm kann mit dem Java-Compiler *javac* übersetzt und anschließend zum Ablauf gebracht werden. Es sind dabei keine besonderen Angaben notwendig, um die JRIO-Schnittstellen verfügbar zu haben.

4.5.3 Index-sequentielle Datenbearbeitung

Zur Demonstration der index-sequentiellen Dateibearbeitung wird ein Programm herangezogen, das die Lebensdauer von Dateien auf einer Kennung überwacht. Das Programm erwartet zwei Parameter, die Benutzerkennung, die zu überwachen ist, und den Namen der Datei, in der das Programm die Daten hinterlegen kann. Beim ersten Aufruf sollte die Datei noch nicht existieren, dann wird sie mit den für das Programm korrekten Attributen angelegt.
Das Beispiel erzeugt als Datenbank eine ISAM-Datei mit fester Satzlänge. Die Fehlerbehandlung ist im Beispiel nicht besonders komfortabel ausgelegt, damit der dafür notwendige umfangreiche Code nicht von der eigentlichen Schnittstellennutzung ablenkt.
Das Programm befindet sich in der Datei *FileHistory.java*:

```
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();
   }
}
```

Das Programm kann mit dem Java-Compiler *javac* übersetzt und anschließend zum Ablauf gebracht werden. Es sind dabei keine besonderen Angaben notwendig, um die JRIO-Schnittstellen verfügbar zu haben.

5 Aufruf der VM von der BS2000- Kommandooberfläche

In der PLAM-Bibliothek *SYSPRC.JENV.090* stehen die Prozeduren */INITIALIZE*, *DELETE* und *START* zur Verfügung.

- Mit */INITIALIZE* werden Umgebungsvariablen gesetzt, die beim Ablauf der VM benötigt werden.
- Mit *START* wird die VM gestartet. Wird */INITIALIZE* nicht vor *START* aufgerufen, so werden die Standardwerte gesetzt.
- Mit *DELETE* werden alle von */INITIALIZE* gesetzten Umgebungsvariablen gelöscht.

Die Prozeduren werden auch in einer compilierten Variante ausgeliefert, so dass der Anwender sie auch ohne das Produkt SDF-P zum Ablauf bringen kann.

Voraussetzung für den Ablauf ist, dass der Anwender die Berechtigung hat, POSIX-Programme ablaufen zu lassen, und dass er Zugriffsberechtigung auf das POSIX-Dateisystem hat, auf dem der POSIX-Teil von JENV installiert ist.

5.1 Prozedur INITIALIZE

Die Prozedur *INITIALIZE* setzt Umgebungsvariablen, die von der Java-VM ausgewertet werden. Dies geschieht, indem die entsprechenden Strukturelemente der Strukturvariablen *SYSPOSIX* gesetzt werden. Andere bereits existierende Strukturelemente dieser Struktur bleiben unverändert. Falls *SYSPOSIX* noch nicht existiert, wird sie neu angelegt.

Parameter

JAVA-HOME

Bestimmt den Wert der Umgebungsvariablen *JAVA_HOME* (siehe Kapitel „Umgebungsvariablen"). Wird der Parameter nicht angegeben oder auf *'*STD'* gesetzt, wird die Variable nicht belegt. Eine ggf. vorhandene Belegung wird gelöscht.

CLASSPATH

Bestimmt den Wert der Umgebungsvariablen *CLASSPATH* (siehe Kapitel „Umgebungsvariablen"). Wird der Parameter nicht angegeben oder auf *'*STD'* gesetzt, wird die Variable nicht belegt. Eine ggf. vorhandene Belegung wird gelöscht.

LD-LIBRARY-PATH

Bestimmt den Wert der Umgebungsvariablen *LD_LIBRARY_PATH* (siehe Kapitel „Umgebungsvariablen"). Wird der Parameter nicht angegeben oder auf *'*STD'* gesetzt, wird die Variable nicht belegt. Eine ggf. vorhandene Belegung wird gelöscht.

PWD

Setzt den Wert der Umgebungsvariablen *PWD* und bestimmt damit das *current working directory*. Wird der Parameter nicht angegeben oder auf *'*STD'* gesetzt, wird das für die Benutzerkennung mit dem Kommando / *MODIFY-POSIX-USER-ATTRIBUTES DIRECTORY=* ... eingestellte Home-Verzeichnis verwendet.

DISPLAY

Bestimmt den Wert der Umgebungsvariablen *DISPLAY*. Sie gibt die Adresse des Bildschirms an, auf dem die graphischen Ausgaben erfolgen sollen. Falls die Anwendung ohne graphische Ausgaben arbeitet, ist der Wert der Variablen irrelevant. Wird der Parameter nicht angegeben oder auf *'*STD'* gesetzt, bleibt die Variable unverändert.

SCOPE

Gibt den Geltungsbereich der Strukturvariablen *SYSPOSIX* an. Standardwert ist *'*TASK'*. Der Parameter wird direkt an den *SCOPE*-Operanden des Kommandos *DECLARE-VARIABLE* übergeben (siehe Manual „SDF-P (BS2000)“ [7]). Als Parameter sind lediglich *'*TASK'* oder *'*PROCEDURE'* mit ihren Unteroperanden sinnvoll, wobei '*PROCEDURE' nur sinnvoll ist, wenn die Prozedur mit *INCLUDE-PROCEDURE* aufgerufen wird.

Außer beim Parameter ist die Groß-Klein-Schreibung relevant. Daher müssen die Parameterwerte in Hochkommata eingeschlossen werden.

Außerdem werden implizit immer die folgenden Umgebungsvariablen gesetzt:

PROGRAM_ENVIRONMENT = 'shell'

da die Java-VM nur in diesem Modus ablauffähig ist.

HOME

auf das Home-Verzeichnis, das für die Benutzerkennung mit dem Kommando /*MODIFY-POSIX-USER-ATTRIBUTES DIRECTORY=* ... eingestellt wurde.

5.2 Prozedur START

Die Funktion *START* startet die VM mit dem Kommando *START-PROGRAM* und übergibt die gesetzten Parameter. Falls die Strukturvariable *SYSPOSIX* noch nicht existiert, wird vorher die Prozedur */NITIALIZE* mit den Standardwerten aufgerufen. Falls *SYSPOSIX* bereits existiert, wird */NITIALIZE* nicht aufgerufen. Die für den Aufruf der Tools notwendigen internen Umgebungsvariablen werden aber gesetzt.

Parameter

CMD

Muss mit einem der folgenden Werte belegt werden:
'appletviewer'
'idlj'
'jar'
'jarsigner'
'java'
'javac'
'javadoc'
'javah'
'javap'
'jconsole'
'jdb'
'jdeps'
'jimage'
'jjs'
'jlink'
'jmod'
'keytool'
'native2ascii'
'orbd'
'pack200'
'policytool'
'rmic'
'rmid'
'rmiregistry'
'schemagen'
'serialver'
'servertool'
'tnameserv'
'unpack200'
'wsgen'
'wsimport'
'xjc'

Die Werte entsprechen den Kommandos unter der Shell.

Sonstige Werte:

'?'

'help' geben einen Hilfetext in Englisch aus.

’hilfe’ gibt einen Hilfetext in Deutsch aus.

ARGS

Die Argumente zu dem obigen Kommando eingeschlossen in Hochkommata.

Die unter der Shell übliche Wildcardersetzung wird nicht unterstützt.

REDIRECT

Dieser Parameter muss verwendet werden, falls die Ein/Ausgabe umgelenkt werden soll. Dies geschieht analog zur entsprechenden Angabe unter der Shell. So legt z.B.: *REDIRECT='2>MyFile'* die Ausgabe von *stderr* auf die Datei *MyFile* um.

Siehe dazu Abschnitt „Umlenkung der Standardströme".

SYSHSI

Muss mit einem der folgenden Werte belegt werden:

’*STD’
'X86'
’S390’

Dieser Parameter legt fest, ob die S390-Variante der Java-VM oder die X86-Variante verwendet werden soll.

Standardwert: ’*STD’

bewirkt, dass die dem System entsprechende Variante verwendet wird.

INSTALLATION-ID

Benutzerkennung der JENV-Installation. Dieser Parameter muss nur dann angegeben werden, wenn das zu startende Objekt der VM nicht unter der gleichen Benutzerkennung abgelegt wurde wie die Prozedurbibliothek, in der die Prozedur *START* liegt.

Umlenkung der Standardströme

Wenn *PROGRAM_ENVIRONMENT='shell'* ist, bezeichnen die Dateinamen, in die die Standardströme umgelenkt werden, Dateien im POSIX-Dateisystem.

Eine Umlenkung auf BS2000-Dateien ist mit dem üblichen Namenspräfix */BS2/* möglich. Um die Umlenkung auf *SYSDTA*, *SYSOUT* oder *SYSLST* zu erreichen, muss dann auch dieses Präfix verwendet werden, also */BS2/(SYSDTA)*, */BS2/(SYSOUT)* bzw. */BS2/(SYSLST)*. Ohne Präfix würde z.B. die Umlenkung auf *(SYSOUT)* bewirken, dass in eine POSIX-Datei mit dem Namen *(SYSOUT)* geschrieben wird.

Entsprechendes gilt auch für Umlenkungen, die unter der Shell eine Sonderbehandlung bedeuten. Außerhalb der Shell wird alles, was rechts von *<* oder *>* steht, als Dateiname interpretiert. So erzeugt z.B. die Umlenkung *2>&1* eine Datei mit dem Namen *&1*.

Die Umlenkung von *stdout* und *stderr* auf die gleiche BS2000-Datei ist nicht möglich, bei Umlenkung auf die gleiche POSIX-Datei können Ausgabedaten verloren gehen.

Beispiel

Falls über die Datei */MyDir/MyTest/Test1.html* ein Applet gestartet werden soll und das Terminal die symbolische Adresse *ABCD1234* hat, so kann dies wie folgt geschehen:

```
/CALL-PROCEDURE *LIB($TSOS.SYSPRC.JENV.090,INITIALIZE),
                (PWD='/MyDir/MyTest ',DISPLAY='ABCD1234:0.0')
/CALL-PROCEDURE *LIB($TSOS.SYSPRC.JENV.090,START),
                 (CMD='appletviewer',ARGS='Test1.html')
```

5.3 Prozedur DELETE

Die Prozedur *DELETE* löscht alle Elemente der Strukturvariablen *SYSPOSIX*, die von der Prozedur *INITIALIZE* gesetzt werden. Falls die Struktur *SYSPOSIX* anschließend keine Elemente enthält, wird sie selbst gelöscht.

Parameter

SCOPE

Gibt den Geltungsbereich der Strukturvariablen *SYSPOSIX* an. Standardwert ist *'*TASK'.* der Wert *'*PROCEDURE'* muss nur angegeben werden, wenn auch bei *INITIALIZE* dieser Wert angegeben wurde (siehe Abschnitt „Prozedur INITIALIZE").

5.4 Aufruf der VM über das Invocation-API

Soll ein C oder C++ Programm, das die VM über das Invocation-API aufruft, mit *START-PROGRAM* gestartet werden, so müssen die Umgebungsvariablen ebenfalls mit der Prozedur *INITIALIZE* gesetzt werden. Beim Kommando *START-PROGRAM* sind folgende Operanden zu setzen:

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

> Ein C/C++-Programm muss mit dem Java-Runtime-Adapter gebunden werden und nicht mit den normalen CRTE-, C++ oder Socket-Libraries (siehe Abschnitt „Invocation-API").

5.5 Besonderheiten

Beim Aufruf eines BS2000-Programmes mit *START-PROGRAM* wird weder die Datei */etc/profile* noch die Datei *.profile* des Anwenders ausgeführt. Das hat zur Folge, dass sich ein Programm unter Umständen anders verhält, als wenn es unter der Shell gestartet würde. Insbesondere wenn in den Profiles mit *umask* die Dateizugriffsrechte von neu angelegten Dateien eingeschränkt werden, hat dies bei den mit *START-PROGRAM* gestarteten Programmen keine Wirkung. D.h. diese Programme erzeugen dann Dateien mit größeren Zugriffsrechten, als beabsichtigt. Es gibt derzeit auch keine Abhilfe dafür. Da die Tools von der Prozedur *START* mit dem Kommando *START-PROGRAM* gestartet werden, sind sie entsprechend davon betroffen.

Die Umgebungsvariable *PATH* ist nach *START-PROGRAM* nicht gesetzt. Das hat zur Folge, dass die Erzeugung eines neuen Prozesses mit *fork/exec* unter Umständen nicht möglich ist, wenn das zu startende Programm nicht gefunden werden kann. Eine Lösung dieses Problems ist möglich, indem Sie vor dem Aufruf von *START-PROGRAM* die SDF-P-Variable *SYSPOSIX.PATH* auf den in der Shell verwendeten Wert setzten, oder im Programm bei *exec()* einen vollständigen Pfadnamen angeben. In Java wirkt sich dieses Problem bei der Methode *Runtime.exec()* aus.

Beispiel

Die folgende Anweisung ist nicht ausführbar, wenn die Umgebungsvariable *PATH* nicht korrekt gesetzt wurde:

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

Die folgende Anweisung behebt das Problem:

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

Auch wenn die VM mit *START-PROGRAM* gestartet wird, geht die Ein/Ausgabe standardmäßig auf Dateien des POSIX-Dateisystems. BS2000-Dateien können über das Package JRIO geöffnet werden. Die Klassendateien müssen immer im POSIX-Dateisystem liegen.

6 JNI unter BS2000

Die folgenden Abschnitte beschreiben die Besonderheiten, die ein Benutzer der Java native Schnittstellen (JNI) im BS2000 zu beachten hat. Auf die allgemeine (also betriebssystemunabhängige) Benutzung der native Schnittstellen wird hier nicht tiefgehend eingegangen.

Dafür stehen Spezifikationen und Tutorials im Internet und auf dem Buchmarkt zur Verfügung.

Die Verwendung des JNI für reale Anwendungen ist nicht einfach, weil komplexe Wechselwirkungen zwischen der Java- und der C-Umgebung möglich sind. Vor einer Entscheidung für die Nutzung des JNI sollten Alternativen sorgfältig diskutiert werden.

6.1 Die Ausprägungen des JNI

Es werden die JNI-Spezifikationen ab Version 1.2 unterstützt.

6.2 Java-Datentypen in C

Zwischen den primitiven Java-Datentypen und der native C-Darstellung wurde ein Mapping definiert, das im Wesentlichen auch für BS2000 gilt. Die folgende Tabelle gibt eine Übersicht und weist auf Besonderheiten hin:

Java-Typ	C-Typ	Kompatibler C-Typ	Bemerkung
`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`	ab C/C++ V3.0B
`float`	`jfloat`	`float`	IEEE
`double`	`jdouble`	`double`	IEEE
`void`	`void`	`void`	

Tabelle 7: Java-Datentypen in C

Für komplexe Datentypen definiert JNI entsprechende Zugriffs- und Konvertierungsfunktionen, die im BS2000 analog zu anderen Betriebssystemen benutzt werden können. Eine besondere Rolle spielen dabei Strings, weil die von Java verwendete UTF8-Codierung von Unicode-Strings zwar eine enge Verwandschaft zu ASCII, aber keinerlei Ähnlichkeit zu einer EBCDIC-Codierung haben. Ein C-Programmierer in einer ASCII-Umgebung (Unix- und Windows-Systeme) erliegt daher sehr leicht der Versuchung, diese Ähnlichkeit zu nutzen, mit der Konsequenz, dass solche C-Programme nicht ohne weiteres im BS2000 (also EBCDIC-Umgebung) verwendet werden können.

Bei der Verknüpfung von C-Code und Java über das JNI kommt es im BS2000 zwangsläufig zum Zusammentreffen verschiedener Daten-Codierungen. Der Anwender muss für sich entscheiden, wo er entsprechende Übergänge zwischen den Daten-Darstellungen vollziehen möchte. Die wesentlichen und kritischen Übergänge sind in der folgenden Tabelle dargestellt:

Daten	Darstellung in Java	normale Darstellung in BS2000	alternative Darstellung in BS2000
Ganze Zahlen	32 und 64 Bit	32 Bit	32 und 64 Bit
Gleitkommazahlen	IEEE-Format	/390-Format	IEEE-Format
Strings, Zeichen	Unicode, UTF-8, ASCII	EBCDIC	ASCII

Tabelle 8: C-Code in Java und BS2000

Damit der Anwender in der Wahl der Übergangsstelle frei ist, werden zu den verschiedenen Themen entsprechende Hilfsmittel durch Compiler und Laufzeitsysteme bereitgestellt.

Typischerweise wird ein Anwender der JNI-Schnittstelle diesen Übergang entweder direkt an der JNI-Schnittstelle vollziehen und seinen gesamten C-Code in der üblichen BS2000-Umgebung laufen lassen. Alternativ kann er Teile

seines C-Codes (oder auch alles) in der näher an Java (und Unix-Systemen) orientierten alternativen Darstellung laufen lassen und z.B. nur beim Übergang zu Legacy-Anwendungen (Nutzung altbewährter Software) entsprechende Konvertierungen vornehmen.

Im Folgenden werden für die verschiedenen Datenarten die angebotenen Unterstützungen dargestellt.

6.2.1 Ganze Zahlen

Der Java-Datentyp *long* ist ein 64-Bit-Datentyp, der im JNI durch die C-Datentypen *jlong* dargestellt wird.

Der C/C++-Compiler (ab Version 3.0B) unterstützt den Datentyp *longlong* bzw. *int64_t*, der mit den oben genannten Datentypen (also *jlong*) kompatibel ist. Damit ist ohne weitere Vorkehrungen die Benutzung dieser Daten in C möglich. Der Umfang der Unterstützung durch C-Laufzeitsystem-Funktionen ab CRTE V2.1B kann der entsprechenden CRTE-Dokumentation entnommen werden.

6.2.2 Gleitkommazahlen

Die Java-Datentypen *float* und *double* sind Gleitkomma-Datentypen, die im JNI durch die C-Datentypen *jfloat* und *jdouble* dargestellt werden.

Diese Datentypen sind formal kompatibel zu den C-Datentypen *float* und *double*. Da aber ihre Darstellung im IEEE-Format (statt /390-Format) erfolgt, können sie ohne Vorkehrungen nicht in C verwendet werden.

Neben expliziten Konvertierungsmöglichkeiten stehen für die Unterstützung des IEEE-Formates entsprechende Compiler- und Laufzeitsystem-Erweiterungen bereit, die es erlauben, direkt mit diesem Zahlenformat in C zu arbeiten.

Explizite Konvertierung

Für die explizite Konvertierung zwischen Gleitkommazahlen im IEEE-Format und im /390-Format stehen einige Funktionen zur Verfügung. Diese sind deklariert in der Header-Datei *ieee_390.h*, die Bestandteil der CRTE-Distribution ist. Diese Konvertierungsfunktion sind im Handbuch „CRTE“ [3] beschrieben.

Beispiel

Das folgende Beispiel zeigt die Verwendung in einer nativen Methode, die arithmetische Manipulationen an einer Gleitkommazahl vornimmt. Auf Java-Seite sei die Methode deklariert als:

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

Das zugehörige C-Programm könnte folgendermaßen aussehen:

```
#include <jni.h>
#include ".....h" // javah generierter 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);
}
```

Es sei noch darauf hingewiesen, dass im obigen Beispielcode noch jegliche Fehlerbehandlung für die möglichen Konvertierungsfehler fehlt.

IEEE-Gleitkommazahlen im C-Code

Der C/C++-Compiler ab der Version V3.0B bietet die Möglichkeit, alternativ zum /390-Format für Gleitkommazahlen, auch Code für das IEEE-Format zu erzeugen. Die über die Compiler-Option *-Kieee_floats* gesteuerte Einstellung gilt für die gesamte CompilierungsEinheit (Source-Datei).

Diese Option hat nur eine Wirkung auf Gleitkomma-Konstanten im Source-Code sowie Arithmetik, Typumwandlung oder Vergleich von Gleitkommazahlen. Sie hat keinen Einfluss auf die Übergabe solcher Daten an andere Funktionen oder einfache Zuweisungen.

Das Setzen dieser Option bewirkt außerdem implizit, dass C-Library-Funktionen mit Gleitkomma-Argumenten und /oder Gleitkomma-Ergebnis in einer Variante für IEEE-Arithmetik benutzt werden kann.

Die gesamte Arithmetik wird über entsprechende Emulationsroutinen abgewickelt. Dies gilt auch für SQ-Maschinen, solange über Asstran noch keine Generierung von native Code für die entsprechenden Befehle möglich ist. Dies geht selbstverständlich erheblich zu Lasten der Performance. C-Programme, die intensiv Gleitkomma-Arithmetik nutzen, sollten daher nicht in diesem Modus arbeiten.

Beispiel

Das vorherige Beispiel könnte dann folgendermaßen realisiert sein:

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

JNIEXPORT jdouble JNICALL
Java_..._manipulate(JNIEnv *env, jobject jthis, jdouble num)
{
    return (num < 1.7)? num * 3.4 : num - 1.0;
}
```

Die Übersetzung muss mit der C-Compiler-Option *-Kieee_floats* erfolgt sein.

IEEE-Gleitkommazahlen im C-Laufzeitsystem

Das C-Laufzeitsystem bietet neben den in *ieee_390.h* deklarierten Konvertierungsroutinen alle wesentlichen XPG4-Funktionen, die mit Gleitkommazahlen umgehen, in einer Variante für IEEE-Arithmetik an. Beim Setzen der Compiler-Option für IEEE-Nutzung werden normalerweise automatisch die entsprechenden Library-Funktionen benutzt, ohne dass der Anwender dafür etwas tun muss. Für den Mischbetrieb können Sie das Verhalten auch verändern (siehe Handbuch „CRTE“ [3]).

Beispiel

Das folgende Beispiel zeigt die Verwendung der IEEE-Version der C-Funktion *tanh* in einer native Methode zur Ermittlung des Tangens hyperbolicus in einer Java-Klasse. Auf Java-Seite sei die Methode deklariert als:

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

Das zugehörige C-Programm könnte folgendermaßen aussehen:

```
#include <math.h>
#include <jni.h
#include ".....h" // javah generierter Header

JNIEXPORT jdouble JNICALL

Java_..._tanhyp(JNIEnv *env, jobject jthis, jdouble num)
{
    //printf("tan_hyp called with: %e\n",num);
    return tanh(num);
}
```

In dieser Form muss die Übersetzung mit der C-Compiler-Option *-Kieee_floats* erfolgt sein.

6.2.3 Strings

Der Java-Datentyp *string* wird im JNI als Datentyp *jstring* bereitgestellt. Dieser Typ kann in C nicht direkt verwendet werden, insbesondere hat er nichts mit dem C-Datentyp *char ** gemein. Um den String in eine Form zu konvertieren, die in C bearbeitbar ist, müssen die entsprechenden JNI-Schnittstellen zur Konvertierung benutzt werden (siehe JNI-Dokumentation).

Der Java-Datentyp *char* wird an der JNI-Schnittstelle als Datentyp *jchar* bereitgestellt. Dieser ist mit dem C-Datentyp *unsigned short* kompatibel und repräsentiert ein Zeichen in Unicode-Darstellung. Unicode ist für die ersten 256 Zeichen identisch mit der ASCII-Codierung nach ISO8859-1. Für Unicode-Zeichen außerhalb dieses Bereiches gibt es in C/C++ im BS2000 keinerlei Support, die Verarbeitung solcher Zeichen muss also vom Anwender selbst vorgenommen werden.

Eine besondere Rolle spielt die UTF-8-Darstellung von Unicode, die von Java im JNI teilweise verwendet wird. In der UTF-8-Darstellung werden Unicode-Zeichen in ein, zwei oder drei Bytes verschlüsselt. Dabei sind die Unicode-Zeichen mit der Codierung 1 bis 127 mit diesem Wert in einem Byte dargestellt, was wiederum exakt der ASCII-Codierung dieser Zeichen entspricht.

Außerdem werden von Java UTF-8 Bytefolgen immer mit einem NULL-Byte abgeschlossen, was die Verarbeitung als C-Strings ermöglicht. Um das zu erreichen, wird das Unicode-NULL-Zeichen in zwei Byte verschlüsselt, um eine Verwechslung mit dem String-Ende in C zu vermeiden, denn im Gegensatz zu C dürfen in Java Strings durchaus auch NULL-Zeichen enthalten.

Für die Verarbeitung von UTF-8-Bytefolgen in C gelten also folgende einfache Regeln:

- Das NULL-Byte kennzeichnet das Ende der Bytefolge und ist zwingend notwendig.
- Bytes, für die die Funktion *isascii_ascii()* den Wert „`true`“ liefert (1-127), sind auch tatsächlich ASCII-Zeichen nach ISO8859-1
- Für alle anderen Bytes gilt, dass sie Bestandteil von Mehr-Byte-Folgen zur Darstellung von Unicode-Zeichen außerhalb des Bereiches 1-127 sind. Diese müssen vom Anwender selbst interpretiert werden.

Da fast alle diese Konvertierungsfunktionen Zeichenfolgen zumindest in einer zu ASCII aufwärtskompatiblen Form darstellen, spielt die Code-Umsetzung von ASCII nach EBCDIC und umgekehrt im BS2000 eine besondere Rolle. Dies gilt natürlich nicht nur für Strings, sondern auch z.B. für Byte-Arrays oder Characters (*jchar*).

Wenn hier von ASCII die Rede ist, dann ist immer der Zeichensatz ISO8859-1 (ISO-Latin1) oder sein 7-Bit Ableger (ISO 646) gemeint. Bei EBCDIC ist der Zeichensatz DF04-1 (Internationale Referenzversion) mit vertauschtem 0x15 und 0x25 bzw. sein 7-Bit-Ableger DF03-1 gemeint.

Neben expliziten Konvertierungsmöglichkeiten stehen für die Unterstützung von ASCII-Strings entsprechende Compiler- und Laufzeitsystem-Erweiterungen bereit, die es erlauben, direkt mit ASCII-Strings und Zeichen in C zu arbeiten.

Explizite Konvertierung

Die Konvertierungsfunktionen des JNI (siehe „Java ™ Native Interface “ [13]) arbeiten im BS2000 exakt wie spezifiziert. Sie liefern bzw. erwarten immer Unicode oder UTF-8.

Für die explizite Konvertierung zwischen ASCII (8859-1) und EBCDIC (DF04-1) stehen einige Funktionen im CRTE zur Verfügung. Diese sind deklariert in einem Headerfile *<ascii_ebcdic.h>*, das Bestandteil der CRTE-Distribution ist. Diese Konvertierungsfunktionen sind im Handbuch „CRTE“ [3] beschrieben.

Beispiel

Das folgende Beispiel zeigt die Verwendung in einer native Methode, die den Wert einer Umgebungsvariablen ermittelt und von diesem das Präfix *JAVA_* entfernt. Auf Java-Seite sei die Methode deklariert als:

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

Das zugehörige C-Programm könnte folgendermaßen aussehen:

```
#include <jni.h>
#include ".....h"         // von javah generierter Header
#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;
}
```

Der obige Beispielcode enthält keinerlei Fehlerbehandlung. Es wird implizit davon ausgegangen, dass in allen Strings nur Zeichen des 7-Bit-ASCII-Zeichensatzes vorkommen. Außerdem ist dieser Code natürlich stark BS2000-spezifisch.

ASCII-Strings im C-Code

Der C/C+-Compiler ab der Version 3.0B hat die Möglichkeit, alternativ zum normalen EBC-DIC-Encoding für String- und Character-Literale auch ein entsprechendes ASCII-Encoding zu generieren. Diese Einstellung muss für eine ganze Compilations-Einheit (Source-Datei) gelten und wird über die Compiler-Optionen *-Kliteral_encoding_ascii* bzw. *-Kliteral_encoding_ascii_full* gesteuert. Der Unterschied der beiden Optionen bezieht sich auf die Behandlung von Oktal- und Hexadezimal-Sequenzen in solchen Literalen. Die Umwandlung von solchen Literalteilen unterbleibt bei *-Kliteral_encoding_ascii*.

ASCII-Strings im C-Laufzeitsystem

Das C-Laufzeitsystem hat neben den o.g. Konvertierungsroutinen noch weitere Unterstützung für die Nutzung von ASCII-Strings und -Zeichen. Alle wesentlichen XPG4-Funktionen, die mit Strings oder Zeichen arbeiten oder solche

liefern, stehen in einer Variante für ASCII-Codierung bereit. Beim Setzen einer der in Abschnitt „ASCII-Strings im C-Code" beschriebenen Compiler-Optionen für ASCII-Nutzung werden normalerweise automatisch die entsprechenden Library-Funktionen benutzt, ohne dass der Anwender dafür etwas tun muss. Für den Mischbetrieb können Sie das Verhalten auch verändern (siehe Handbuch „CRTE“ [3]).

Ist gleichzeitig die Compiler-Option *-Kieee_floats* gesetzt, so werden die kombinierten AS-CII/IEEE-Varianten benutzt (z.B. bei *printf*).

Ab dem C-Compiler V3.1A und CRTE V2.4C werden die Argumente des Vektors *argv[]* bei der Compilierung des Main-Programms mit einer der in Abschnitt „ASCII-Strings im C-Code" beschriebenen Compiler-Optionen als ASCII-Strings übergeben. Die globalen Variablen des C-Laufzeitsystems *tzname* und die Strings von *environ* werden als ASCII-Strings gespeichert. Die explizite Konvertierung von *argv[]* entfällt damit.

Falls explizit auf die Strings der globalen Variablen *tzname* oder *environ* zugegriffen wird, so ist zu beachten, dass diese Strings seit JENV V1.4B als ASCII-Strings abgelegt sind (vorher EBCDIC-Strings). Vom expliziten Zugriff auf die Variable *environ* wird aber durch den Technical Standard „The Single UNIX Specification“ abgeraten (siehe „X /Open System Interface (XSI) Specification“ [16]). Der implizite Zugriff über *getenv()* und *putenv()* funktioniert wie bisher kompatibel zu Vorversionen.

Beispiel

Das obige C-Programm könnte mit diesen Möglichkeiten jetzt folgendermaßen aussehen:

```
#include <jni.h>
#include ".....h" //        von javah generierter 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);
}
```

Dies entspricht vollständig einer Implementierung, die auch auf Unix-Systemen zum Einsatz kommen könnte. Daher ist diese Form sicher am ehesten für portierten Code zu empfehlen.

6.3 Dynamisches Laden von native Methoden

Native Methoden für Java müssen dynamisch ladbar sein. Das Vorgehen dabei ist sehr eng an die etablierten Methoden in Unix-Systemen (Shared Libraries) angelehnt. Die Konzepte für Unix-Systeme und die Realisierung für BS2000 werden zunächst gegenübergestellt. Anschließend wird die BS2000-Lösung und die damit verbundenen Anforderungen für den Anwender detailliert dargestellt.

Für Java-Anwendungen auf Unix-Plattformen müssen native Methoden als Shared Libraries produziert werden. Die native Methoden können dann dynamisch nachgeladen und aufgerufen werden. Dazu werden die C-Systemfunktionen *dlopen()* und *dlsym()* verwendet.

Obwohl es in OSD-POSIX inzwischen eine Shared Libraries-Implementierung gibt, wird der aus den Vorversionen bekannte analoge Mechanismus beibehalten. Dabei wird allerdings nicht die volle Funktionalität der Shared Libraries angeboten, sondern nur die im Java-Umfeld benötigte.

6.3.1 Shared Libraries auf Unix-Systemen

Shared Libraries enthalten ein Objekt (also einen Modul, der vom Lader des Systems geladen und ausgeführt werden kann) mit einer besonderen Struktur (ein sogenanntes „Shared Object“). Shared Objects können (unter anderem) während des Programmlaufs dynamisch nachgeladen werden.

Liste notwendiger Objekte

Ein Shared Object kann weitere Objekte angeben, die für seine Ausführung notwendig sind. Diese Objekte werden beim Laden eines Shared Objects mit geladen und bei der Auflösung unbefriedigter Externverweise mit berücksichtigt. Dabei kann jedes dieser Objekte wieder weitere notwendige Objekte angeben, so dass Ketten entstehen können.

Namensräume

Beim Laden eines Shared Objects wird nicht auf andere dynamisch nachgeladene Shared Objects zugegegriffen, es sei denn, sie befinden sich in der Liste der notwendigen Objekte.

Ausnahme ist der Kontext, in den das Programm beim Start geladen wurde (und alle dabei dynamisch geladenen Objekte).

Dadurch entsteht eine Abschottung der Namensräume.

Suchreihenfolge

Die Suche nach Shared Objects zum Ablaufzeitpunkt wird durch die Umgebungsvariable *LD_LIBRARY_PATH* gesteuert, in der verschiedene Verzeichnisse angegeben sein können, die in der angegebenen Reihenfolge nach zu ladenden Shared Objects durchsucht werden.

Auflösung von Extern-Verweisen

Beim Laden eines Shared Objects werden unbefriedigte Externverweise zunächst aus dem primären Lade-Kontext aufgelöst. Anschließend wird das aktuelle Shared Object herangezogen und schließlich die Objekte, die als notwendige Objekte geladen wurden. (Dies ist eine verkürzte Darstellung, Details finden Sie in den Schnittstellenbeschreibungen von *dlopen()* und *dlsym()* der jeweiligen Unix-Handbücher.)

Da die Externverweise innerhalb eines Shared Objects nicht aufgelöst sind, kann eine Funktion, die im Shared Object existiert, durch eine Funktion des primären Lade-Kontexts überladen werden (das ist in LLMs nicht möglich!).

Namenskonvention

Shared Libraries beginnen immer mit dem Präfix *lib* und enden mit dem Suffix *.so*, also z.B. *libhello.so*. Häufig findet man auch einen Versionssuffix am Namen für die Koexistenz und eindeutige Zuordnung verschiedener Schnittstellen-Versionen, etwa z.B. *libXm.so.1.2*.

6.3.2 Shared Libraries im BS2000

Wie schon erwähnt, gibt es im BS2000 keine exakte Entsprechung für die aus Unix-Systemen bekannten Shared Objects. Die für Java wesentlichen Eigenschaften des dynamischen Nachladens, des Abschottens von Namensräumen und des dynamischen Ermittelns von Funktionsadressen werden im Rahmen der Java-Portierung nachgebildet. Nicht nachgebildet werden können dagegen die namensgebende Eigenschaft der mehrfachen Verwendung, das implizite Laden von Shared Objects beim Programmstart sowie Feinheiten des Resolvings. Dies wäre nur mit entsprechenden Erweiterungen des Binde-Lade-Systems möglich.

Da das Binde-Lade-System (BLS) des BS2000 keine Module aus dem POSIX-Dateisystem nachladen kann, müssen native Methoden als LLMs erzeugt und in PLAM-Bibliotheken abgelegt werden.

Da es im LLM keine Möglichkeit gibt, eine „Liste notwendiger Objekte“ anzugeben, diese Funktionalität für Java aber notwendig ist und eine Suchmethodik analog der Unix-Systeme im POSIX-Dateisystem sinnvoll erscheint, wird zusätzlich eine Beschreibungsdatei eingeführt. In dieser Datei ist sozusagen ein Shared Object beschrieben. Sie wird im POSIX-Dateisystem abgelegt, genügt den gleichen Namenskonventionen wie Shared Libraries in Unix-Systemen und enthält alle von Java benötigten Informationen, um die native Methoden nachzuladen und aufzurufen.

Diese Informationen sind vor allem die PLAM-Bibliothek, in der der LLM abgelegt ist, der Namen des Moduls (bzw. der Module) und ggf. die Liste der notwendigen Objekte.

Liste notwendiger Objekte

In der Beschreibungsdatei kann eine Liste notwendiger Objekte eingetragen sein. Diese Objekte werden vor dem aktuellen Objekt nachgeladen. Objekte, die bereits vorhanden sind, werden nicht noch einmal nachgeladen. Objekte werden über ihren POSIX-Dateinamen identifiziert.

Diese Objekte werden beim Laden des aktuellen Objekts mit zur Auflösung von Externverweisen herangezogen.

Unterschiedliche Shared Objects können durchaus gleiche Objekte in ihrer Liste notwendiger Objekte enthalten. Der erste Verweis auf ein solches Objekt führt dann zum Nachladen.

Namensräume (Link-Kontexte)

Jedes Objekt wird in einen eigenen Link-Kontext geladen. Objekte sind daher in ihrem Namensraum abgeschottet.

Das BLS des BS2000 gestattet 200 Link-Kontexte. Werden mehr Objekte nachgeladen, wird die Anwendung abgebrochen.

Suchreihenfolge

Die Suche nach Shared Objects (genauer gesagt nach den Beschreibungsdateien) erfolgt, genau wie in Unix-Systemen, gesteuert durch die Umgebungsvariable *LD_LIBRARY_PATH*.

Auflösung von Extern-Referenzen

Die Kontexte, in die die notwendigen Objekte geladen wurden, werden als Referenz-Kontexte angegeben. Als Referenz-Kontext mit höchster Priorität wird der Default-Kontext verwendet.

Das Durchsuchen des Share-Scopes wird explizit verhindert, weil es derzeit nicht möglich ist, dafür zu sorgen, dass dies erst nach den Referenz-Kontexten geschieht.

Zum Auflösen unbefriedigter Externverweise wird daher zunächst der Default-Kontext durchsucht, anschließend werden die notwendigen Objekte durchsucht. Andere Objekte werden nicht berücksichtigt.

Das Verfahren ist anders als in Unix-Systemen. Insbesondere sind in einem LLM alle inneren Externverweise kurzgeschlossen, so dass keine Funktion in einem LLM überladen werden kann.

6.3.3 Erzeugen von Shared Objects

Die folgenden Abschnitte erläutern das Vorgehen beim Erzeugen eines Shared Objects mit native Methoden, die später dynamisch durch die Java-VM geladen werden können.

Compilieren der Sources

Zum Compilieren der C-Sources von Java-Native-Methoden muss der Compiler C/C++ ab V3.0B für die mit JNI arbeitenden Source-Teile eingesetzt werden.

Bei der Compilierung von C- oder C++-Teilen sind unbedingt die folgenden Compiler-Optionen zu benutzen:

-I <Installations-Pfad>**/include**

Diese Option ist notwendig, damit die Header-Dateien der Java-Distribution gefunden werden. Dabei ist für *<Installations-Pfad>* der Pfad zu substituieren, wo JENV installiert wurde. Für eine Standard-Installation ist dies */opt/java/jdk-9.0.4*. Die aktuell gültige Bezeichnung ist der Freigabemitteilung zu entnehmen.

-K workspace_stack

Dies ist notwendig, damit der Garbage Collector auch die in den C-Teilen verwendeten Java-Objekte finden kann und die Objekte threadfest sein können.

-K c_names_unlimited

Dies ist notwendig, damit das Name-Mangling für Native-Interface-Funktionen korrekt funktioniert.

-K llm_keep

Dies ist notwendig, damit das Name-Mangling für Native-Interface-Funktionen korrekt funktioniert und die Runtimesystem-Funktionen gefunden werden.

-K llm_case_lower

Dies ist notwendig, damit das Name-Mangling für Native-Interface-Funktionen korrekt funktioniert und die Runtimesystem-Funktionen gefunden werden.

-D __SNI_THREAD_SUPPORT

Diese Option muss bei C++-Übersetzungen unbedingt angegeben werden.

Die folgenden Compiler-Optionen können sinnvoll sein:

-K ieee_floats

Wenn das IEEE-Format für Gleitkomma-Zahlen auch im C-Code verwendet werden soll.

-K literal_encoding_ascii
-K literal_encoding_ascii_full

Wenn im C-Code mit ASCII-Strings gearbeitet werden soll.

-K enum_long

Sollte immer gesetzt sein, da die Standard-Einstellung nicht dem ANSI-Standard entspricht.

Außerdem muss unbedingt im ANSI-Modus (*-Xa* oder *-Xc*) übersetzt werden.

Binden eines Großmoduls

Sollte die Implementierung eines Shared Objects aus mehreren Modulen bestehen, so sollten diese zu einem Großmodul zusammengebunden werden. Das geschieht mit dem Kommando *cc* oder *c89*, wobei mindestens folgende Option angegeben sein muss.

-r

Diese Option bewirkt das Binden eines Großmoduls ohne Hinzunahme von Standard-Bibliotheken wie CRTE. Diese dürfen auch keinesfalls explizit mit *-lc* oder *-socket* dazugebunden werden.

-B llm4

Diese Option veranlasst den Binder, einen Großmodul im LLM4-Format zu erzeugen, was für die langen Namen der Java-Native-Methoden notwendig ist.

Erzeugen einer LMS-Bibliothek

Der erzeugte (und im POSIX-Dateisystem vorliegende) Großmodul muss in einer PLAM-Bibliothek als Element mit dem Elementtyp L abgelegt werden. Dies geschieht am besten mit dem POSIX-Kommando *bs2cp*, das die Bibliothek auch neu erzeugt, falls sie noch nicht existiert.

In einer solchen PLAM-Bibliothek können durchaus mehrere Shared Objects abgelegt werden.

ⓘ Der Element-Name des Moduls in der Bibliothek darf maximal 32 Zeichen lang sein.

Erzeugen der Object-Beschreibung

Für die Erzeugung der notwendigen Beschreibungsdatei für ein Shared Object steht das Kommando *mk_shobj* zur Verfügung. Zur Ansicht des Inhalts einer solchen Beschreibungsdatei dient das Kommando *pr_shobj*. Beide Kommandos sind Bestandteil der Java-Distribution und detailliert im Kapitel „Kommandos für BS2000" beschrieben.

C++-Objekte müssen als solche gekennzeichnet werden (siehe Abschnitt „Optionen" im Abschnitt "mk_shobj").

6.3.4 Nutzung von Shared Objects aus Java

Zum Nachladen von native Methoden des Anwenders müssen Sie in Java z.B. die Methode *System.loadLibrary()* aufrufen.

Aus dem bei *loadLibrary()* angegebenen Namen wird ein neuer Name gebildet, unter dem die Library dann gesucht wird, dieser Name ist *lib<name>.so*.

Die Library wird unter Benutzung der Umgebungsvariablen *LD_LIBRARY_PATH* gesucht. Die erste auf diese Weise gefundene Beschreibungsdatei wird dann benutzt, um den oder die zugehörigen Module dynamisch zu laden.

Die JVM und die native Methoden von Java werden in eigenen shared Libraries gehalten und in jeweils separate Kontexte geladen. Werden daher über das JNI-Interface hinausgehende Schnittstellen der JVM genutzt (was nicht sein sollte), müssen die entsprechenden Abhängigkeiten zu den shared Libraries in den Benutzer-Libraries eingetragen werden.

Mit der ersten C++-Methode wird ein C++-Laufzeitsystem (inklusive Tools- und Standard-Library) nachgeladen und C++ wird initialisiert, falls das noch nicht geschehen ist.

Beim Nachladen von Shared Libraries (BIND-Makro) wird der „Share Scope“ zum Auflösen offener Extern-Referenzen derzeit nicht durchsucht, weil dann nicht mehr garantiert werden könnte, dass das Java-private CRTE bzw. Sockets genutzt wird.

Wird vom Anwender selbst via BIND Code nachgeladen, muss dies ebenfalls so geschehen, zumindest wenn Referenzen zum C-Laufzeitsystem und den Sockets bestehen.

Java native Methoden bzw. die sie enthaltenden Großmodule können mit dem derzeitigen Binde-Lade-System nicht vorgeladen werden.

6.4 Invocation-API

Das Invocation-API ist ein Teil des JNI zum Aufruf von Java aus C/C++-Anwendungen. Es werden nur die JNI-Spezifikationen ab Version 1.2 unterstützt.

Änderungen am Invocation API

Das Invocation-API sieht keine Schnittstelle vor, mit der eingestellt werden kann, welche Variante der HotSpot™ VM (Client, Server, o.a.) vom Programm genutzt werden soll.

Normalerweise wird im BS2000 die Client-VM standardmäßig benutzt. Über eine Umgebungsvariable *JENV_VMTYPE*- kann in dieser Implementierung aber auch eine andere VM-Variante (falls verfügbar) ausgewählt werden (siehe Kapitel „Umgebungsvariablen").

6.4.1 Compilieren der C- und C++-Sourcen

Bei der Compilierung der C/C++-Teile sind die im Abschnitt „Implementieren des JavaCodes" im Abschnitt "Implementierung einer native Methode in C" beschriebenen Compiler-Optionen zu benutzen.

Die HotSpot® VM behandelt Überlaufereignisse selbst. Damit es nicht zu Interrupts kommt, ist zusätzlich anzugeben:

-K no_integer_overflow

Diese Option muss für das Hauptprogramm gesetzt werden.

Der C/C++ Compiler ab Version 3.1A20 wurde inkompatibel geändert, so dass bei einem mit diesem Compiler übersetzten Main-Programm die Argument-Strings automatisch als ASCII-Strings übergeben werden, wenn die Option *-K literal_encoding_ascii* oder *-K literal_encoding_ascii_full* gesetzt wurde. Die explizite Konvertierung mit z. B. *_e2a()* kann also entfallen. Falls ein bereits bestehendes Hauptprogramm diese Konvertierung durchführt und nicht verändert werden soll, ist aus Kompatibilitätsgründen folgende Option anzugeben:

-K environment_encoding_ebcdic

Die Argument-Strings werden dann weiterhin als EBCDIC-Strings übergeben.

6.4.2 Binden von Anwendungen in C und C++ mit Java und Green-Threads

Beim Binden der C/C++-Anwendung sind die im Abschnitt „Implementierung einer native Methode in C++" beschriebenen Binder-Optionen zu benutzen.

Mit JENV wird ein Laufzeit-Adapter zur Verfügung gestellt, der zu C/C++ -Anwendungen, die Java über das Invocation-API (Teil des JNI) aufrufen wollen, gebunden werden muss. Dieser Adapter enthält die Funktionen des Invocation-API sowie die Adapter zum threadfesten C- und C++ -Laufzeitsystem und zur threadfesten Socket-Bibliothek.

Der Laufzeit-Adapter liegt in einer optimierten Variante vor. Die Laufzeitsysteme befinden sich in PLAM-Bibliotheken, die Bestandteil des Lieferumfanges von JENV sind:

Für die S390-Variante:

SYSLNK.JENV.090.GREEN-JAVA

Für die X86-Variante:

SKULNK.JENV.090.GREEN-JAVA

Beim Binden einer Anwendung mit diesem Laufzeit-Adapter muss noch beachtet werden, dass er wegen der in Java vorkommenden langen Namen vom LLM-Typ 4 ist. Daher ist unbedingt die Compiler-Option *-B llm4* beim Binden zu verwenden. Darüber hinaus ist zu beachten, dass der C-Compiler normalerweise beim Binden automatisch ein CRTE und im C++ -Fall eine Standard-Library bindet. Dies muss verhindert werden, damit keine Konflikte mit dem bereits im Laufzeit-Adapter enthaltenen threadfesten Laufzeitsystem entstehen. Dies wird mit dem Schalter *-Kno_link_stdlibs* erreicht. Ebenso darf aus gleichem Grund keine Socket-Bibliothek und keine Tools-Bibliothek explizit gebunden werden. Beim Binden dürfen daher niemals die Optionen *-lc, -lsocket* oder *-ltools* verwendet werden.

Unter POSIX kann eine C-Anwendung mit JENV folgendermaßen gebunden werden:

```
export BLSLIB00='$.SYSLNK.JENV.090.GREEN-JAVA'
cc -Kno_link_stdlibs -B llm4 -o <program> \
    <objekte> -l BLSLIB
```

Das gebundene Programm kann ohne weitere Vorkehrungen ausgeführt werden, benötigt aber natürlich zu seinem Ablauf ein vollständig installiertes JENV unter dem Standard-Installationspfad. Soll ein anderswo installiertes Java benutzt werden, so muss die Umgebungsvariable *JAVA_HOME* auf den Installationspfad der Java-Laufzeitumgebung gesetzt werden (siehe Kapitel „Umgebungsvariablen").

Das Kommando *cc* bindet den POSIX-Bindeschalter implizit dazu. Falls nicht unter der Shell mit Kommando *cc* gebunden wird, sondern unter der BS2000-Kommandooberfläche mit dem *BINDER*, so muss dieser Schalter aus *$.SYSLNK.CRTE.POSIX* dazugebunden werden.

Um mit dem BINDER das erforderliche LLM4-Format zu erhalten, muss, wenn auf einer OSD V3 produziert wird, bei *SAVE-LLM* der Operand *FOR-BS2000-VERSIONS=*FROM-OSD-V4* angegeben werden. Die Objekte sind auch auf der OSD V3 ablauffähig.

Diese Vorgehensweise gilt analog für C++ -Anwendungen, wobei zum Binden das Kommando *CC* mit den oben angegebenen Optionen zu benutzen ist.

Eine Anwendung, die explizit die C-Schnittstellen der POSIX-Sockets aufruft, darf nicht die Module der Socket-Bibliothek einbinden, sondern muss den Modul *LIBSOCKET* aus der *SYSLNK.JENV.090.GREEN* (bzw. *SKULNK.JENV.090.GREEN*) mit einbinden.

6.5 Beispiele

An vier Beispielen soll nachfolgend der gesamte Prozess zur Erzeugung einer Java-Applikation mit JNI-Nutzung demonstriert werden.

6.5.1 Implementierung einer native Methode in C

Die Beispiel-Applikation bestehe aus zwei Java-Klassen *Hello* und *Work*, die jeweils eine native Methode enthalten. Die eine gibt eine Begrüßungsmeldung aus, die zweite berechnet etwas. Das Beispiel ist künstlich konstruiert, normalerweise würde dies kein Anwender durch native Methoden erledigen lassen.

Die native Methoden beider Klassen sollen in einer gemeinsamen Bibliothek *example1* abgelegt werden.

Implementieren des Java-Codes

In einer Datei *Hello.java* sei die folgende Java-Klasse definiert:

```
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 der Datei *Work.java* sei die zweite Java-Klasse definiert:

```
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));
   }
}
```

Sollen die native Methoden in unterschiedlichen Libraries abgelegt sein, müssten beide Klassen bei der Initialisierung ihre jeweilige Library laden.

Compilieren des Java-Codes

Die beiden Java-Klassen können jetzt mit dem Kommando

```
javac Hello.java
```

compiliert werden. Die abhängige Klasse *Work* wird bei dieser Übersetzung mit erzeugt.

Erzeugen der Header-Dateien

Die für die Implementierung der native Methoden benötigten Header-Dateien können mit dem Tool *javah* aus den Klassendateien generiert werden:

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

Als Ergebnis stehen die Header-Dateien *Hello.h* und *Work.h* mit den Prototypen der native Funktionen zur Verfügung.

Implementieren des C-Codes

Die Implementierung der native Methoden erfolgt nun typischerweise in entsprechenden Source-Dateien. Für unser Beispiel seien dies die Dateien *Hello.c* und *Work.c*. Beide Dateien inkludieren den mit JENV bereitgestellten Header *jni.h* und jeweils den zugehörigen zuvor generierten Header *Hello.h* bzw. *Work.h*. Die Funktionsdefinition muss zum generierten Prototyp passen. Die weitere Codierung hängt von der gewünschten Implementierung ab.

Das Programm *Hello.c* sei nun folgendermaßen implementiert:

```
#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("Hier meldet sich das Programm %s\n",ebcdic_text);
  free(ebcdic_text);
}
```

Die Datei *Work.c* enthalte folgenden 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 der Datei *Work.c* wurde von der weiter oben beschriebenen Möglichkeit der transparenten Nutzung der IEEE-Funktionen Gebrauch gemacht. In der Datei *Hello.c* werden explizite ASCII-EBCDIC-Konvertierungen durchgeführt.

Um die Beispiele übersichtlich und klein zu halten, wurde auf ausführliche Fehlerbehandlung verzichtet.

Übersetzen der C-Sources

Die im vorigen Abschnitt implementierten C-Sources müssen nun mit den richtigen Compiler-Optionen übersetzt werden. Für *Hello.c* sind das die Standard-Optionen, die weiter oben beschrieben wurden:

```
cc -c -I<Installations-Pfad>/include \
      -Kllm_keep,llm_case_lower \
      -Kworkspace_stack,c_names_unlimited Hello.c
```

Für *Work.c* muss zusätzlich noch die IEEE-Arithmetik berücksichtigt werden:

```
cc -c -I<Installations-Pfad>/include \
      -Kllm_keep,llm_case_lower \
      -Kworkspace_stack,c_names_unlimited \
      -Kieee_floats Work.c
```

Als Ergebnis stehen die Objekt-Dateien zur Verfügung.

Erzeugen des Shared Objects

Mit dem folgenden Kommando können die zuvor erzeugten Objekte zu einem Großmodul gebunden werden:

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

Anschließend wird der erzeugte Großmodul in einer BS2000-Bibliothek abgelegt:

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

Zuletzt wird eine Beschreibungsdatei entsprechend den Namenskonventionen erzeugt, die die richtigen Verweise enthalten muss:

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

Ablauf des Programms

Für den Ablauf des Programms ist es jetzt nur noch notwendig, die Umgebungsvariable *LD_LIBRARY_PATH* so einzustellen, dass das erzeugte Shared Object auch gefunden wird. In unserem Beispiel ist dies mit

```
export LD_LIBRARY_PATH=.
```

möglich. Die Anwendung kann nun mit

```
java Hello
```

zum Ablauf gebracht werden.

6.5.2 Implementierung einer native Methode in C++

Die Implementierung in C++ ist weitestgehend mit dem Vorgehen bei der Implementierung in C identisch. Bei dem obigen Beispiel ergeben sich folgende Unterschiede:

Das Programm *Hello.cpp* wird nun folgendermaßen implementiert:

```
#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 <<  "Hier meldet sich das Programm " << ebcdic_text << endl;
  free(ebcdic_text);
}
```

Beim Compilieren ist statt des Kommandos *cc* das Kommando *CC* zu verwenden. Beim Erzeugen des Shared Objects muss das Flag *cpp* gesetzt werden:

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

6.5.3 Nutzung von Java aus einer C-Anwendung

Das folgende Beispiel demonstriert die Verwendung des Java-Invocation-APIs (Teil des JNI) für den Aufruf von Java-Programmen aus C heraus. Das gewählte Beispiel ist absichtlich einfach gehalten.

Ein Java-Echo-Programm gebe alle seine Argumente auf die Standard-Ausgabe aus. Aus einem C-Programm wird dieses Java-Programm dann aufgerufen.

Implementierung des Java-Codes

In der Datei *Echo.java* sei die folgende Klasse definiert:

```
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("");
  }
}
```

Compilieren des Java-Codes

Die oben definierte Java-Klasse kann jetzt einfach mit dem Kommando

```
javac Echo.java
```

compiliert werden. Durch Aufruf von

```
java Echo Das ist ein Versuch
```

können Sie sich vom Funktionieren des Programms überzeugen.

Implementieren des C-Codes

Das folgende C-Programm soll obiges Java-Programm aufrufen und ihm seine Aufrufargumente dabei übergeben. Besonders zu beachten ist wiederum, dass alle an en übergebe-nen Strings ASCII-codiert sein müssen. Dieses Beispiel wird daher vollständig im ASCII-Modus implementiert und produziert.

Die Datei *Echo.c* sei nun folgendermaßen implementiert:

```
#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;
}
```

Das Programm arbeitet in dieser Form nur mit einer Standard-Installation von JENV zusammen. Für den Ablauf mit einer Privat-Installation müssen Sie die Umgebungsvariable *JAVA_HOME* entsprechend setzen (siehe Kapitel „Umgebungsvariablen").

Übersetzen der C-Source

Die im vorigen Abschnitt implementierte C-Source muss nun mit den richtigen CompilerOptionen übersetzt werden. Für *Echo.c* muss zusätzlich zu den Standard-Optionen, die weiter oben beschrieben wurden, noch der ASCII-Modus berücksichtigt werden:

```
cc -c -I<Installations-Pfad>/include \
      -Kllm_keep,llm_case_lower \
      -Kworkspace_stack,c_names_unlimited \
      -Kliteral_encoding_ascii \
      -Kno_integer_overflow Echo.c
```

Als Ergebnis steht eine Objekt-Datei zur Verfügung.

Binden und Ausführen der Anwendung

Beim Binden der Anwendung muss berücksichtigt werden, dass der Laufzeit-Adapter von Java mit eingebunden wird und nicht die „normalen" Laufzeitsysteme.

Mit den folgenden Kommandos kann die Anwendung gebunden werden:

```
export BLSLIB00='$.SYSLNK.JENV.090.GREEN-JAVA'
cc -Kno_link_stdlibs -B llm4 -o Echo \
      Echo.o -l BLSLIB
```

Das Programm kann dann wie jedes andere POSIX-Programm aufgerufen werden. Es benötigt zum Ablauf aber ein unter dem Standard-Installationspfad installiertes JENV. Für die Verwendung eines anderswo installierten JENV muss die Umgebungsvariable *JAVA_HOME* entsprechend gesetzt werden (siehe Kapitel „Umgebungsvariablen").

Der Aufruf mittels

```
Echo Das ist ein Java-Echo
```

führt zu der erwarteten Ausgabe:

```
Das ist ein Java-Echo
```

Das Programm wird mit der in Abschnitt „Option zur Auswahl des HotSpot ™ VM-Typs" im Abschnitt "java" beschriebenen Default-VM ausgeführt. Durch vorheriges Setzen der Umgebungsvariable *JENV_VMTYPE* kann der VM-Typ explizit bestimmt werden. Also z.B.:

```
export JENV_VMTYPE=client
```

Dies bewirkt, dass die HotSpot™ Client-VM zur Ausführung verwendet wird.

6.5.4 Nutzung von Java aus einer C++-Anwendung

Gegenüber der Nutzung von Java aus einer C-Anwendung (siehe Abschnitt „Nutzung von Java aus einer C-Anwendung") ergeben sich die folgenden Unterschiede:

Implementieren des C++-Codes

Die Datei *Echo.cpp* sei nun folgendermaßen implementiert:

```
#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;
}
```

Übersetzen der C++-Source

Die im vorigen Abschnitt implementierte C++-Source muss nun mit dem Kommando CC und den richtigen Compiler-Optionen übersetzt werden. Für *Echo.cpp* muss zusätzlich zu den Standard-Optionen, die weiter oben beschrieben wurden, noch der ASCII-Modus berücksichtigt werden. Außerdem wird in diesem Beispiel eine Anwendung produziert, die mit der X86-Variante von JENV auf SQ-Anlagen ablauffähig ist:

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

Binden und Ausführen der Anwendung

Beim Binden der Anwendung muss berücksichtigt werden, dass der X86-Laufzeit-Adapter von Java mit eingebunden wird und nicht die „normalen“ Laufzeitsysteme.

Mit den folgenden Kommandos kann die Anwendung gebunden werden:

```
export BLSLIB00='$.SKULNK.JENV.090.GREEN-JAVA'

CC -Kno_link_stdlibs -B llm4 -o Echo \
      Echo.o -l BLSLIB
```

Das Programm kann dann wie jedes andere POSIX-Programm auf einer SQ-Anlage aufgerufen werden. Es benötigt zum Ablauf aber eine unter dem Standard-Installationspfad installierte X86-Variante von JENV. Für die Verwendung eines anderswo in- stallierten JENV muss die Umgebungsvariable JAVA_HOME entsprechend gesetzt werden (siehe Kapitel „Umgebungsvariablen").

Der Aufruf mittels

```
Echo Das ist ein Java-Echo
```

führt zu der erwarteten Ausgabe:

```
Das ist ein Java-Echo
```

Das Programm wird mit der in Abschnitt „Option zur Auswahl des HotSpot ™ VM-Typs" im Abschnitt "java" beschriebenen Default-VM ausgeführt. Durch vorheriges Setzen der Umgebungsvariable *JENV_VMTYPE* kann der VM-Typ explizit bestimmt werden. Also z.B.:

```
export JENV_VMTYPE=client
Echo Das ist ein Java-Echo
```

Dies bewirkt, dass die HotSpot® Client-VM zur Ausführung verwendet wird.

7 JCI - Invocation-API für COBOL

Das Java-COBOL-Interface (JCI) ist eine Sammlung von Funktionen und COBOL-COPY-Elementen für die einfachere Bedienung der Schnittstellen des *Java Invocation API* aus COBOL-Programmen.

Das *Java Invocation API* ist ein Teil des *Java Native Interface* (JNI). Da es für die Sprache C/C++ ausgelegt ist, sind seine Schnittstellen direkt von COBOL-Programmen aus nur umständlich zu bedienen.

Das JCI unterstützt folgende Funktionen:

- Java-VM starten
- Klassen laden
- Methoden aufrufen
- Java-Objekte erzeugen und bearbeiten
- prüfen, ob eine Exception geworfen wurde
- die Java-VM beenden.

Nicht unterstützt wird die Möglichkeit, native COBOL-Methoden zu erstellen und aufzurufen.

7.1 Übersetzen der COBOL-Sourcen

Zum Übersetzen einer COBOL-Source, die JCI-Schnittstellen verwendet, wird ein COBOL2000-Compiler ab Version V1.4A benötigt.

7.1.1 Zuweisung der JCI-COPY-Bibliothek

Die JCI-COPY-Elemente befinden sich im POSIX-Verzeichnis *<Installations-Pfad>* `/include`. Dabei ist für *<Installations-Pfad>* der Pfad einzusetzen, unter dem JENV installiert wurde. Für eine Standard-Installation ist dies */opt/java/jdk-9.0.4*. Die aktuell gültige Bezeichnung ist der Freigabemitteilung zu entnehmen.

Dieser Pfad muss dem Compiler unter der BS2000-Kommandooberfläche mittels der S-Variablen `SYSIOL-<libname>` bzw. `SYSIOL-COBLIB` bekannt gemacht werden:

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

Genaueres siehe „COBOL2000 (BS2000) Benutzerhandbuch“ [5].

Unter POSIX muss die Umgebungsvariable `<libname>` bzw. `COBLIB` gesetzt werden:

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

Genaueres siehe „COBOL2000 (BS2000) Benutzerhandbuch“ [5].

7.1.2 Erforderliche Optionen/Direktiven

Da an der Schnittstelle zu JCI-Funktionen Datenstrukturen verwendet werden, die Zeiger enthalten, ist beim Übersetzen des COBOL-Programms folgende Option notwendig:

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

Unter POSIX entspricht dies der Option:

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

Alle JCI-Funktionen liefern einen ganzzahligen Rückgabewert nach ILCS-Konventionen (d.h. im Mehrzweckregister R1). Damit dieser im COBOL-Programm verwendet werden kann, muss er nach der Rückkehr im COBOL-Sonderregister RETURN-CODE zur Verfügung gestellt werden. Dazu haben Sie folgende Möglichkeiten

- Angabe der Option
  `SOURCE-PROPERTIES=*PAR(RETURN-CODE=*FROM-ALL-SUBPROGRAMS,...)`
- bzw. unter POSIX
  `-C ACTIVATE-XPG4-RETURNCODE=YES`
- oder gezielt im Quellprogramm durch Angabe der Direktive
  `>>CALL-CONVENTION ILCS-SET-RETURN-CODE`

Dabei gelten die Optionen für das gesamte Quellprogramm, die Direktive nur, bis eine `>>CALL-CONVENTION` Direktive mit einem anderen Wert angegeben wird (siehe „COBOL2000 (BS2000) Sprachbeschreibung“ [6]).

Das erzeugte Modul muss im LLM-Format vorliegen. Dazu ist bei Übersetzung unter der BS2000-Kommandooberfläche folgende Option erforderlich:

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

Bei Übersetzung unter POSIX existiert keine entsprechende Option, da dort immer ein LLM generiert wird.

7.2 Binden von COBOL-Anwendungen mit Java

Die JCI-Funktionen werden in zwei PLAM-Bibliotheken bereitgestellt:

SYSLNK.JENV.090.GREEN-JAVA (für die *S390*-Variante),

SKULNK.JENV.090.GREEN-JAVA (für die *X86*-Variante)

Zusätzlich enthalten diese Bibliotheken die vom JCI gerufenen JNI-Funktionen, das threadfeste C/C++-Laufzeitsystem und das komplette COBOL-Laufzeitsystem, letzteres immer im *S390*-Format.

Externverweise aus Anwendungen, die JCI-Funktionen aufrufen, müssen beim Binden vorrangig aus einer dieser Bibliotheken befriedigt werden.

Unter POSIX ist dazu die Zuweisung der Umgebungsvariablen `BLSLIB00` erforderlich:

```
export BLSLIB00='$.SYSLNK.JENV.090.GREEN-JAVA'
cobol -g -M <PROG-ID> -o <program> <objekte> -l BLSLIB
```

Das `cobol`-Kommando bindet den POSIX-Bindeschalter implizit dazu. Falls nicht unter der Shell mit dem `cobol`-Kommando gebunden wird, sondern unter der BS2000-Kommandooberfläche mit dem BINDER, so muss dieser Schalter aus *$.SYSLNK.CRTE.POSIX* dazu gebunden werden.

7.3 Ablauf von COBOL-Anwendungen mit Java

Bevor eine Anwendung, die JCI-Funktionen aufruft, von der BS2000-Kommandooberfläche gestartet wird, muss die POSIX-Umgebung mit der Prozedur *//INITIALIZE* für den Ablauf initialisiert werden (siehe "Prozedur INITIALIZE").

Das COBOL-Laufzeitsystem verhält sich dann im Wesentlichen so, als wäre es unter der POSIX-Shell gestartet worden (siehe „COBOL2000 (BS2000) Benutzerhandbuch“ [5] und Abschnitt „Besonderheiten").

Nach Beendigung der Anwendung muss die POSIX-Umgebung durch den Aufruf der Prozedur *DELETE* unbedingt wieder zurückgesetzt werden. Andernfalls ist die Umgebung für weitere Übersetzungen falsch eingestellt.

7.4 Zeichen und Zeichenfolgen

Alphanumerische und nationale Zeichenfolgen werden an JCI-Funktionen in Strukturen übergeben, die zusätzlich zum Datenbereich ein Längenfeld enthalten.

Beispiel:

```
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 diesem Kapitel werden solche Strukturen in den Formaten als `Cobvar` bzw. `CobNvar` bezeichnet.

Ob Leerzeichen am Ende des Textbereiches ignoriert oder beibehalten werden, hängt von der aufgerufenen Funktion ab. In einigen Funktionen kann dieses Verhalten über einen zusätzlichen Parameter gesteuert werden.

Alphanumerische Zeichen und COBOL-Zeichenfolgen besitzen eine EBCDIC-Darstellung, während die Java-VM eine UTF-Darstellung (je nach Schnittstelle UTF8 oder UTF16) erwartet bzw. liefert. Notwendige Konvertierungen werden in den JCI-Funktionen automatisch durchgeführt. Dazu müssen alle Zeichen in EDF03IRV darstellbar sein. Für Zeichen(folgen), für die keine solche Darstellung existiert, müssen nationale Zeichen(folgen) (UTF16-Darstellung) verwendet werden, ansonsten ist das Ergebnis undefiniert. Für Zeichenfolgen, die binäre Nullen enthalten, müssen ebenfalls nationale Zeichenfolgen verwendet werden. Für Schnittstellen, für die keine nationalen Zeichenfolgen definiert sind (z.B. Klassen- und Methodennamen), dürfen nur konvertierbare Zeichen verwendet werden.

Java-Strings liegen als Objekte vor. Die Konvertierung zwischen Java-Strings und COBOL-Zeichenfolgen geschieht automatisch in den JCI-Funktionen.

Die Konvertierung besteht aus 2 Teilschritten:

- Konvertierung zwischen EBCDIC-Strings und UTF8-Strings (nur für alphanumerische Zeichenfolgen).
- Konvertierung zwischen UTF8- bzw. UTF16-Strings und Objekten.

Kommt es bei einer dieser Konvertierungen zu einem Fehler (z.B. Speichermangel), so wird die Bedingungsvariable `ResErrCode` (COPY-Element *JCI-METHODRES*) auf den Wert `RES-ERR-NOMEM` (Fehler im ersten Schritt) bzw. `RES-ERR-OBJECT` (Fehler im zweiten Schritt) gesetzt.

Ist das Längenfeld der COBOL-Struktur vor der Konvertierung gleich 0, so bleibt bei der Konvertierung eines Java-Strings in einen COBOL-String der Textbereich unverändert, bei Konvertierung in die andere Richtung wird ein Objekt für einen Null-String erzeugt. Ist das Längenfeld vor der Konvertierung kleiner als 0, so wird die Bedingungsvariable `ResErrCode` auf `RES-ERR-LENGTH` gesetzt.

7.5 Gleitkommazahlen

Die Java Gleitkommatypen *float* und *double* sind im IEEE-Format dargestellt, während die COBOL Gleitkommatypen *COMP-1* und *COMP-2* im /390-Format dargestellt werden. Die Konvertierung wird in den JCI-Funktionen automatisch durchgeführt.
Bei der Konvertierung kann es zu folgenden Ausnahmesituationen kommen, die dem Aufrufer bei Rückkehr aus der JCI-Funktion als Bedingungsvariable im Feld `ResErrCode` (COPY-Element *JCI-METHODRES*) angezeigt werden:

- COMP-1 ---> IEEE:

  `RES-ERR-FLOAT-UNDERFLOW`

  Die /390-Gleitpunktzahl ist betragsmäßig kleiner als die kleinste darstellbare IEEE-Gleitpunktzahl.

  `RES-ERR-FLOAT-OVERFLOW`

  Die /390-Gleitpunktzahl ist betragsmäßig größer als die größte darstellbare IEEE-Gleitpunktzahl.

- COMP-2 ---> IEEE:

  (keine)

- IEEE ---> COMP-1:

  `RES-ERR-FLOAT-INVALID`

  Die IEEE-Gleitpunktzahl entspricht *NaN* oder *Infinity*.

- IEEE ---> COMP-2:

  `RES-ERR-FLOAT-UNDERFLOW`

  Die IEEE-Gleitpunktzahl ist betragsmäßig kleiner als die kleinste darstellbare /390-Gleitpunktzahl.

  `RES-ERR-FLOAT-OVERFLOW`

  Die IEEE-Gleitpunktzahl ist betragsmäßig größer als die größte darstellbare /390-Gleitpunktzahl.

  `RES-ERR-FLOAT-INVALID`

  Die IEEE-Gleitpunktzahl entspricht *NaN* oder *Infinity*.

Gehen bei der Konvertierung Bitstellen verloren, so führt dies nicht zu einer Ausnahmesituation.

7.6 Objektreferenzen

Java-Objekte werden als lokale Objektreferenzen an das COBOL-Programm übergeben.

Um zu verhindern, dass der *Garbage Collector* die referenzierten Objekte entfernt, registriert die VM alle übergebenen Referenzen.

Die Referenzen sind gültig, bis eine native Methode zu Java zurückkehrt. Dies ist bei einem COBOL-Hauptprogramm aber nie der Fall. Um die für die Registrierung benötigten Ressourcen freizugeben und es dem *Garbage Collector* zu ermöglichen, die durch die Objektreferenzen referenzierten Objekte zu entfernen, müssen die Referenzen daher explizit durch das COBOL-Programm freigegeben werden (siehe Abschnitt „Objektreferenzen").

Für Objektreferenzen wird im COPY-Element *JCI-TYPEDEF* der TYPEDEF `JCI-object` definiert.

7.7 Java-Handle

Manche JCI-Funktionen verwenden Parameter mit opakem Datentyp. Diese werden in den Formaten als Java-Handle bezeichnet.

Für Java-Handle wird im COPY-Element *JCI-TYPEDEF* der TYPEDEF `JCI-handle` definiert.

7.8 Returncode im Sonderregister RETURN-CODE

Alle JCI-Funktionen sind `int` Funktionen, die entweder einen Wahrheitswert oder einen Fehlerindikator im Sonderregister RETURN-CODE zurückliefern.

Für die Rückgabe anderer Werte wird ein separater Parameter verwendet. Falls nicht anders beschrieben, ist der Inhalt des durch diesen Parameter referenzierten Ergebnisfeldes im Fehlerfall undefiniert.

7.9 Argumente und Ergebniswerte von Java-Methoden

Für die Übergabe von Argumenten und Ergebniswerten zwischen dem COBOL-Programm und JCI-Funktionen, die Java-Methoden aufrufen oder Java-Datenfelder bearbeiten, werden Strukturen verwendet. Diese müssen alle notwendigen Informationen enthalten. Die Strukturen sind in den COPY-Elementen *JCI-METHODARGS* und *JCI-METHODRES* definiert (siehe Abschnitte „JCI-METHODARGS - Funktionsargumente" und „JCI-METHODRES - Funktionsergebnis"). Sie werden in diesem Kapitel als `MethodArg` bzw. `MethodRes` bezeichnet.

Sofern bei den Funktionsbeschreibungen nichts anderes definiert wird, gelten für Aufrufe von Funktionen, die ein Argument des Typs `MethodArg` bzw. `MethodRes` referenzieren, folgende Voraussetzungen:

- Argumente

  Das Feld `CallArgNum` muss vor dem Aufruf der Funktion die Anzahl der zu übergebenen Argumente enthalten.

  Für jedes Argument muss in der Struktur ein Element der Tabelle `CallArg` mit Werten versorgt sein.

  Im Feld `ArgType` muss der dem COBOL-Datentyp entsprechende Bedingungsname `ARG-...` gesetzt werden.

  Für Zeichenfolgen muss die Adresse einer `Cobvar` bzw. `CobNvar` Struktur angegeben werden. Falls nachfolgende Leerzeichen ignoriert werden sollen, muss zusätzlich `IGNORE-TRAILING-SPACES` gesetzt werden. Andere Argumente müssen direkt in die Struktur übertragen werden.
- Ergebniswerte

  Für Ergebniswerte muss der dem COBOL-Datentyp entsprechende Bedingungsname `RES-...` im Feld `ResType` gesetzt werden oder `RES-VOID`, falls kein Rückgabewert erwartet wird. Wird als Rückgabewert eine Zeichenfolge erwartet, so muss die Adresse einer `Cobvar` bzw. `CobNvar` Struktur mit maximaler Länge für den Datenbereich im Feld `ResValAddr` angegeben werden. Ist die Länge <=0, so findet keine Übertragung des Ergebniswertes statt.

  Nach Rückkehr aus der Funktion enthält eine als Rückgabewert referenzierte `Cobvar` bzw. `CobNvar` Struktur im Längenfeld die Anzahl der übertragenen Zeichen (maximal Eingabewert) und im Datenbereich die übertragenen Zeichen. Für andere Datentypen wird der Rückgabewert direkt in der Struktur übergeben.

Falls bei der Konvertierung eines Gleitpunkt-Datenfeldes oder String-Objektes eine Ausnahmesituation auftrat, enthält das Feld `ResErrCode` nach der Rückkehr aus der Funktion den entsprechenden Fehlercode. Dieser kann durch die Bedingungsvariable `RES-ERR-<bedingung>` abgefragt werden (siehe Abschnitte „Zeichen und Zeichenfolgen" und „Gleitkommazahlen"). War ein Argument fehlerhaft (RETURN-CODE RET-ERR-EARGUMENT), so enthält das Feld `ResErrIndex` die Nummer des Ar gumentes.

Die folgende Tabelle gibt eine Übersicht über die Definitionen und die entsprechenden COBOL- und Java-Typen. Für COBOL-Typen, deren Name mit '`JCI-`' beginnt, existiert eine Typ-Definition im COPY-Element *JCI-TYPEDEF.* Ein '*' in der letzten Spalte gibt an, dass in den JCI-Funktionen eine automatische Konvertierung des Argumentes bzw. Ergebniswertes vorgenommen wird.

Java-Typ	**COBOL-Typ bzw. TYPEDEF**	**Variable** `ResVal` **...,** `ArgVal` **...**	**Bedingungsname** `RES-` **...,** `ARG-` **...**	
String	Struktur `CobVar` Struktur `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-Objekt (auch String-Objekt)	JCI-object	Object	OBJECT	

7.10 Exceptions

Exceptions können sowohl implizit durch die JCI-Funktionen als auch explizit durch eine Java-Methode ausgelöst werden. Dies ist im Allgemeinen nicht am Rückgabewert der Funktion erkennbar.

Es stehen JCI-Funktionen zur Verfügung, um die Existenz einer Exception abzufragen, sich Informationen ausgeben zu lassen und die Exception zu entfernen (siehe Abschnitt „Exceptions").

Wenn eine Exception ausgelöst wurde, muss sie zunächst durch den Aufruf von `JCI_ExceptionClear` entfernt werden, bevor der fehlerfreie Ablauf weiterer JCI-Funktionen gewährleistet ist.

7.11 COPY-Elemente

Für allgemeine Definitionen und Strukturen werden COPY-Elemente bereitgestellt.

Sie befinden sich im POSIX-Verzeichnis `include` unterhalb von dem Pfad, unter dem JENV installiert wurde.

7.11.1 JCI-CONST - Definition von Konstanten

Dieses Element definiert die COBOL-Teilstruktur `JCI-Const`, die alle Werte von JCI-relevanten Konstanten als Datenfelder enthält:

```
*> 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'.
```

7.11.2 JCI-TYPEDEFS - Typdefinitionen

Dieses Element enthält alle elementaren JCI-relevanten Typdefinitionen:

```
*> 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).
```

7.11.3 JCI-VMOPT - Struktur zur Übergabe von Optionen

Das Element enthält die zur Übergabe von Optionen beim Start der VM benötigte Teilstruktur `JCI-VMopt`:

```
*> 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.
```

Im Folgenden wird diese Struktur als `OptArg` bezeichnet.

Die Anzahl der Elemente, mit der die Optionentabelle expandiert werden soll (maximale Anzahl von Argumenten), muss durch die REPLACING-Angabe der COPY-Anweisung gesetzt werden:

`COPY JCI-VMOPT REPLACING == <max-options> == BY` *num* `.`

Zur dynamischen Initalisierung der Struktur als Ganzes ist folgende Anweisung erforderlich, um die korrekten Werte für intern reservierte Felder sicherzustellen:

```
INITIALIZE JCI-VMopt WITH FILLER ALL TO VALUE
```

7.11.4 JCI-METHODARGS - Funktionsargumente

Das Element enthält die für die Übergabe von Argumenten notwendige Teilstruktur `JCI-MethodArgs:`

```
*> 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).
```

Die Anzahl der Elemente, mit der die Argumenttabelle expandiert werden soll (maximale Anzahl von Argumenten), muss durch die REPLACING-Angabe der COPY-Anweisung gesetzt werden:

`COPY JCI-METHODARGS REPLACING == <max-arguments> == BY` *num* `.`

Zur dynamischen Initalisierung der Struktur als Ganzes ist folgende Anweisung erforderlich, um die korrekten Werte sowohl für intern reservierte Felder als auch für die Tabellenelemente sicherzustellen:

```
INITIALIZE JCI-MethodArgs WITH FILLER ALL TO VALUE
    THEN REPLACING ALPHANUMERIC BY ALL X'00'
    THEN TO DEFAULT
```

7.11.5 JCI-METHODRES - Funktionsergebnis

Das Element enthält die für die Übergabe von Ergebniswerten und Fehlerinformationen notwendige Teilstruktur `JCI-MethodRes`:

```
*> 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).
```

Zur dynamischen Initalisierung der Struktur als Ganzes ist folgende Anweisung erforderlich, um die korrekten Werte für intern reservierte Felder sicherzustellen:

```
INITIALIZE JCI-MethodRes WITH FILLER ALL TO VALUE
```

7.12 Funktionen

In diesem Abschnitt sind die Schnittstellen der JCI-Funktionen nach inhaltlichen Gesichtspunkten zusammengefasst.

In den Formaten werden Objektreferenzen der Einfachheit halber zumeist als Objekte bezeichnet.

Klassen-Objekt bezeichnet eine Referenz auf ein Objekt der Klasse *java.lang.Class*.

7.12.1 Starten und Beenden der Java-VM

Dieser Abschnitt beschreibt die JCI-Funktionen, die zum Starten und Beenden der Java-VM benötigt werden.

7.12.1.1 JCI_CreateJavaVM

Die Funktion erzeugt, d.h. lädt und initialisiert, die Java-VM.
Sie ist äquivalent zur JNI-Funktion `JNI_CreateJavaVM`.

Aufruf

`CALL 'JCI_CreateJavaVM' USING` *opt*

opt Optionen für die Java-VM

Argumente

opt eine Struktur der Form `OptArg` mit folgenden Elementen:

`VMOptNum`

Anzahl der VM-Optionen; der Wert darf nicht größer als der für `<max-options>` angegebene Wert sein (siehe Abschnitt „JCI-VMOPT - Struktur zur Übergabe von Optionen").

`VMOptFlag`

Zeigt an, ob unbekannte Optionen ignoriert werden sollen (Bedingungsname `IGNORE-UNRECOGNIZED`).

`VMOptVstring`

Für jede Option die Adresse einer `Cobvar`-Struktur.Nachfolgende Leerzeichen am Ende des Textes werden ignoriert

`VMExtrainf`

Optionsabhängig die Adresse einer externen Funktion.Es können alle Optionen angegeben werden, die auch bei der JNI-Funktion `JNI_CreateJavaVM` zulässig sind.

Returnwert (RETURN-CODE)

`JCI-RET-OK`

Der Aufruf war erfolgreich.

`JCI-RET-EVERSION`

Die statisch generierte Versionsnummer in *opt* ist ungültig (eventuell überschrieben).

`JCI-RET-EOPTNUM`

Die Anzahl der übergebenen Optionen (`VMOptNum`) ist kleiner als 0 oder größer als der Wert, der für `<max-options>` angegebene wurde (siehe Abschnitt „JCI-VMOPT - Struktur zur Übergabe von Optionen").

`JCI-RET-EEXIST`

Es wurde bereits eine Java-VM erzeugt.

`JCI-RET-ENOMEM`

Für das Erzeugen der Java-VM steht nicht genug Speicherplatz zur Verfügung.

`JCI-RET-ERR`

Es ist ein nicht näher spezifizierter Fehler aufgetreten (z.B. ungültige Option und `IGNORE-UNRECOGNIZED` nicht gesetzt).

Hinweise

Es kann nur eine *JavaVM* in einem Programmlauf erzeugt werden.

Auch nach Beendigung der VM mit `JCI_DestroyJavaVM` kann keine neue Java-VM erzeugt werden.

Beispiel

```
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
...
```

7.12.1.2 JCI_DestroyJavaVM

Die Funktion gibt Ressourcen der Java-VM frei.
Sie ist äquivalent zur JNI-Funktion `JNI_DestroyJavaVM`.

Aufruf

```
CALL 'JCI_DestroyJavaVM'
```

Returnwert (RETURN-CODE)

`JCI-RET-OK`

Der Aufruf war erfolgreich.

`JCI-ERR`

Es ist ein nicht näher spezifizierter Fehler aufgetreten.

Hinweise

- Die Funktion sollte auch aufgerufen werden, wenn der Aufruf der Funktion `JCI_CreateJavaVM` nicht erfolgreich war.
- Nach erfolgreicher Ausführung der Funktion können keine JCI-Funktionen mehr aufgerufen werden.
- Die Java-VM wird nicht entladen.
- Ein Neustart der Java-VM mit `JCI_CreateJavaVM` ist nicht möglich.

7.12.2 Klassen und Methoden

Dieser Abschnitt beschreibt die JCI-Funktionen, die benötigt werden, um Klassen zu laden und Methoden aufzurufen.

7.12.2.1 JCI_FindClass

Die Funktion lokalisiert und lädt eine Klasse.
Sie ist äquivalent zur JNI-Funktion `FindClass`.

Aufruf

`CALL 'JCI_FindClass' USING` *cName cObj*

cName Name der Klasse

cObj von der Funktion geliefertes Klassen-Objekt

Argumente

cName Struktur vom Typ `Cobvar`
Vollqualifizierter Name der Klasse (d.h. ein Package-Name getrennt durch "/" gefolgt vom Namen der Klasse), die gesucht werden soll. Beginnt der Name mit "[" (Array Signatur Zeichen), wird eine Array-Klasse geliefert.
Nachfolgende Leerzeichen am Ende des Textes werden ignoriert.

cObj Datenfeld des Typs `JCI-object`
Nach erfolgreicher Ausführung der Funktion enthält das Feld ein Klassenobjekt der gesuchten Klasse.
Im Fehlerfall wird der Wert JCI-NULL zurückgeliefert.

Returnwert (RETURN-CODE)

`JCI-RET-OK`

Der Aufruf war erfolgreich.

`JCI-RET-ENOVM`

Es ist keine Java-VM gestartet.

`JCI-RET-ENOTFOUND`

Die Klasse konnte nicht geladen werden.

Exceptions

Die von der Funktion geworfenen Exceptions entsprechen denen der JNI-Funktion `FindClass`.

Beispiel

```
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
...
```

7.12.2.2 JCI_GetStaticMethodID

Die Funktion liefert die Methoden-ID (Java-Handle) für eine statische Methode einer Klasse.
Sie ist äquivalent zur JNI-Funktion `GetStaticMethodID`.

Aufruf

`CALL 'JCI_GetStaticMethodID' USING` *cObj mName mSig mID*

cObj	Klassenobjekt
mName	Name der Methode
mSig	Signatur der Methode
mID	von der Funktion gelieferte Methoden-ID

Argumente

cObj	Datenfeld des Typs `JCI-object` Objekt der Klasse, in der die Methode gesucht werden soll.
mName	Struktur vom Typ `Cobvar` Name der Methode, die gesucht werden soll. Nachfolgende Leerzeichen am Ende des Textes werden ignoriert.
mSig	Struktur vom Typ `Cobvar` Signatur der Methode, die gesucht werden soll. Nachfolgende Leerzeichen am Ende des Textes werden ignoriert.
mID	Datenfeld des Typs `JCI-handle` Nach erfolgreicher Ausführung der Funktion enthält das Feld die Methoden-ID der gesuchten Methode. Im Fehlerfall wird der Wert JCI-NULL zurückgeliefert.

Returnwert (RETURN-CODE)

`JCI-RET-OK`

Der Aufruf war erfolgreich.

`JCI-RET-ENOVM`

Es ist keine Java-VM gestartet.

`JCI-RET-ENULLOBJ`

cObj ist `JCI-NULL`.

`JCI-RET-EARGUMENT`

cObj ist kein Klassen-Objekt.

`JCI-RET-ENOTFOUND`

Die Methode konnte nicht gefunden werden.

Exceptions

Die von der Funktion geworfenen Exceptions entsprechen denen der JNI-Funktion `GetStaticMethodID`.

Hinweise

Die Methode wird durch den Namen und die Signatur identifiziert. Die Signatur kann durch die Anweisung

```
javap -s <Klassenname>
```

erhalten werden, wobei <`Klassenname`> der Name der durch *cObj* identifizierten Klasse ist.

Beispiel

```
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
...
```

7.12.2.3 JCI_CallStaticMethod

Die Funktion ruft eine statische Methode auf.
Sie ist äquivalent zu den JNI-Funktionen `CallStatic<type>Method`. Sie bietet jedoch zusätzlich die Möglichkeit, Zeichenketten direkt zu übergeben bzw. zu erhalten.

Aufruf

`CALL 'JCI_CallStaticMethod' USING` *cObj mID arg res*

cObj Klassenobjekt

mID Methoden-ID

arg Methodenargumente

res Methodenergebnis

Argumente

cObj Datenfeld des Typs `JCI-object`
Klassen-Objekt, dessen Methode gerufen werden soll.

mID Datenfeld des Typs `JCI-handle`
ID der Methode, die gerufen werden soll. Die Methoden-ID muss durch den Aufruf der Funktion `JCI_GetStaticMethodID` für die Klasse *cObj* beschafft werden.

arg eine Struktur der Form `MethodArg` Beschreibung der Argumente für den Methodenaufruf (siehe Abschnitt „Argumente und Ergebniswerte von Java-Methoden").

res eine Struktur der Form `MethodRes`
Beschreibung von Rückgabewert für den Methodenaufruf und Fehlerinformation (siehe Abschnitt „Argumente und Ergebniswerte von Java-Methoden"). Ist der Rückgabewert der Methode ein NULL-Objekt, so wird für die Typen `RES-ANUM-STRING` und `RES-NAT-STRING` das Längenfeld der Zielstruktur auf 0 gesetzt, der Textbereich bleibt unverändert.

Returnwert (RETURN-CODE)

`JCI-RET-OK`

Der Aufruf war erfolgreich.

`JCI-RET-ENOVM`

Es ist keine Java-VM gestartet.

`JCI-RET-ENULLOBJ`

cObj ist `JCI-NULL`.

`JCI-RET-ENULLID`

mId ist `JCI-NULL`.

`JCI-RET-EARGUMENT`

cObj ist kein Klassen-Objekt.

`JCI-RET-EARGVERS`

Die statisch generierte Versionsnummer in *arg* ist ungültig (eventuell überschrieben).

`JCI-RET-ERESVERS`

Die statisch generierte Versionsnummer in *res* ist ungültig (eventuell überschrieben).

`JCI-RET-EARGNUM`

Die Anzahl der übergebenen Argumente (`CallArgNum`) ist kleiner als 0 oder größer als der Wert, der für `<max-arguments>` eingesetzt wurde (siehe Abschnitt „JCI-METHODARGS - Funktionsargumente").

`JCI-RET-EARGTYPE`

Der Wert des Feldes `ArgType` ist ungültig. Das Feld `ResErrIndex` enthält die Nummer des fehlerhaften Argumentes.

`JCI-RET-ERESTYPE`

Der Wert des Feldes `ResType` ist ungültig.

`JCI-RET-EARGCONV`

Bei der Konvertierung eines Argumentes ist ein Fehler aufgetreten.
Das Feld `ResErrIndex` enthält die Nummer des Argumentes, das Feld `ResErrCode` einen genaueren Fehlercode.

`JCI-RET-ERESCONV`

Bei der Konvertierung des Ergebnisses ist ein Fehler aufgetreten.
Das Feld `ResErrCode` enthält einen genaueren Fehlercode.

`JCI-RET-EEXCEPT`

Nach dem Aufruf der Methode existiert eine Exception. Es wird nicht unterschieden, ob die Exception durch diesen oder einen früheren Funktionsaufruf ausgelöst wurde.
Das dem Methodenergebnis entsprechende Feld in der Struktur *res* ist unverändert.

Exceptions

Alle Exceptions, die von der aufgerufenen Methode geworfen werden.

Beispiel

```
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
...
```

7.12.2.4 JCI_GetMethodID

Die Funktion liefert die Methoden-ID (Java-Handle) für eine Instanz-Methode einer Klasse bzw. eines Interfaces. Sie ist äquivalent zur JNI-Funktion `GetMethodID`.

Aufruf

`CALL 'JCI_GetMethodID' USING` *cObj mName mSig mID*

cObj	Klassenobjekt
mName	Name der Methode
mSig	Signatur der Methode
mID	von der Funktion gelieferte Methoden-ID

Argumente

Siehe Funktion `JCI_GetStaticMethodID`.

Returnwert (RETURN-CODE)

Siehe Funktion `JCI_GetStaticMethodID`.

Exceptions

Die von der Funktion geworfenen Exceptions entsprechen denen der JNI-Funktion `GetMethodID`.

Hinweise

Die Methode kann in einer Oberklasse der durch *cObj* referenzierten Klasse definiert sein und von dieser geerbt werden.

Die Methode wird durch den Namen und die Signatur identifiziert. Die Signatur kann durch die Anweisung

```
javap -s <Klassenname>
```

erhalten werden, wobei <`Klassenname`> der Name der durch *cObj* identifizierten Klasse ist.

7.12.2.5 JCI_CallMethod

Die Funktion ruft eine Instanz-Methode auf.
Sie ist äquivalent zu den JNI-Funktionen `Call<type>Method`. Sie bietet jedoch zusätzlich die Möglichkeit, Zeichenketten direkt zu übergeben bzw. zu erhalten.

Aufruf

`CALL 'JCI_CallMethod' USING` *obj mID arg res*

obj	Instanz-Objekt
mI	Methoden-ID
Darg	Methodenargumente
res	Methodenergebnis

Argumente

obj Datenfeld des Typs `JCI-object`
Instanz-Objekt, für das die Methode gerufen werden soll.

mID Datenfeld des Typs `JCI-handle`
ID der Methode, die gerufen werden soll. Die Methoden-ID muss durch den Aufruf der Funktion `JCI_GetStaticMethodID` für die Klasse des Objekts *obj* oder einer seiner Oberklassen beschafft werden.

arg eine Struktur der Form `MethodArg`
Beschreibung der Argumente für den Methodenaufruf (siehe Abschnitt „Argumente und Ergebniswerte von Java-Methoden").

res eine Struktur der Form `MethodRes`
Beschreibung von Rückgabewert für den Methodenaufruf und Fehlerinformation (siehe Abschnitt „Argumente und Ergebniswerte von Java-Methoden"). Ist der Rückgabewert der Methode ein NULL-Objekt, so wird für die Typen `RES-ANUM-STRING` und `RES-NAT-STRING` das Längenfeld der Zielstruktur auf 0 gesetzt, der Textbereich bleibt unverändert.

Returnwert (RETURN-CODE)

`JCI-RET-ENULLOBJ`

obj ist `JCI-NULL`.

Alle übrigen Werte wie in `JCI_CallStaticMethod`.

Exceptions

Alle Exceptions, die von der aufgerufenen Methode geworfen werden.

7.12.2.6 JCI_CallNonvirtualMethod

Die Funktion ruft eine Instanz-Methode einer vorgegebenen Klasse auf.Sie ist äquivalent zu den JNI-Funktionen `CallNonvirtual<type>Method`. Sie bietet jedoch zusätzlich die Möglichkeit, Zeichenketten direkt zu übergeben bzw. zu erhalten.

Aufruf

`CALL 'JCI_CallNonvirtualMethod' USING` *obj* `cObj` *mID arg res*

obj Instanz-Objekt

cObj Objekt der Klasse, in der die Methode definiert ist

mID Methoden-ID

arg Methodenargumente

res Methodenergebnis

Argumente

obj Datenfeld des Typs `JCI-object`
Instanz-Objekt, für das die Methode gerufen werden soll.

cObj Datenfeld des Typs `JCI-object`
Objekt der Klasse, deren Methode aufgerufen werden soll.

mID Datenfeld des Typs `JCI-handle`
ID der Methode, die gerufen werden soll.
Die Methoden-ID muss durch den Aufruf der Funktion `JCI_GetMethodID` für die Klasse *cObj* beschafft werden. Diese Klasse muss mit der Klasse des Objekts *obj* oder einer seiner Oberklassen übereinstimmen.

arg eine Struktur der Form `MethodArg`
Beschreibung der Argumente für den Methodenaufruf (siehe Abschnitt „Argumente und Ergebniswerte von Java-Methoden").

res eine Struktur der Form `MethodRes`
Beschreibung von Rückgabewert für den Methodenaufruf und Fehlerinformation (siehe Abschnitt „Argumente und Ergebniswerte von Java-Methoden"). Ist der Rückgabewert der Methode ein NULL-Objekt, so wird für die Typen `RES-ANUM-STRING` und `RES-NAT-STRING` das Längenfeld der Zielstruktur auf 0 gesetzt, der Textbereich bleibt unverändert.

Returnwert (RETURN-CODE)

`JCI-RET-ENULLOBJ`

obj oder *cObj* ist `JCI-NULL`.

Alle übrigen Werte wie in `JCI_CallStaticMethod`.

Exceptions

Alle Exceptions, die von der aufgerufenen Methode geworfen werden.

7.12.3 Objektreferenzen

Dieser Abschnitt beschreibt die zur Verwaltung von lokalen Objektreferenzen benötigten JCI-Funktionen.

7.12.3.1 JCI_DeleteLocalRef

Die Funktion löscht eine lokale Objektreferenz.
Sie ist äquivalent zur JNI-Funktion `DeleteLocalRef`.

Aufruf

```
CALL 'JCI_DeleteLocalRef' USING obj
```

obj Objektreferenz

Argumente

obj Datenfeld des Typs `JCI-object`
Objektreferenz, die gelöscht werden soll.

Returnwert (RETURN-CODE)

`JCI-RET-OK`

Der Aufruf war erfolgreich.

`JCI-RET-ENOVM`

Es ist keine Java-VM gestartet.

Hinweise

Nach dem Aufruf der Funktion `JCI_DeleteLocalRef` darf die Objektreferenz *obj* nicht mehr verwendet werden.

7.12.3.2 JCI_NewLocalRef

Die Funktion erzeugt eine neue lokale Referenz auf ein Objekt.
Sie ist äquivalent zur JNI-Funktion `NewLocalRef`.

Aufruf

```
CALL 'JCI_NewLocalRef' USING obj newObj
```

obj Objektreferenz

newObj von der Funktion gelieferte Objektreferenz.

Argumente

obj Datenfeld des Typs `JCI-object`
Objektreferenz auf das Objekt, auf das eine neue Referenz erzeugt werden soll.

newObj Datenfeld des Typs `JCI-object`
Neue Objektreferenz auf das durch *obj* referenzierte Objekt.

Returnwert (RETURN-CODE)

`JCI-RET-OK`

Der Aufruf war erfolgreich.

`JCI-RET-ENOVM`

Es ist keine Java-VM gestartet.

7.12.4 Objekte

Dieser Abschnitt beschreibt die zur Erzeugung und Bearbeitung von Java-Objekten benötigten JCI-Funktionen.

7.12.4.1 JCI_NewObject

Die Funktion erzeugt ein neues Java-Objekt.
Sie ist äquivalent zur JNI-Funktion `NewObject`. Sie bietet jedoch zusätzlich die Möglichkeit, Zeichenketten direkt zu übergeben.

Aufruf

`CALL 'JCI_NewObject' USING` *cObj mID arg res*

cObj	Klassenobjekt
mID	Methoden-ID
arg	Konstruktorargumente
res	Ergebnis

Argumente

cObj Datenfeld des Typs `JCI-object`
Klassen-Objekt, für das ein Objekt erzeugt werden soll. Es darf nicht auf eine Array-Klasse verweisen.

mID Datenfeld des Typs `JCI-handle`
ID der Konstruktor-Methode. Die Methoden-ID muss durch den Aufruf der Funktion `JCI_GetMethodID` mit Namen `<init>` und Signatur `(...)V` für die Klasse *cObj* beschafft werden.

arg eine Struktur der Form `MethodArg`
Beschreibung der Argumente für den Konstruktor-Aufruf (siehe Abschnitt „Argumente und Ergebniswerte von Java-Methoden").

res eine Struktur der Form `MethodRes`
Rückgabewert (neues Objekt) und Fehlerinformation (nur Ausgabe, Ergebnis in `ResValObject`). Im Fehlerfall wird JCI-NULL zurückgeliefert.

Returnwert (RETURN-CODE)

`JCI-RET-OK`

Der Aufruf war erfolgreich.

`JCI-RET-ENULLOBJ`

cObj ist `JCI-NULL`.

`JCI-RET-ENULLID`

mId ist `JCI-NULL`.

`JCI-RET-EARGUMENT`

cObj ist kein Klassen-Objekt oder verweist auf ein Array.

`JCI-RET-ENOVM`

Es ist keine Java-VM gestartet.

`JCI-RET-EARGVERS`

Die statisch generierte Versionsnummer in *arg* ist ungültig (eventuell überschrieben).

`JCI-RET-ERESVERS`

Die statisch generierte Versionsnummer in *res* ist ungültig (eventuell überschrieben).

`JCI-RET-EARGNUM`

Die Anzahl der übergebenen Argumente (`CallArgNum`) ist kleiner als 0 oder größer, als der Wert, der für `<max-arguments>` eingesetzt wurde (siehe Abschnitt „JCI-METHODARGS - Funktionsargumente").

`JCI-RET-EARGTYPE`

Der Wert des Feldes `ArgType` ist ungültig. Das Feld `ResErrIndex` enthält die Nummer des fehlerhaften Argumentes.

`JCI-RET-EARGCONV`

Bei der Konvertierung eines Argumentes ist ein Fehler aufgetreten.
Das Feld `ResErrIndex` enthält die Nummer des Argumentes, das Feld `ResErrCode` einen genaueren Fehlercode.

`JCI-RET-ERR`

Das Objekt konnte nicht erzeugt werden.

Exceptions

Alle Exceptions, die vom Konstruktor geworfen werden.

Die übrigen von der Funktion geworfenen Exceptions entsprechen denen der JNI-Funktion `NewObject`.

Beispiel

```
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
...
```

7.12.4.2 JCI_GetObjectClass

Die Funktion liefert das Klassenobjekt eines Objekts.
Sie ist äquivalent zur JNI-Funktion `GetObjectClass`.

Aufruf

`CALL 'JCI_GetObjectClass' USING` *obj cObj*

obj Instanz-Objekt

cObj von der Funktion geliefertes Klassen-Objekt

Argumente

obj Datenfeld des Typs `JCI-object`
Instanz-Objekt, dessen Klassenobjekt geliefert werden soll. Das Objekt darf nicht 0 sein.

cObj Datenfeld des Typs `JCI-object`
Nach erfolgreicher Ausführung der Funktion enthält das Feld das Klassenobjekt der gesuchten Klasse.

Returnwert (RETURN-CODE)

`JCI-RET-OK`

Der Aufruf war erfolgreich.

`JCI-RET-ENOVM`

Es ist keine Java-VM gestartet.

`JCI-RET-ENULLOBJ`

obj ist `JCI-NULL`.

7.12.4.3 JCI_IsInstanceOf

Die Funktion überprüft, ob ein Objekt eine Instanz einer Klasse ist.
Sie ist äquivalent zur JNI-Funktion `IsInstanceOf`.

Aufruf

`CALL 'JCI_IsInstanceOf' USING` *obj cObj*

obj Instanz-Objekt

cObj Klassen-Objekt

Argumente

obj Datenfeld des Typs `JCI-object`
Objekt, das überprüft werden soll. Ist *obj* `JCI-NULL`, so ist es eine Instanz jeder Klasse

cObj Datenfeld des Typs `JCI-object`
Klasse, auf die geprüft werden soll.

Returnwert (RETURN-CODE)

`JCI-RET-TRUE`

`obj` ist eine Instanz von `cObj`.

`JCI-RET-FALSE`

`obj` ist keine Instanz von `cObj`.

`JCI-RET-ENOVM`

Es ist keine Java-VM gestartet.

`JCI-RET-ENULLOBJ`

cObj ist `JCI-NULL`.

`JCI-RET-EARGUMENT`

cObj ist kein Klassen-Objekt.

7.12.4.4 JCI_IsSameObject

Die Funktion überprüft, ob zwei Objektreferenzen auf dasselbe Objekt verweisen.
Sie ist äquivalent zur JNI-Funktion `IsSameObject`.

Aufruf

`CALL 'JCI_IsSameObject' USING` *obj1 obj2*

obj1 Objekt

obj2 Objekt

Argumente

obj1, *obj2*

Datenfelder des Typs `JCI-object`
Objekte, die verglichen werden sollen.

Returnwert (RETURN-CODE)

`JCI-RET-TRUE`

Beide Objektreferenzen verweisen auf dasselbe Objekt oder sind beide `JCI-NULL`.

`JCI-RET-FALSE`

Die Objektreferenzen verweisen auf unterschiedliche Objekte.

`JCI-RET-ENOVM`

Es ist keine Java-VM gestartet.

7.12.5 Felder

Dieser Abschnitt beschreibt die JCI-Funktionen, mit denen Felder in Java-Objekten bearbeitet werden können.

7.12.5.1 JCI_GetStaticFieldID

Die Funktion liefert die Feld-ID (Java-Handle) für ein statisches Feld einer Klasse.
Sie ist äquivalent zur JNI-Funktion `GetStaticFieldID`.

Aufruf

`CALL 'JCI_GetStaticFieldID' USING` *cObj fName fSig fID*

cObj	Klassenobjekt
fName	Name des Feldes
fSig	Signatur des Feldes
fID	von der Funktion gelieferte Feld-ID

Argumente

cObj Datenfeld des Typs `JCI-object`
Objekt der Klasse, in der die Methode gesucht werden soll.

fName Struktur vom Typ `Cobvar`
Name des Feldes, das gesucht werden soll.
Nachfolgende Leerzeichen am Ende des Textes werden ignoriert.

fSig Struktur vom Typ `Cobvar`
Signatur des Feldes, das gesucht werden soll.
Nachfolgende Leerzeichen am Ende des Textes werden ignoriert.

fID Datenfeld des Typs `JCI-handle`
Nach erfolgreicher Ausführung der Funktion enthält das Feld die ID des gesuchten Feldes.
Im Fehlerfall wird der Wert JCI-NULL zurückgeliefert.

Returnwert (RETURN-CODE)

`JCI-RET-OK`

Der Aufruf war erfolgreich.

`JCI-RET-ENOVM`

Es ist keine Java-VM gestartet.

`JCI-RET-ENULLOBJ`

cObj ist `JCI-NULL`.

`JCI-RET-EARGUMENT`

cObj ist kein Klassen-Objekt.

`JCI-RET-ENOTFOUND`

Das Feld konnte nicht gefunden werden.

Exceptions

Die von der Funktion geworfenen Exceptions entsprechen denen der JNI-Funktion `GetStaticFieldID`.

Hinweise

Das Feld wird durch den Namen und die Signatur identifiziert. Die Signatur kann durch die Anweisung

```
javap -s <Klassenname>
```

erhalten werden, wobei <Klassenname> der Name der durch *cObj* identifizierten Klasse ist.

7.12.5.2 JCI_GetStaticField

Die Funktion liefert den Wert eines statischen Feldes einer Klasse.
Sie ist äquivalent zu den JNI-Funktionen `GetStatic<type>Field`. Sie bietet jedoch zusätzlich die Möglichkeit, Zeichenketten direkt zu erhalten.

Aufruf

`CALL 'JCI_GetStaticField' USING` *cObj flD res*

cObj Klassenobjekt

flD Feld ID

res Ergebnis

Argumente

cObj Datenfeld des Typs `JCI-object`
Klassen-Objekt, dessen Feldinhalt geliefert werden soll.

flD Datenfeld des Typs `JCI-handle`
ID des Feldes, dessen Inhalt geliefert werden soll. Die Feld-ID muss durch den Aufruf der Funktion `JCI_GetStaticFieldID` für die Klasse *cObj* beschafft werden.

res eine Struktur der Form `MethodRes`
Beschreibung von Rückgabewert für den Feldinhalt und Fehlerinformation (siehe Abschnitt „Argumente und Ergebniswerte von Java-Methoden"). Ist der Inhalt des Feldes ein NULL-Objekt, so wird für die Typen `RES-ANUM-STRING` und `RES-NAT-STRING` das Längenfeld der Zielstruktur auf 0 gesetzt, der Textbereich bleibt unverändert.

Returnwert (RETURN-CODE)

`JCI-RET-OK`

Der Aufruf war erfolgreich.

`JCI-RET-ENOVM`

Es ist keine Java-VM gestartet.

`JCI-RET-ENULLOBJ`

cObj ist `JCI-NULL`.

`JCI-RET-ENULLID`

flD ist `JCI-NULL`.

`JCI-RET-EARGUMENT`

cObj ist kein Klassen-Objekt.

`JCI-RET-ERESVERS`

Die statisch generierte Versionsnummer in *res* ist ungültig (eventuell überschrieben).

`JCI-RET-ERESTYPE`

Der Wert des Feldes `ResType` ist ungültig.

`JCI-RET-ERESCONV`

Bei der Konvertierung des Ergebnisses ist ein Fehler aufgetreten.
Das Feld `ResErrCode` enthält einen genaueren Fehlercode.

7.12.5.3 JCI_SetStaticField

Die Funktion setzt den Wert eines statischen Feldes einer Klasse.
Sie ist äquivalent zu den JNI-Funktionen `SetStatic<type>Field`. Sie bietet jedoch zusätzlich die Möglichkeit, Zeichenketten direkt zu übergeben.

Aufruf

`CALL 'JCI_SetStaticField' USING` *cObj flD arg res*

cObj Klassenobjekt

flD Feld ID

arg neuer Wert

res Ergebnis

Argumente

cObj Datenfeld des Typs `JCI-object`
Klassen-Objekt, dessen Feldinhalt gesetzt werden soll.

flD Datenfeld des Typs `JCI-handle`
ID des Feldes, dessen Inhalt gesetzt werden soll. Die Feld-ID muss durch den Aufruf der Funktion `JCI_GetStaticFieldID` für die Klasse *cObj* beschafft werden.

arg eine Struktur der Form `MethodArg`
Beschreibung des neuen Wertes für den Feldinhalt (siehe Abschnitt „Argumente und Ergebniswerte von Java-Methoden").
Es wird nur die Teilstruktur `CallArg(1)` benötigt.

res eine Struktur der Form `MethodRes`
Fehlerinformation (nur Ausgabe).

Returnwert (RETURN-CODE)

`JCI-RET-OK`

Der Aufruf war erfolgreich.

`JCI-RET-ENOVM`

Es ist keine Java-VM gestartet.

`JCI-RET-ENULLID`

flD ist `JCI-NULL`.

`JCI-RET-ENULLOBJ`

cObj ist `JCI-NULL`.

`JCI-RET-EARGUMENT`

cObj ist kein Klassen-Objekt.

`JCI-RET-EARGVERS`

Die statisch generierte Versionsnummer in *arg* ist ungültig (eventuell überschrieben).

`JCI-RET-ERESVERS`

Die statisch generierte Versionsnummer in *res* ist ungültig (eventuell überschrieben).

`JCI-RET-EARGTYPE`

Der Wert des Feldes `ArgType` ist ungültig.

`JCI-RET-EARGCONV`

Bei der Konvertierung des Argumentes ist ein Fehler aufgetreten.
Das Feld `ResErrCode` enthält einen genaueren Fehlercode.

7.12.5.4 JCI_GetFieldID

Die Funktion liefert die Feld-ID (Java-Handle) für ein Instanz-Feld einer Klasse.
Sie ist äquivalent zur JNI-Funktion `GetFieldID`.

Aufruf

`CALL 'JCI_GetFieldID' USING` *cObj fName fSig fID*

cObj	Klassenobjekt
fName	Name des Feldes
fSig	Signatur des Feldes
fID	von der Funktion gelieferte Feld-ID

Argumente

Siehe Funktion JCI_GetStaticFieldID.

Returnwert (RETURN-CODE)

Siehe Funktion JCI_GetStaticFieldID.

Exceptions

Die von der Funktion geworfenen Exceptions entsprechen denen der JNI-Funktion `GetFieldID`.

Hinweise

Siehe Funktion JCI_GetStaticField.

7.12.5.5 JCI_GetField

Die Funktion liefert den Wert eines Instanz-Feldes eines Objekts.
Sie ist äquivalent zu den JNI-Funktionen `Get<type>Field`. Sie bietet jedoch zusätzlich die Möglichkeit, Zeichenketten direkt zu erhalten.

Aufruf

`CALL 'JCI_GetField' USING` *obj flD res*

obj Instanz-Objekt

flD Feld ID

res Ergebnis

Argumente

obj Datenfeld des Typs `JCI-object`
Instanz-Objekt, dessen Feldinhalt geliefert werden soll.

flD Datenfeld des Typs `JCI-handle`
ID des Feldes, dessen Inhalt geliefert werden soll. Die Feld-ID muss durch den Aufruf der Funktion `JCI_GetFieldID` beschafft werden.

res eine Struktur der Form `MethodRes`
Beschreibung von Rückgabewert für den Feldinhalt und Fehlerinformation (siehe Abschnitt „Argumente und Ergebniswerte von Java-Methoden"). Ist der Inhalt des Feldes ein NULL-Objekt, so wird für die Typen `RES-ANUM-STRING` und `RES-NAT- STRING` das Längenfeld der Zielstruktur auf 0 gesetzt, der Textbereich bleibt unverändert.

Returnwert (RETURN-CODE)

`JCI-RET-ENULLOBJ`

obj ist `JCI-NULL`.

Alle übrigen Werte wie in JCI_GetStaticField.

7.12.5.6 JCI_SetField

Die Funktion setzt den Wert eines statischen Feldes einer Klasse.
Sie ist äquivalent zu den JNI-Funktionen `Set<type>Field`. Sie bietet jedoch zusätzlich die Möglichkeit, Zeichenketten direkt zu übergeben.

Aufruf

`CALL 'JCI_SetField' USING` *obj fID arg res*

obj Instanz-Objekt

fID Feld ID

arg neuer Wert

res Ergebnis

Argumente

obj Datenfeld des Typs `JCI-object`
Instanz-Objekt, dessen Feldinhalt geändert werden soll.

fID Datenfeld des Typs `JCI-handle`
ID des Feldes, dessen Inhalt gesetzt werden soll. Die Feld-ID muss durch den Aufruf der Funktion `JCI_GetFieldID` beschafft werden.

arg eine Struktur der Form `MethodArg`
Beschreibung des neuen Wertes für den Feldinhalt (siehe Abschnitt „Argumente und Ergebniswerte von Java-Methoden").
Es wird nur die Teilstruktur `CallArg(1)` benötigt.

res eine Struktur der Form `MethodRes`
Fehlerinformation (nur Ausgabe).

Returnwert (RETURN-CODE)

`JCI-RET-ENULLOBJ`

obj ist `JCI-NULL`.

Alle übrigen Werte wie in JCI_GetStaticField.

7.12.6 Zeichenketten

Dieser Abschnitt beschreibt die JCI-Funktionen, mit denen Java-Strings erzeugt und verarbeitet werden können.

7.12.6.1 JCI_NewString

Die Funktion erzeugt ein neues Java-String-Objekt aus einer COBOL-Zeichenkette.
Sie ist äquivalent zur JNI-Funktion `NewString`. Sie bietet jedoch zusätzlich die Möglichkeit, alphanumerische (EBCDIC) Zeichenketten zu übergeben.

Aufruf

`CALL 'JCI_NewString' USING` *arg res*

arg Argumentbeschreibung

res Ergebnisbeschreibung

Argumente

arg eine Struktur der Form `MethodArg`

Beschreibung der Zeichenkette, aus der das String-Objekt erzeugt werden soll (siehe Abschnitt „Argumente und Ergebniswerte von Java-Methoden").
Es wird nur die Teilstruktur `CallArg(1)` benötigt.
Als Wert von `ArgType(1)` ist nur `ARG-ANUM-STRING` bzw. `ARG-NAT-STRING` zulässig.

res eine Struktur der Form `MethodRes`
Rückgabewert und Fehlerinformation (nur Ausgabe, Ergebnis in `ResValObject`).
Im Fehlerfall wird der Wert JCI-NULL zurückgeliefert.

Returnwert (RETURN-CODE)

`JCI-RET-OK`

Der Aufruf war erfolgreich.

`JCI-RET-ENOVM`

Es ist keine Java-VM gestartet.

`JCI-RET-EARGVERS`

Die statisch generierte Versionsnummer in *arg* ist ungültig (eventuell überschrieben).

`JCI-RET-ERESVERS`

Die statisch generierte Versionsnummer in *res* ist ungültig (eventuell überschrieben).

`JCI-RET-EARGTYPE`

Der Wert des Feldes `ArgType` ist ungültig.

`JCI-RET-EARGCONV`

Bei der Konvertierung des Argumentes ist ein Fehler aufgetreten.
Das Feld `ResErrCode` enthält einen genaueren Fehlercode.

`JCI-RET-ERR`

Das Objekt konnte nicht erzeugt werden.

Exceptions

Die von der Funktion geworfenen Exceptions entsprechen denen der JNI-Funktion `NewString`.

7.12.6.2 JCI_GetStringLength

Die Funktion liefert die Länge (Anzahl der Unicode-Zeichen) eines Java-Strings.
Sie ist äquivalent zur JNI-Funktion `GetStringLength`.

Aufruf

`CALL 'JCI_GetStringLength' USING` *sObj len*

sObj String-Objekt

len Länge

Argumente

sObj Datenfeld des Typs `JCI-object`
String-Objekt, dessen Länge geliefert werden soll.

len Datenfeld des Typs `JCI-size`
Nach erfolgreicher Ausführung der Funktion enthält das Feld die Anzahl der Unicode-Zeichen des durch *sObj* referenzierten String-Objekts.

Returnwert (RETURN-CODE)

`JCI-RET-OK`

Der Aufruf war erfolgreich.

`JCI-RET-ENOVM`

Es ist keine Java-VM gestartet.

`JCI-RET-ENULLOBJ`

sObj ist `JCI-NULL`.

`JCI-RET-EARGUMENT`

sObj ist kein String-Objekt.

7.12.6.3 JCI_GetString

Die Funktion kopiert einen Teil eines Java-Strings in einen bereitgestellten Datenbereich.
Sie ist äquivalent zur JNI-Funktion `GetStringRegion`. Sie bietet jedoch zusätzlich die Möglichkeit, alphanumerische (EBCDIC) Zeichenketten zu erhalten.

Aufruf

`CALL 'JCI_GetString' USING` *sObj start res*

sObj String-Objekt

start Startposition

res Ergebnisbeschreibung

Argumente

sObj Datenfeld des Typs `JCI-object`
String-Objekt, dessen Inhalt kopiert werden soll.

start Datenfeld des Typs `JCI-size`
Position des ersten Zeichens, das geliefert werden soll (beginnend bei 1).

res eine Struktur der Form `MethodRes`
Rückgabewert und Fehlerinformation (siehe Abschnitt „Argumente und Ergebniswerte von Java-Methoden").
Als Wert von `ResType` ist nur `RES-ANUM-STRING` bzw. `RES-NAT-STRING` zulässig.

Returnwert (RETURN-CODE)

`JCI-RET-OK`

Der Aufruf war erfolgreich.

`JCI-RET-ENOVM`

Es ist keine Java-VM gestartet.

`JCI-RET-ENULLOBJ`

sObj ist `JCI-NULL`.

`JCI-RET-EARGUMENT`

sObj ist kein String-Objekt.

`JCI-RET-EINDAOB`

start ist kleiner als 1 oder größer als die Anzahl der Zeichen des Java-Strings.

`JCI-RET-ERESVERS`

Die statisch generierte Versionsnummer in *res* ist ungültig (eventuell überschrieben).

`JCI-RET-ERESTYPE`

Der Wert des Feldes `ResType` ist ungültig.

`JCI-RET-ERESCONV`

Bei der Konvertierung des Strings ist ein Fehler aufgetreten.
Das Feld `ResErrCode` enthält einen genaueren Fehlercode.

Hinweise

Die Übertragung findet maximal in der Länge (Länge Java-String – *start* + 1) bzw. des Wertes *len* in der Ausgabestruktur statt.

Beispiel

```
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
...
```

7.12.7 Arrays

Dieser Abschnitt beschreibt die JCI-Funktionen, mit denen Java-Arrays erzeugt und verarbeitet werden können.

7.12.7.1 JCI_GetArrayLength

Die Funktion ist äquivalent zur JNI-Funktion `GetArrayLength`.

Aufruf

`CALL 'JCI_GetArrayLength' USING` *aObj num*

aObj Array-Objekt

num Anzahl Elemente

Argumente

aObj Datenfeld des Typs `JCI-object`
Array-Objekt, dessen Elementzahl geliefert werden soll.

num Datenfeld des Typs `JCI-size`
Nach erfolgreicher Ausführung der Funktion enthält das Feld die Anzahl der Elemente des durch *aObj* referenzierten Array-Objekts.

Returnwert (RETURN-CODE)

`JCI-RET-OK`

Der Aufruf war erfolgreich.

`JCI-RET-ENOVM`

Es ist keine Java-VM gestartet.

`JCI-RET-ENULLOBJ`

aObj ist `JCI-NULL`.

`JCI-RET-EARGUMENT`

aObj ist kein Array-Objekt.

7.12.7.2 JCI_NewObjectArray

Die Funktion erzeugt ein Array-Objekt für Objekt-Elemente.
Sie ist äquivalent zur JNI-Funktion `NewObjectArray`.

Aufruf

`CALL 'JCI_NewObjectArray' USING` *num cObj eObj res*

num	Anzahl der Elemente
cObj	Element-Klasse
eObj	Element-Initialwert
res	Ergebnisbeschreibung

Argumente

num Datenfeld des Typs `JCI-size`
Anzahl der Elemente des Arrays.

cObj Datenfeld des Typs `JCI-object`
Klassen-Objekt für die Klasse der Array-Elemente.

eObj Datenfeld des Typs `JCI-object`
Initialwert für die Array-Elemente (darf auch JCI-NULL sein).

res eine Struktur der Form `MethodRes`
Rückgabewert (neue Objektreferenz) in `ResValObject`. Im Fehlerfall wird der Wert JCI-NULL zurückgeliefert.

Returnwert (RETURN-CODE)

`JCI-RET-OK`

Der Aufruf war erfolgreich.

`JCI-RET-ENULLOBJ`

cObj ist `JCI-NULL`.

`JCI-RET-EARGUMENT`

cObj ist kein Klassen-Objekt.

`JCI-RET-EINDAOB`

num ist kleiner als 0.

`JCI-RET-ENOVM`

Es ist keine Java-VM gestartet.

`JCI-RET-ERESVERS`

Die statisch generierte Versionsnummer in *res* ist ungültig (eventuell überschrieben).

`JCI-RET-ERR`

Das Array-Objekt konnte nicht erzeugt werden.

Exceptions

Die von der Funktion geworfenen Exceptions entsprechen denen der JNI-Funktion `NewObjectArray`.

Beispiel

```
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
...
```

7.12.7.3 JCI_GetObjectArrayElement

Die Funktion liefert ein Element eines Objekt-Arrays.
Sie ist äquivalent zur JNI-Funktion `GetObjectArrayElement`. Sie bietet jedoch zusätzlich die Möglichkeit, Zeichenketten anstelle von String-Objekten zu erhalten.

Aufruf

`CALL 'JCI_GetObjectArrayElement' USING` *aObj index res*

aObj Array-Objekt

index Array-index

res Ergebnisbeschreibung

Argumente

aObj Datenfeld des Typs `JCI-object`
Array-Objekt, dessen Element geliefert werden soll.

index Datenfeld des Typs `JCI-size`
Position des Elementes im Array, das geliefert werden soll (beginnend bei 1).

res eine Struktur der Form `MethodRes`
Rückgabewert und Fehlerinformation (siehe Abschnitt „Argumente und Ergebniswerte von Java-Methoden").
Als Wert von `ResType` sind nur `RES-OBJECT`, `RES-ANUM-STRING` und `RES-NAT-STRING` zulässig. Ist das Array-Element ein NULL-Objekt, so wird für die Typen `RES-ANUM-STRING` und `RES-NAT-STRING` das Längenfeld der Zielstruktur auf 0 gesetzt, der Textbereich bleibt unverändert.

Returnwert (RETURN-CODE)

`JCI-RET-OK`

Der Aufruf war erfolgreich.

`JCI-RET-ENOVM`

Es ist keine Java-VM gestartet.

`JCI-RET-ENULLOBJ`

aObj ist `JCI-NULL`.

`JCI-RET-EARGUMENT`

aObj ist kein Array-Objekt.

`JCI-RET-ERESVERS`

Die statisch generierte Versionsnummer in *res* ist ungültig (eventuell überschrieben).

`JCI-RET-ERESTYPE`

Der Wert des Feldes `ResType` ist ungültig.

`JCI-RET-ERESCONV`

Bei der Konvertierung des Elementes ist ein Fehler aufgetreten.
Das Feld `ResErrCode` enthält einen genaueren Fehlercode.

`JCI-RET-EINDAOB`

index ist kleiner als 1 oder größer als die Anzahl der Elemente des Arrays.

Beispiel

```
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
...
```

7.12.7.4 JCI_SetObjectArrayElement

Die Funktion setzt ein Element eines Objekt-Arrays.
Sie ist äquivalent zur JNI-Funktion `SetObjectArrayElement`. Sie bietet jedoch zusätzlich die Möglichkeit, Zeichenketten anstelle von String-Objekten zu übergeben.

Aufruf

`CALL 'JCI_SetObjectArrayElement' USING` *aObj index arg res*

aObj	Array-Objekt
index	Array-index
arg	Argumentbeschreibung
res	Ergebnisbeschreibung

Argumente

aObj	Datenfeld des Typs `JCI-object`Array-Objekt, das geändert werden soll.
index	Datenfeld des Typs `JCI-size` Position des Elementes im Array, das gesetzt werden soll (beginnend bei 1).
arg	eine Struktur der Form `MethodArg` Beschreibung des neuen Wertes für das Array-Element (siehe Abschnitt „Argumente und Ergebniswerte von Java-Methoden"). Es wird nur die Teilstruktur `CallArg(1)` benötigt. Als Wert von `ArgType(1)` sind nur `ARG-OBJECT`, `ARG-ANUM-STRING` und `ARG-NAT-STRING` zulässig.
res	eine Struktur der Form `MethodRes` Fehlerinformation (nur Ausgabe).

Returnwert (RETURN-CODE)

`JCI-RET-OK`

Der Aufruf war erfolgreich.

`JCI-RET-ENOVM`

Es ist keine Java-VM gestartet.

`JCI-RET-ENULLOBJ`

aObj ist `JCI-NULL`.

`JCI-RET-EARGUMENT`

aObj ist kein Array-Objekt.

`JCI-RET-EARGVERS`

Die statisch generierte Versionsnummer in *elem* ist ungültig (eventuell überschrieben).

`JCI-RET-ERESVERS`

Die statisch generierte Versionsnummer in *res* ist ungültig (eventuell überschrieben).

`JCI-RET-EARGTYPE`

Der Wert des Feldes `ArgType` ist ungültig.

`JCI-RET-EARGCONV`

Bei der Konvertierung des Argumentes ist ein Fehler aufgetreten.
Das Feld `ResErrCode` enthält einen genaueren Fehlercode.

`JCI-RET-EINDAOB`

index ist kleiner als 1 oder größer als die Anzahl der Elemente des Arrays.

Exceptions

Die von der Funktion geworfenen Exceptions entsprechen denen der JNI-Funktion `SetObjectArrayElement`.

7.12.7.5 JCI_NewArray

Die Funktion erzeugt ein mit binären Nullen initialisiertes Array-Objekt für nicht-Objekt-Elemente.
Sie ist äquivalent zu den JNI-Funktionen `New<PrimitiveType>Array`.

Aufruf

`CALL 'JCI_NewArray' USING` *num arg res*

num Anzahl der Elemente

arg Elementbeschreibung

res Ergebnisbeschreibung

Argumente

num Datenfeld des Typs `JCI-size`
Anzahl der Elemente des Arrays.

arg eine Struktur der Form `MethodArg`
Typbeschreibung der Array-Elemente.
Es wird nur das Feld `ArgType(1)` benötigt.
`ArgType(1)` darf weder `ARG-OBJECT` noch `ARG-ANUM-STRING` noch `ARG-NAT-STRING` sein.

res eine Struktur der Form `MethodRes`
Rückgabewert (neue Objektreferenz) in `ResValObject`. Im Fehlerfall wird der Wert JCI-NULL zurückgeliefert.

Returnwert (RETURN-CODE)

`JCI-RET-OK`

Der Aufruf war erfolgreich.

`JCI-RET-ENOVM`

Es ist keine Java-VM gestartet.

`JCI-RET-EINDAOB`

num ist kleiner als 0.

`JCI-RET-EARGVERS`

Die statisch generierte Versionsnummer in *elem* ist ungültig (eventuell überschrieben).

`JCI-RET-ERESVERS`

Die statisch generierte Versionsnummer in *res* ist ungültig (eventuell überschrieben).

`JCI-RET-EARGTYPE`

Der Wert des Feldes `ArgType` ist ungültig.

`JCI-RET-ERR`

Das Array-Objekt konnte nicht erzeugt werden.

Exceptions

Die von der Funktion geworfenen Exceptions entsprechen denen der JNI-Funktionen `New<PrimitiveType>Array`.

7.12.7.6 JCI_GetArray

Die Funktion kopiert Elemente eines Java-Arrays in eine bereitgestellte COBOL-Tabelle.
Sie ist äquivalent zu den JNI-Funktionen `Get<PrimitiveType>ArrayRegion`.

Aufruf

`CALL 'JCI_GetArray' USING` *aObj start num res*

aObj Array-Objekt

start Startposition

num Anzahl

res Ergebnisbeschreibung

Argumente

aObj Datenfeld des Typs `JCI-object`
Array-Objekt, dessen Elemente kopiert werden sollen.

start Datenfeld des Typs `JCI-size`
Position des ersten Elementes im Java-Array, das übertragen werden soll (beginnend bei 1).

num Datenfeld des Typs `JCI-size`
Maximale Anzahl der Elemente, die übertragen werden sollen.
Nach dem Aufruf enthält *num* die Anzahl der Elemente, die tatsächlich übertragen wurden.

res eine Struktur der Form `MethodRes`
Rückgabewert und Fehlerinformation (siehe Abschnitt „Argumente und Ergebniswerte von Java-Methoden").
`ResType` muss entsprechend des COBOL-Datentyps der Tabellenelemente gesetzt werden. Es sind weder `RES-OBJECT` noch `RES-ANUM-STRING` noch `RES-NAT-STRING` zulässig.
Die Adresse der COBOL-Tabelle, in die die Elemente kopiert werden sollen, wird unabhängig vom Datentyp immer im Feld `ResValAddr` übergeben.

Returnwert (RETURN-CODE)

`JCI-RET-OK`

Der Aufruf war erfolgreich.

`JCI-RET-ENOVM`

Es ist keine Java-VM gestartet.

`JCI-RET-ENULLOBJ`

aObj ist `JCI-NULL`.

`JCI-RET-EARGUMENT`

aObj ist kein Array-Objekt.

`JCI-RET-EINDAOB`

num ist kleiner als 0 oder *start* ist kleiner als 1 oder größer als die Anzahl der Elemente des Arrays.

`JCI-RET-ERESVERS`

Die statisch generierte Versionsnummer in *res* ist ungültig (eventuell überschrieben).

`JCI-RET-ERESTYPE`

Der Wert des Feldes `ResType` ist ungültig.

`JCI-RET-ERESCONV`

Bei der Konvertierung der Tabellenelemente ist ein Fehler aufgetreten.
Das Feld `ResErrIndex` enthält die Nummer des COBOL-Tabellenelementes (beginnend bei 1), das Feld `ResErrCode` einen genaueren Fehlercode.
Alle Elemente bis zum fehlerhaften Element werden übertragen, alle nachfolgenden Felder der COBOL-Tabelle bleiben unverändert.

Hinweise

Es werden maximal (Anzahl Array-Elemente - *start* + 1) bzw. *num* Elemente übertagen.

7.12.7.7 JCI_SetArray

Die Funktion kopiert eine COBOL-Tabelle in die Elemente eines Java-Arrays.
Sie ist äquivalent zu den JNI-Funktionen `Set<PrimitiveType>ArrayRegion`.

Aufruf

`CALL 'JCI_SetArray' USING` *aObj start num arg res*

aObj	Array-Objekt
start	Startposition
num	Anzahl
arg	Argumentbeschreibung
res	Ergebnisbeschreibung

Argumente

aObj Datenfeld des Typs `JCI-object`
Array-Objekt, dessen Elemente gesetzt werden sollen.

start Datenfeld des Typs `JCI-size`
Position des ersten Elementes im Java-Array, das überschrieben werden soll (beginnend bei 1).

num Datenfeld des Typs `JCI-size`
Maximale Anzahl der Elemente, die übertragen werden sollen.
Nach dem Aufruf enthält *num* die Anzahl der Elemente, die tatsächlich übertragen wurden, im Fehlerfall 0.

arg eine Struktur der Form `MethodArg`
Beschreibung der Array-Elemente.
Es werden nur die Felder `ArgType(1)` und `ArgValAddr(1)` benötigt.
`ArgType(1)` muss entsprechend des COBOL-Datentyps der Tabellenelemente gesetzt werden. Es sind weder `ARG-OBJECT` noch `ARG-ANUM-STRING` noch `ARG-NAT-STRING` zulässig.
Die Adresse der COBOL-Tabelle, aus der die Elemente kopiert werden sollen, wird unabhängig vom Datentyp immer im Feld `ArgValAddr(1)` übergeben.

res eine Struktur der Form `MethodRes`
Fehlerinformation (nur Ausgabe).

Returnwert (RETURN-CODE)

`JCI-RET-OK`

Der Aufruf war erfolgreich.

`JCI-RET-ENOVM`

Es ist keine Java-VM gestartet.

`JCI-RET-ENULLOBJ`

aObj ist `JCI-NULL`.

`JCI-RET-EARGUMENT`

aObj ist kein Array-Objekt.

`JCI-RET-EARGVERS`

Die statisch generierte Versionsnummer in *elem* ist ungültig (eventuell überschrieben).

`JCI-RET-ERESVERS`

Die statisch generierte Versionsnummer in *res* ist ungültig (eventuell überschrieben).

`JCI-RET-EARGTYPE`

Der Wert des Feldes `ArgType` ist ungültig.

`JCI-RET-EARGCONV`

Bei der Konvertierung der Tabellenelemente ist ein Fehler aufgetreten.Das Feld `ResErrIndex` enthält die Nummer des COBOL-Tabellenelementes (beginnend bei 1), das Feld `ResErrCode` einen genaueren Fehlercode.
Tritt bei einem Element ein Konvertierungsfehler auf, so findet keine Übertragung statt, d. h. alle Felder des Java-Arrays bleiben unverändert

`JCI-RET-EINDAOB`

num ist kleiner als 0 oder *start* ist kleiner als 1 oder größer als die Anzahl der Elemente des Arrays.

Hinweise

Es werden maximal (Anzahl Array-Elemente - *start* + 1) bzw. *num* Elemente übertagen.

7.12.8 Exceptions

Dieser Abschnitt beschreibt die zur Verarbeitung von Java-Exceptions benötigten JCI-Funktionen.

7.12.8.1 JCI_ExceptionCheck

Die Funktion überprüft, ob eine anstehende Exception existiert.
Sie ist äquivalent zur JNI-Funktion `ExceptionCheck`.

Aufruf

```
CALL 'JCI_ExceptionCheck'
```

Returnwert (RETURN-CODE)

`JCI-RET-TRUE`

Es steht eine Exception an.

`JCI-RET-FALSE`

Es steht keine Exception an.

Hinweise

Wird die Funktion aufgerufen, ohne dass die Java-VM gestartet wurde, wird `JCI-RET-FALSE` zurückgeliefert.

7.12.8.2 JCI_ExceptionOccurred

Die Funktion überprüft, ob eine anstehende Exception existiert und liefert das zugehörige Exception-Objekt. Sie ist äquivalent zur JNI-Funktion `ExceptionOccurred`.

Aufruf

`CALL 'JCI_ExceptionOccurred' USING` *eObj*

eObj Exception-Objekt

Argumente

eObj Datenfeld des Typs `JCI-object`
Referenz auf das anstehende Exception-Objekt.
Wurde kein Objekt geworfen, wird `JCI-NULL` zurückgeliefert.

Returnwert (RETURN-CODE)

`JCI-RET-OK`

Der Aufruf war erfolgreich.

Hinweise

Wird die Funktion aufgerufen, ohne dass die Java-VM gestartet wurde, wird `JCI-NULL` und `JCI-RET-OK` zurückgeliefert.

7.12.8.3 JCI_ExceptionDescribe

Die Funktion gibt Informationen in englischer Sprache über eine anstehende Exception nach `stderr` aus. Sie ist äquivalent zur JNI-Funktion `ExceptionDescribe`.

Aufruf

```
CALL 'JCI_ExceptionDescribe'
```

Returnwert (RETURN-CODE)

`JCI-RET-OK`

Der Aufruf war erfolgreich.

Hinweise

Die Funktion darf auch aufgerufen werden, wenn keine Java-VM gestartet wurde.

Wurde die VM nicht gestartet oder steht keine Exception an, so entfällt die Ausgabe.

Wurde das Programm von der BS2000-Kommandooberfläche gestartet, so geht die Ausgabe nach SYSOUT.

7.12.8.4 JCI_ExceptionClear

Die Funktion entfernt eine anstehende Exception.
Sie ist äquivalent zur JNI-Funktion `ExceptionClear`.

Aufruf

```
CALL 'JCI_ExceptionClear'
```

Returnwert (RETURN-CODE)

`JCI-RET-OK`

Der Aufruf war erfolgreich.

Hinweise

Die Funktion darf auch aufgerufen werden, wenn keine Java-VM gestartet wurde oder keine Exception ansteht.

Beispiel

```
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.
...
```

Wenn die Klasse *hello* nicht existiert, sieht die Ausgabe in etwa folgendermaßen aus:

```
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)
```

7.12.9 Weitere Funktionen

Dieser Abschnitt beschreibt alle JCI-Funktionen, für die es keine äquivalenten JNI-Funktionen gibt.

7.12.9.1 JCI_GetVersion

Die Funktion liefert die Version des Java-COBOL-Interface-Moduls.

Aufruf

```
CALL 'JCI_GetVersion' USING vers
```

vers Version

Argumente

vers Datenfeld des Typs `JCI-int`
Datenfeld, in das die Versionsnummer des JCI übertragen werden soll.

Returnwert (RETURN-CODE)

`JCI-RET-OK`

Der Aufruf war erfolgreich.

Hinweise

Die Version der in der COBOL-Anwendung verwendeten COPY-Elemente ist in *JCI-CONST* als `JCI-interface-version` definiert. Diese darf nicht größer sein, als die durch `JCI_GetVersion` gelieferte Version.

7.12.9.2 JCI_GetErrorInformation

Die Funktion liefert nähere Fehlerinformationen.

Aufruf

`CALL 'JCI_GetErrorInformation' USING` *elnf*

elnf Fehlerinformation

Argumente

elnf eine Struktur vom Typ `Cobvar`
Struktur, in die die maximal 256 Zeichen lange Fehlerinformation des JCI übertragen werden soll.
Die Übertragung erfolgt maximal in der Länge des Längenfeldes. Wenn dieses kleiner oder gleich 0 ist oder keine Fehlerinformation existiert, so wird es auf 0 gesetzt und die Übertragung entfällt.
Steht keine Fehlerinformation zur Verfügung, so wird das Längenfeld in *elnf* auf 0 gesetzt und der Textbereich bleibt unverändert.

Returnwert (RETURN-CODE)

`JCI-RET-OK`

Der Aufruf war erfolgreich.

Hinweise

Diese Funktion kann immer aufgerufen werden, auch wenn der Aufruf von Funktionen des JCI als ungültig beschrieben ist, wie z.B. in Abschnitt „Exceptions".

Von allen JCI-Funktionen, die einen Fehler-Returncode liefern können, wird im Fehlerfall eine nähere Information zu diesem Fehler in einem gemeinsamen Feld hinterlegt. Tritt kein Fehler auf, wird das Feld gelöscht. Dadurch steht immer nur die Information der zuletzt aufgerufenen Funktion zur Verfügung. Der Text ist in englischer Sprache und nur für die Ausgabe an den Benutzer gedacht.

Nach Aufruf der Funktion `JCI_GetErrorInformation` steht die Fehlerinformation nicht mehr für weitere Aufrufe zur Verfügung.

Beispiel

```
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
...
```

7.13 Beispiele

In diesem Beispiel stehen alle Java-Sourcen im Verzeichnis `/myhome/jcitest` bereit.

JENV sei im Verzeichnis `/myjava` unter der Kennung `$MYJAVA` installiert.

7.13.1 Java-Klasse

In der Datei `Hello.java` sei folgende Klasse definiert:

```
class Hello {
  public static void hello(String arg)
  {
    System.out.println(">> Hello " + arg + "!");
  }
}
```

7.13.2 Übersetzen des Java-Codes

Die oben definierte Java-Klasse kann jetzt einfach mit dem Kommando

```
javac /myhome/jcitest/Hello.java
```

übersetzt werden.

Der Aufruf erzeugt die Datei `Hello.class` im Verzeichnis `/myhome/jcitest`.

Der Aufruf von

```
javap -s -cp /myhome/jcitest Hello
```

liefert unter Anderem die Signatur der Methode `hello`:

```
public static void hello(java.lang.String);
descriptor: (Ljava/lang/String;)V
```

7.13.3 COBOL-Programm

Das COBOL-Programm *HELLO* sei in der Datei `Hello.cob` folgendermaßen implementiert:

```
       >>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 ">> Bitte Name eingeben" 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.
```

7.13.4 Übersetzen des COBOL-Programms im POSIX

In diesem Beispiel liegt das COBOL-Quellprogramm im POSIX-Verzeichnis `/myhome/jcitest`.

Zum Übersetzen des COBOL-Programms *HELLO* sind folgende Kommandos notwendig:

```
export COBLIB='/myjava/include'
cobol -c -C PERMIT-STANDARD-DEVIATION=YES \
/myhome/jcitest/Hello.cob
```

Als Ergebnis steht die Objektdatei `Hello.o` zur Verfügung.

7.13.5 Binden des COBOL-Programms im POSIX

Beim Binden der Anwendung muss berücksichtigt werden, dass die Laufzeitroutinen für die Sprachen C/C++ und COBOL aus der Java-Laufzeitbibliothek und nicht aus dem CRTE eingebunden werden.

Mit den folgenden Kommandos kann die Anwendung gebunden werden:

```
export BLSLIB00='$MYJAVA.SYSLNK.JENV.090.GREEN-JAVA'
cobol -M HELLO -o Hello Hello.o -l BLSLIB
```

7.13.6 Ablauf des COBOL-Programms im POSIX

Da für dieses Beispiel nicht der Standard-Installationspfad von JENV verwendet werden soll, muss die Umgebungsvariable JAVA_HOME vor dem Aufruf des Programms entsprechend gesetzt werden.

Aufruf und Ablauf sehen dann folgendermaßen aus:

```
export JAVA_HOME=/myjava/jre
Hello
>> Bitte Name eingeben
Susanne
>> Hello Susanne!
```

7.13.7 Übersetzen des COBOL-Programms unter der BS2000- Kommandooberfläche

In diesem Beispiel liegt das COBOL-Quellprogramm in der LMS-Bibliothek `SRC.LIB`, die JCI-COPY-Elemente aber im POSIX-Verzeichnis `/myjava/include`.

Zum Übersetzen sind daher folgende Kommandos notwendig:

```
/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)
```

7.13.8 Binden des COBOL-Programms unter der BS2000- Kommandooberfläche

Zusätzlich zu Funktionen und CRTE aus der Java-Laufzeitbibliothek müssen die POSIX-Schalter eingebunden werden:

```
/START-BINDER
//START-LLM-CREATION HELLO
//INCLUDE LIB=MOD.LIB,ELEM=HELLO
//INCLUDE LIB=$.SYSLNK.CRTE.POSIX
//RESOLVE LIB=$MYJAVA.SYSLNK.JENV.090.GREEN-JAVA
//SAVE-LLM LIB=LLM.LIB,ELEM=HELLO
//END
```

7.13.9 Ablauf des COBOL-Programms unter der BS2000- Kommandooberfläche

Vor dem Start der Anwendung muss die POSIX-Umgebung für den Ablauf initialisiert werden. Das COBOL-Laufzeitsystem verhält sich dann so, als wäre es unter der POSIX-Shell gestartet worden (siehe „COBOL2000 (BS2000) Benutzerhandbuch“ [5]).

Nach Beendigung der Anwendung muss die POSIX-Umgebung durch den Aufruf der Prozedur *DELETE* unbedingt wieder zurückgesetzt werden. Andernfalls ist die Umgebung für weitere Übersetzungen falsch eingestellt.

Aufruf und Ablauf sehen unter der Kennung $MYHOME dann folgendermaßen aus:

```
/CALL-PROCEDURE *LIB($MYJAVA.SYSPRC.JENV.090,INITIALIZE),
     (PWD='myhome/work',JAVA-HOME='/myjava/jre')
/START-PROGRAM *MODULE(LIBRARY=LLM.LIB,ELEMENT=HELLO,
PROGRAM-MODE=ANY,RUN-MODE=*ADVANCED(SHARE-SCOPE=*NONE))
%  BLS0523 ELEMENT 'HELLO', VERSION '@', TYPE 'L' FROM LIBRARY
':LUNB:$MYHOME.LLM.LIB' IN PROCESS
%  BLS0524 LLM 'HELLO', VERSION ' ' OF '2016-04-13 15:17:10' LOADED
>> Bitte Name eingeben
Susanne
>> Hello Susanne!
/CALL-PROCEDURE *LIB($MYJAVA.SYSPRC.JENV.090,DELETE)
```

8 Kommandos für BS2000

Die zum JDK gehörenden Tools sind in „JDK Tools and Utilities“ [11] beschrieben. JENV unterstützt alle dort für Solaris aufgeführten Tools, mit folgenden Ausnahmen:

- Monitoring und Management Tools *jps* , *jstat* , *jstatd*
- Troubleshooting Tools *jcmd* , *jinfo* , *jhat* , *jmap* , *jsadebugd* , *jstack*
- Scripting Tool *jrunscript*.

In diesem Kapitel werden nur die von der Beschreibung in „JDK Tools and Utilities“ [11] abweichenden Kommandos berücksichtigt. Dies sind im Einzelnen:

- Die Kommandos mk_shobj und pr_shobj .
  Diese bietet JENV zusätzlich zur Unterstützung der Shared Object Beschreibungsdateien an.
- Kommando java
  Dessen Optionen weichen von den für Solaris beschriebenen ab.
- Kommando *native2ascii*
  Dieses wird hier wegen seiner größeren Bedeutung in EBCDIC-Umgebung genauer beschrieben.
- Kommandos jconsole und jdb

8.1 mk_shobj

Das Kommando *mk_shobj* erzeugt und bearbeitet Beschreibungsdateien für Shared Objects.

Syntax

mk_shobj [Optionen ...] dateiname

Optionen ...

Eine oder mehrere Kommandozeilenoptionen, durch Leerzeichen getrennt.

dateiname

Beschreibungsdatei für Shared Objects im POSIX-Dateisystem, die *mk_shobj* erzeugen soll.

Beschreibung

Das Kommando *mk_shobj* erzeugt eine Beschreibungsdatei für Shared Objects im POSIX-Dateisystem. Diese Beschreibungsdatei wird vom Java-Interpreter ausgewertet, wenn native Methoden nachgeladen werden (Methode *loadLibrary()* bzw. *load()* der Klassen *Runtime* und *System*).

Die Namen der Beschreibungsdateien müssen so gebildet sein, dass sie von der VM nach dem oben beschriebenen Suchverfahren unter Nutzung von Shared Objects aus Java gefunden werden können, also z.B. mit dem Präfix *lib* beginnend und mit dem Suffix *.so* endend.

Optionen

-?

Ausgabe von Hilfeinformationen zum Kommando.

-l *lib*

Spezifikation der PLAM-Bibliothek (im BS2000), in der der nachzuladende LLM abgelegt ist.

-o *userid*

BS2000-Benutzerkennung, unter der die PLAM-Bibliothek *lib* installiert ist. Dabei bedeutet „**.**“ die aktuelle Benutzerkennung, „*$*“ die Systemkennung, die Form *%name*, dass die zu nutzende Benutzerkennung zur Laufzeit der Umgebungsvariablen *name* zu entnehmen ist und jede andere Angabe den Namen einer Benutzerkennung.

Standard: Aktuelle Benutzerkennung.

-m *modulname*

Spezifikation des Moduls, der nachgeladen werden soll. Diese Option kann mehrfach angegeben werden, wobei dann alle angegebenen Module dynamisch nachgeladen werden. Der Modulname darf nicht länger als 32 Zeichen sein.

-n *dateiname*

Spezifikation eines benötigten Shared Objects (Beschreibungsdatei). Das hier an-gegebene Shared Object wird vor dem primären Shared Object nachgeladen. Diese Option kann mehrfach angegeben werden, wobei dann alle benötigten Shared Objects vor dem aktuellen Shared Object geladen werden.

-u

Die angegebene Beschreibungsdatei muss existieren und wird mit den angegebenen Informationen aktualisiert. Dies kann z.B. verwendet werden, um die Benutzerkennung nachträglich zu modifizieren. Ist *-u* nicht angegeben, wird die Beschreibungsdatei neu angelegt.

-f cpp

Dieses Flag muss gesetzt werden, wenn das Shared Object in C++ implementiertworden ist, damit die notwendigen Laufzeitbibliotheken nachgeladen und initialisiert werden können.

-d

Wenn dieses Flag gesetzt ist, wird der Modul in den Defaultkontext *LOCAL#DEFAULT* geladen.

-c ctxt

Der Modul wird in den angegebenen Kontext geladen.

Beispiel

Das Kommando

```
mk_shobj -l syslnk.hello -m helloworld libhello.so
```

legt im aktuellen Dateiverzeichnis des POSIX-Dateisystems die Datei *libhello.so* an. Es hinterlegt, dass beim Laden von *hello* der Modul *helloworld* aus der PLAM-Bibliothek *syslnk.hello* der aktuellen Benutzerkennung nachgeladen werden soll. *loadLibrary(hello*) wird dabei vom Java-Interpreter ergänzt zu *libhello.so*.

8.2 pr_shobj

Das Kommando *pr_shobj* gibt den Inhalt einer Shared Object Beschreibungsdatei aus.

Syntax

pr_shobj dateiname

dateiname

Beschreibungsdatei, deren Inhalt ausgegeben werden soll.

Beschreibung

Das Kommando *pr_shobj* gibt den Inhalt einer Shared Object Beschreibungsdatei aufbereitet nach *stdout* aus.

Beispiel

```
pr_shobj libhello.so
```

Ausgabe:

```
Library: syslnk.hello
UserID : .
Module : helloworld
```

8.3 java

Option zur Auswahl des HotSpotTM VM-Typs

-client

Es wird die HotSpotTM Client-VM verwendet. Diese VM optimiert den generierten Objektcode für Kurzläuferprogramme (voreingestellt).

-server

Die Option wird nicht unterstützt.

-d32
-d64

Die Optionen werden nicht unterstützt.

Optionen zur Auswahl der HSI-Variante

-s390 Es wird die S390-Variante von JENV verwendet (falls verfügbar). Diese Option ist nur dann sinnvoll, wenn auf einem System sowohl die S390-Variante als auch die X86-Variante von JENV installiert wurde und explizit eine Variante zum Ablauf ausgewählt werden soll.

Diese Option hat Vorrang vor einer Angabe in der Umgebungsvariablen *JENV_SYSHSI* (siehe Kapitel „Umgebungsvariablen").

Standardmäßig, d.h. wenn auch der Umgebungsvariable *JENV_SYSHSI* kein Wert zugewiesen wurde, wird die dem System entsprechende Variante verwendet.

-x86 Es wird die X86-Variante von JENV verwendet (falls verfügbar). Diese Option ist nur dann sinnvoll, wenn auf einem SQ-System sowohl die S390-Variante als auch die X86-Variante von JENV installiert wurde und explizit eine Variante zum Ablauf ausgewählt werden soll.

Diese Option hat Vorrang vor einer Angabe in der Umgebungsvariablen *JENV_SYSHSI* (siehe Kapitel „Umgebungsvariablen").

Standardmäßig, d.h. wenn auch der Umgebungsvariable *JENV_SYSHSI* kein Wert zugewiesen wurde, wird die dem System entsprechende Variante verwendet.

Non-Standardoptionen

-Xmaxjitcodesize *size*

Im Gegensatz zur Originalbeschreibung wird die Cachegröße *size* ohne ein Gleichheitszeichen angegeben, z. B.:

```
-Xmaxjitcodesize48m
```

Steuern des Java-Heapspeichers

Die folgenden Optionen ermöglichen dem Benutzer die Steuerung der speicherbereinigten Heap-Erweiterung bzw. -Verkleinerung. Da die Standardeinstellungen für die Heap-Erweiterung für die meisten Anwendungen geeignet sind, ist die Verwendung dieser Optionen in den meisten Situationen nicht erforderlich. Sie sollten sie nur verwenden, wenn Sie die Auswirkungen der Optionen auf die jeweilige Anwendung verstehen. Durch das willkürliche Einstellen dieser Optionen kann die Leistung des Systems ebenso leicht beeinträchtigt wie gesteigert werden.

Es gilt im BS2000, dass der Heapspeicher immer von Anfang an in seiner Maximalgröße vom System angefordert wird und in dieser Größe auch immer reserviert bleibt. Mit der Option *-Xms* wird lediglich gesteuert, wie viel vom Heapspeicher aktuell verwendet werden soll. Je kleiner dieser Bereich ist, desto schneller geht die Garbage Collection, da nur der aktuell verwendete Bereich durchsucht werden muss. Andererseits kann es sein, dass die Garbage Collection unnötig oft aufgerufen werden muss, wenn im aktuell verwendeten Bereich für neue Objekte nur noch wenig Platz ist.

Für diese Optionen sind von der Originalbeschreibung abweichende Minimal- und Standardwerte festgelegt:

-Xsssize

Minimalwert: 512K
Standardwert: 1M

-Xmssize

Minimalwert: 1M
Standardwert: 3.5M

-Xmxsize

Minimalwert: 1M
Standardwert: 64M

Der angegebene Wert wird intern auf das nächste Vielfache von 2M aufgerundet.

8.4 native2ascii

Dieses Kommando konvertiert eine Datei aus einem beliebigen Codeset in den Codeset US-ASCII (7-Bit ASCII) und umgekehrt.

Syntax

native2ascii [Optionen ...] [Eingabedatei [Ausgabedatei]]

Optionen ...

Eine oder mehrere Kommandozeilenoptionen, durch Leerzeichen getrennt.

Eingabedatei

Datei, die konvertiert werden soll. Wird *Eingabedatei* nicht angegeben, wird die Eingabe auf *stdin* erwartet.

Ausgabedatei

Zieldatei für die Konvertierung. Wird *Ausgabedatei* nicht angegeben, erfolgt die Ausgabe auf *stdout*.

Ausgabedatei und *Eingabedatei* dürfen auch gleich sein.

Beschreibung

Das Kommando *native2ascii* konvertiert Text, der in einem beliebigen Codeset (z.B OSD_EBCDIC_DF04_1) vorliegt, in den US-ASCII (7-Bit ASCII), wobei dort nicht darstellbare Zeichen in der portablen Unicode-Darstellung (\uxxxx) ausgedrückt werden. Die umgekehrte Konvertierung ist ebenfalls möglich. Die portable Unicode-Darstellung wird z.B. beim Laden von Property-Files ausgewertet.

Werden Property-Files in JAR-Archiven abgelegt, so müssen sie im Codeset ISO8859-1 vorliegen. Das gleiche gilt für die Manifest-Files oder andere Texte. Dieses Kommando ermöglicht es, die entsprechenden Dateien darauf vorzubereiten, da der Codeset US-ASCII vollständig in ISO8859-1 enthalten ist.

Ab JENV V1.4A müssen Policy-Files, die von der Standard Policy-Implementierung verwendet werden, im UTF-8 Codeset codiert sein. Für die Konvertierung kann *native2ascii* verwendet werden, da der UTF-8 Codeset in den ersten 127 Zeichen mit US-ASCII übereinstimmt.

Optionen

-encoding *Zeichensatz*

Gibt den Zeichensatz an, aus dem bzw. in den das Kommando konvertiert. Ist die Option nicht angegeben, wird der über die System-Property *file.encoding* eingestellte Wert verwendet. Ab JENV V1.2A ist der Standardwert für diese System-Property OSD_EBCDIC_DF04_1. Zulässige Werte können der Spezifikation „Supported Encodings“ [14] entnommen werden. Die ab JENV V1.2A zusätzlich unterstützten Zeichensätze sind im Abschnitt „Codesets" beschrieben.

-reverse

Die Konvertierung wird in umgekehrter Richtung ausgeführt: Ein Text, der im Zeichensatz US-ASCII vorliegt, wird in den mit *-encoding* angegebenen Zeichensatz konvertiert. In der Eingabe vorkommende portable

Unicode-Darstellungen (\uxxxx) werden dabei interpretiert. Zeichen, die im Ausgabe-Zeichensatz nicht darstellbar sind, werden dort in portabler Unicode-Darstellung ausgegeben.

-J *javaoption*

Übergibt *javaoption* an die JVM, wobei *javaoption* eine der Optionen ist, die für *java* beschrieben sind.

8.5 jconsole

Im BS2000 wird im Gegensatz zu *jconsole* auf Solaris die Verwendung einer Prozessid (pid) zum Aufbau einer Verbindung mit einer Java-Anwendung nicht unterstützt.

8.6 jdb

jdb funktioniert nicht, wenn die Standardeingabe mit einem BS2000-Blockterminal verbunden ist, daher wird *jdb* in diesem Fall mit einer Fehlermeldung beendet.

9 Anhang: Kompatibilität zu Vorgängerversionen und Migration

JENV V9.0A ist eine Implementierung der „Java Platform, Standard Edition“ (Java SE™) für BS2000, basierend auf OpenJDK 9.

Mit OpenJDK 9 wurde in Java ein Modulkonzept eingeführt. Aufgrund dieses Modulkonzepts kann weder Source- noch Binärkompatibilität zu den Vorgängerversionen garantiert werden. Unter folgendem Link https://docs.oracle.com/javase/9/migrate/toc.htm ist eine ausführliche Migrationsanleitung verfügbar.

9.1 Inkompatibilitäten

Es sind keine BS2000-spezifischen Inkompatibilitäten bekannt.

10 Literatur

Die Handbücher finden Sie im Internet unter http://manuals.ts.fujitsu.com. Handbücher, die mit einer Bestellnummer angezeigt werden, können Sie auch in gedruckter Form bestellen.

[1] **POSIX** (BS2000)
POSIX, Grundlagen für Anwender und Systemverwalter
Benutzerhandbuch

[2] CRTE
C-Bibliotheksfunktionen für POSIX-Anwendungen
Referenzhandbuch

[3] CRTE
Common RunTime Environment
Benutzerhandbuch

[4] **C/C++** (BS2000)
C/C++-Compiler
Benutzerhandbuch

[5] **COBOL2000** (BS2000)
COBOL-Compiler
Benutzerhandbuch

[6] **COBOL2000** (BS2000)
COBOL-Compiler
Sprachbeschreibung

[7] **SDF-P** (BS2000)
Programmieren in der Kommandosprache
Benutzerhandbuch

[8] BS2000 OSD/BC
Einführung in das DVS
Benutzerhandbuch

10.1 Texte zu Java

Die folgenden weiterführenden Texte finden Sie im Internet, hauptsächlich auf den Web-Seiten von Oracle America Inc.:

Alle nachfolgenden Links waren bei Redaktionsschluss gültig. Es kann jedoch keine Gewähr für deren zukünftige Gültigkeit übernommen werden. Die Informationen in diesem Handbuch haben stets Vorrang vor denen im Internet.

[9] Java Platform Standard Edition 9 Documentation
https://docs.oracle.com/javase/9/index.html

[10] The Java™ Language and Virtual Machine Specifications
http://docs.oracle.com/javase/specs/

[11] JDK Tools and Utilities
http://docs.oracle.com/javase/9/docs/technotes/tools/index.html

[12] The Java™ Platform, Standard Edition 9 API Specification
http://docs.oracle.com/javase/9/docs/api/

[13] Java™ Native Interface
https://docs.oracle.com/javase/9/docs/specs/jni/intro.html

[14] Supported Encodings
https://docs.oracle.com/javase/9/intl/supported-encodings.htm

10.2 Weiterführende Literatur

[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