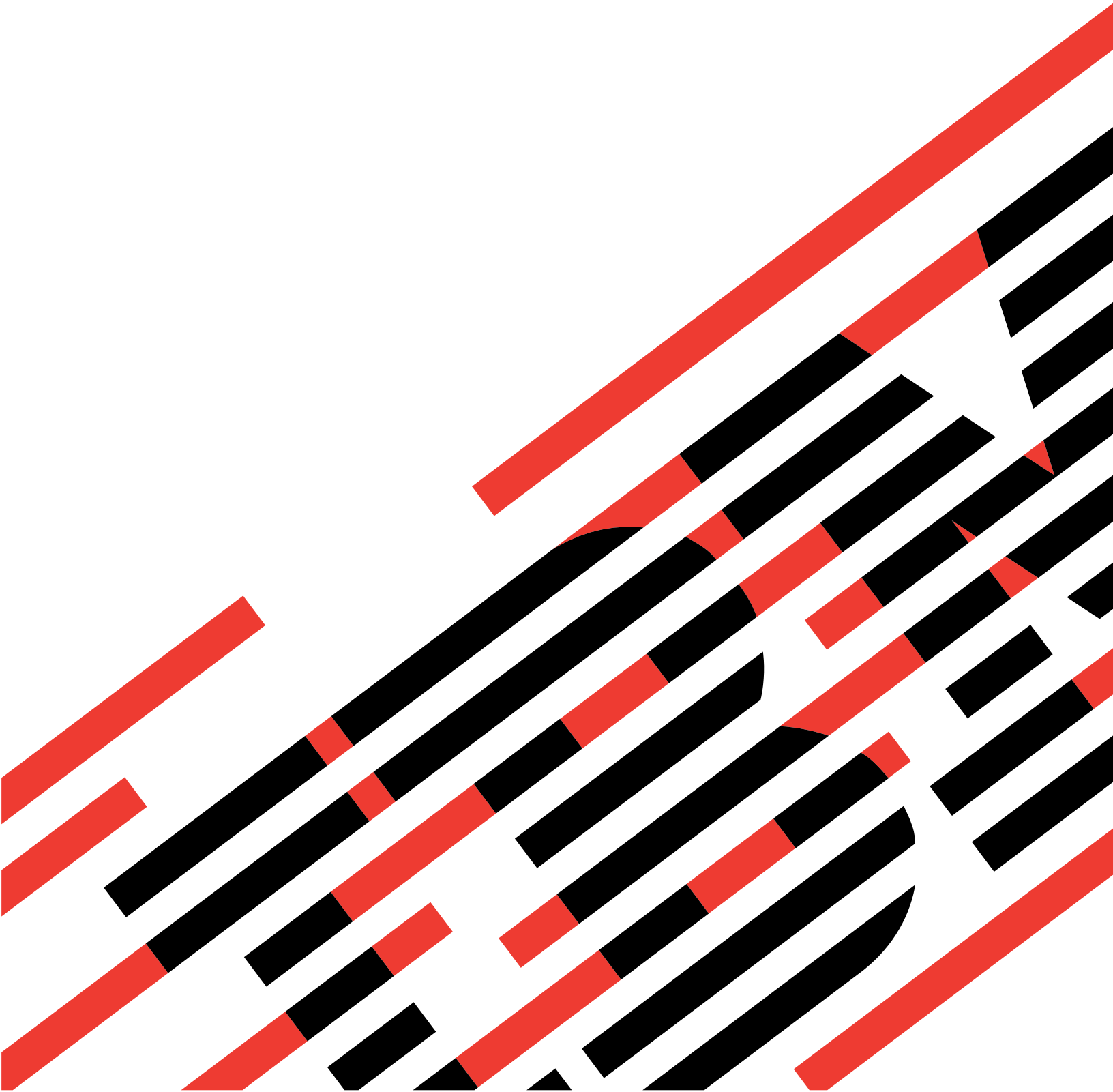
IBM

eserver

iSeries

IBM Developer Kit para Java

Versión 5 Release 3



IBM

@server

iSeries

IBM Developer Kit para Java

Versión 5 Release 3

Nota

Antes de utilizar esta información y el producto al que da soporte, lea la información de “Avisos”, en la página 403.

Novena edición (agosto de 2005)

Esta edición es aplicable a la versión 5, release 3, modificación 0 de IBM Developer Kit para Java (producto número 5722-JV1) y a todos los releases y modificaciones subsiguientes hasta que se indique lo contrario en nuevas ediciones. Esta versión no funciona en todos los modelos RISC (reduced instruction set computer) ni en los modelos CISC.

Contenido

IBM Developer Kit para Java 1
Novedades de V5R3 para IBM Developer Kit para Java 2
Cómo visualizar las novedades o cambios . . . 4
Imprimir este tema 4
Instalar y configurar IBM Developer Kit para Java . . 4
Instalar IBM Developer Kit para Java 5
Instalar un programa bajo licencia con el mandato Restaurar programa bajo licencia . . 5
Soporte para varios Java 2 Software Development Kits 6
Instalar ampliaciones de IBM Developer Kit para Java 7
Bajar e instalar paquetes Java 7
Ejecutar el primer programa Java Hello World . . 9
Correlacionar una unidad de red con el servidor iSeries 10
Crear un directorio en el servidor iSeries . . . 10
Crear, compilar y ejecutar un programa Java HelloWorld 10
Crear y editar archivos fuente Java 12
Con iSeries Access para Windows 12
En una estación de trabajo 12
Con EDTF 12
Con el programa de utilidad para entrada del fuente 12
Personalizar el servidor iSeries para IBM Developer Kit para Java 12
Vía de acceso de clases Java 13
Propiedades del sistema Java 15
Archivo SystemDefault.properties 15
Ejemplo: archivo SystemDefault.properties 16
Lista de propiedades del sistema Java . . . 16
Internacionalización 23
Configuración del huso horario 24
QUTCOFFSET y user.timezone 24
LOCALE 25
Codificaciones de caracteres de Java 26
Valores de file.encoding y CCSID de iSeries 26
Valores de file.encoding por omisión . . . 34
Ejemplos: Crear un programa Java internacionalizado 35
Compatibilidad entre releases 36
Acceso a base de datos con IBM Developer Kit para Java 36
Acceder a la base de datos de iSeries con el controlador JDBC de IBM Developer Kit para Java 36
Iniciación a JDBC 38
Tipos de controladores JDBC 39
Requisitos de JDBC 40
Ejercicio de aprendizaje de JDBC 41
Utilizar JNDI para los ejemplos 43
Conexiones 43
DriverManager 44
Propiedades de conexión 47
Utilizar DataSources con UDBDataSource 56
Propiedades de DataSource 58
Otras implementaciones de DataSource . . 64
Propiedades de la JVM para JDBC 65
Interfaz DatabaseMetaData para IBM Developer Kit para Java 66
Crear un objeto DatabaseMetaData . . . 66
Recuperar información general 67
Determinar el soporte de característica . . 67
Límites de origen de datos 67
Objetos SQL y sus atributos 67
Soporte de transacción 68
Cambios en JDBC 3.0 68
Excepciones 68
SQLException 69
SQLWarning 70
DataTruncation 70
Truncamiento silencioso 73
Transacciones 73
Modalidad de compromiso automático . . 74
Niveles de aislamiento de las transacciones 74
Puntos de salvar 76
Transacciones distribuidas 77
Transacciones con JTA 77
Utilizar el soporte de UDBXADataSource para agrupación y transacciones distribuidas 78
Propiedades de XADataSource 78
ResultSets y transacciones 78
Multiplexado 79
Compromiso de dos fases y anotación de transacciones 79
Tipos de sentencia 80
Sentencias 80
PreparedStatements 82
Crear y utilizar PreparedStatements . . . 82
Procesar PreparedStatements 84
CallableStatements 84
Procesar CallableStatements 87
ResultSets 88
Características de ResultSet 88
Movimiento de cursores 90
Recuperar datos de ResultSet 91
Cambiar ResultSets 92
Crear ResultSets 93
Agrupación de objetos JDBC 93
Utilizar el soporte de DataSource para la agrupación de objetos 94
Propiedades de ConnectionPoolDataSource 95
Agrupación de sentencias basada en DataSource 97
Crear su propia agrupación de conexiones 97
Actualizaciones por lotes 99
Actualización por lotes de Statement . . 100
Actualización por lotes de PreparedStatement 101

BatchUpdateException 101
Soporte de inserción por bloques 102
Tipos de datos avanzados 103
Tipos distintivos (distinct) 103
Objetos grandes 103
Tipos de datos SQL3 no soportados . . . 104
Escribir código que utilice BLOB 104
Escribir código que utilice CLOB 105
Escribir código que utilice Enlaces de datos (Datalinks) 105
RowSets 105
Características de RowSet 106
DB2CachedRowSet 107
Utilizar DB2CachedRowSet. 107
Crear y llenar un DB2CachedRowSet . . 108
Acceso a datos de DB2CachedRowSet y manipulación de cursores 112
Cambiar datos de DB2CachedRowSet y reflejar de nuevo los cambios en el origen de datos 116
Otras características de DB2CachedRowSet 120
DB2JdbcRowSet 125
Eventos DB2JdbcRowSet. 127
Consejos sobre el rendimiento del controlador JDBC de IBM Developer Kit para Java 129
Acceder a bases de datos mediante el soporte SQLJ DB2 de IBM Developer Kit para Java . . 132
Puesta a punto de SQLJ 132
Herramientas SQLJ 132
Restricciones de SQLJ DB2 133
Perfiles de lenguaje de consulta estructurada para Java 133
Conversor de lenguaje de consulta estructurada para Java (SQLJ) (sqlj) 133
Precompilar sentencias SQL en un perfil mediante el personalizador de perfiles SQLJ DB2, db2profc 133
Imprimir el contenido de los perfiles SQLJ de DB2 (db2profp y profp) 137
Instalador de auditores de perfiles SQLJ (profdb) 137
Convertir una instancia de perfil serializado a formato de clase Java mediante la herramienta de conversión de perfiles SQLJ (profconv) 138
Intercalar sentencias SQL en la aplicación Java 138
Variables de lenguaje principal del lenguaje de consulta estructurada para Java 139
Compilar y ejecutar programas SQLJ . . . 139
Rutinas SQL Java 140
Utilizar rutinas SQL Java 141
Procedimientos almacenados Java 143
Estilo de parámetro JAVA 143
Estilo de parámetro DB2GENERAL . . . 145
Restricciones de procedimientos almacenados Java 146
Funciones escalares Java definidas por usuario 147
Estilo de parámetro Java. 147
Estilo de parámetro DB2GENERAL . . . 148
Restricciones de funciones definidas por usuario Java 151
Funciones de tabla definidas por usuario Java 151
Procedimientos SQLJ que manipulan archivos JAR 153
SQLJ.INSTALL_JAR 153
SQLJ.REMOVE_JAR 154
SQLJ.REPLACE_JAR 155
SQLJ.UPDATEJARINFO 156
SQLJ.RECOVERJAR 156
SQLJ.REFRESH_CLASSES 157
Convenciones de pase de parámetros para procedimientos almacenados y UDF Java . . 157
Java con otros lenguajes de programación 158
Utilizar la interfaz nativa Java para métodos nativos 159
API de invocación Java 162
Funciones de la API de invocación . . . 162
Soporte para varias máquinas virtuales Java 164
Métodos nativos Java y consideraciones acerca de las hebras 164
Los métodos nativos y la interfaz nativa Java (JNI) 165
Las series en los métodos nativos. 166
Series literales en métodos nativos . . . 166
Convertir series dinámicas a y desde EBCDIC, Unicode y UTF-8 167
Métodos nativos IBM OS/400 PASE para Java 167
Variables de entorno Java de OS/400 PASE 168
Ejemplos: variables de entorno para el ejemplo de IBM OS/400 PASE. 169
Utilizar QIBM_JAVA_PASE_CHILD_STARTUP . 169
Utilizar QIBM_JAVA_PASE_ALLOW_PREV . . . 170
Códigos de error Java de OS/400 PASE . . 171
Errores de arranque 171
Errores de ejecución 171
Gestionar bibliotecas de métodos nativos . . 172
Convenios de denominación de bibliotecas Java de OS/400 PASE y AIX 172
Orden de búsqueda de bibliotecas Java 172
Métodos nativos del modelo de almacenamiento en teraespacio para Java. 173
Crear métodos nativos de teraespacio . . . 174
Crear programas de servicio de teraespacio que utilicen métodos nativos 174
Utilizar las API de invocación de Java con métodos nativos de teraespacio 175
Comparación entre el entorno de lenguajes integrados y Java 175
Utilizar java.lang.Runtime.exec() 175
Procesar distintos tipos de mandatos . . . 176
propiedad del sistema os400.runtime.exec 176
Ejemplos: Llamar a mandatos con java.lang.Runtime.exec() 177
Comunicaciones entre procesos 178

Utilizar sockets para la comunicación entre procesos 178
Utilizar corrientes de entrada y de salida para la comunicación entre procesos. . . . 178
Plataforma Java 178
Applets y aplicaciones Java. 179
Máquina virtual Java 180
Entorno de ejecución Java 180
Intérprete de Java 181
Archivos JAR y de clase Java 182
Hebras Java 182
Sun Microsystems, Inc. Java Development Kit 183
Paquetes Java 183
Herramientas Java. 184
Temas avanzados 184
Clases, paquetes y directorios Java 184
Los archivos del sistema de archivos integrado 186
Autorizaciones de archivo Java en el sistema de archivos integrado. 186
Ejecutar Java en un trabajo de proceso por lotes 187
Ejecutar la aplicación Java en un sistema principal que no tiene una interfaz gráfica de usuario . . . 187
NAWT (Native Abstract Windowing Toolkit) 187
Niveles de soporte de NAWT 189
NAWT y J2SDK, versión 1.3 189
NAWT y J2SDK, versión 1.4 189
Instalar y utilizar Native Abstract Windowing Toolkit 189
Instalar y utilizar NAWT 189
NAWT y OS/400 PASE 189
Consejos para la utilización de VNC. . . . 189
Iniciar un servidor de pantalla VNC desde un programa CL 189
Detener un servidor de pantalla VNC desde un programa CL 190
Buscar servidores de pantalla VNC en ejecución 190
Consejos para el uso de NAWT con WebSphere Application Server. 190
Garantizar las comunicaciones seguras 190
Utilizar la comprobación de autorización X 191
Seguridad Java 192
El modelo de seguridad Java 193
Ampliación de criptografía Java 193
Java Secure Socket Extension 194
Utilizar SSL (JSSE, versión 1.0.8) 195
Preparar el servidor iSeries para el soporte de capa de sockets segura 196
Productos Cryptographic Access Provider 196
Cambiar el código Java para que utilice fábricas de sockets. 197
Cambiar el código Java para que utilice la capa de sockets segura 197
Seleccionar un certificado digital 198
Utilizar el certificado digital al ejecutar la aplicación Java 198
Utilizar Java Secure Socket Extension, versión 1.4 199
Configurar el servidor iSeries para dar soporte a **JSSE** 200
Proveedores JSSE 200
Propiedades de seguridad de JSSE . . . 201
Propiedades del sistema Java de JSSE . . 202
Utilizar el proveedor iSeries JSSE nativo 204
Ejemplos: IBM Java Secure Sockets Extension. 205
Servicio de autenticación y autorización Java 206
Preparar y configurar un servidor iSeries para el servicio de autenticación y autorización Java 206
Ejemplos del servicio de autenticación y autorización Java 208
IBM Java Generic Security Service (JGSS) . . . 208
Conceptos de JGSS 210
Principales y credenciales 210
Establecimiento del contexto 211
Protección e intercambio de mensajes . . 211
Borrado y liberación de recursos 211
Mecanismos de seguridad 211
Configurar el servidor iSeries para utilizar IBM JGSS. 212
Configurar el servidor iSeries para utilizar JGSS con J2SDK, versión 1.3 212
Configurar JGSS para emplear el proveedor iSeries JGSS nativo 212
Configurar el servidor iSeries para utilizar **JGSS** con J2SDK, versión 1.4 213
Proveedores JGSS 214
Utilizar un gestor de seguridad 214
Permisos de JVM 214
Comprobaciones de permisos de JAAS 215
Ejecutar aplicaciones IBM JGSS 216
Obtener credenciales de Kerberos y crear claves secretas 217
Herramientas Kinit y Ktab 217
Interfaz de inicio de sesión de Kerberos JAAS 218
Archivos de configuración y política. . . 220
Desarrollar aplicaciones IBM JGSS 222
Procedimiento de programación de aplicaciones IBM JGSS 223
Crear un GSSManager 224
Crear un GSSName 224
Crear un GSSCredential 224
Crear GSSContext 225
Solicitar servicios de seguridad opcionales 225
Establecer el contexto. 226
Utilizar los servicios por mensaje. . . . 227
Enviar mensajes 227
Recibir mensajes 228
Suprimir el contexto 228
Utilizar JAAS con la aplicación JGSS . . 229
Depuración 229
Clase de depuración de JGSS 230
Ejemplos: IBM Java Generic Security Service (JGSS) 230
Descripción de los programas de ejemplo 230
Ver los ejemplos de IBM JGSS 231
Ejemplos: bajar y visualizar información de javadoc para los ejemplos de IBM JGSS 231

Ejemplos: bajar y ejecutar los programas de ejemplo 232
Ejemplos: bajar los ejemplos de IBM JGSS 232
Ejemplos: prepararse para ejecutar los programas de ejemplo 233
Ejemplos: ejecutar los programas de ejemplo 233
Información de consulta de javadoc de IBM JGSS 234
Ajustar el rendimiento de un programa Java con IBM Developer Kit para Java 234
Consideraciones sobre el rendimiento de Java 235
Crear programas Java optimizados 235
Utilizar el compilador Just-In-Time 236
Utilizar antememorias para cargadores de clases de usuario 236
Seleccionar la modalidad que debe utilizarse al ejecutar un programa Java 238
Intérprete de Java 240
Compilación estática 241
Consideraciones sobre el rendimiento de la compilación estática Java. 241
Compilador Just-In-Time 242
Comparación entre el compilador Just-In-Time y el proceso directo 242
Recogida de basura de Java 243
Recogida de basura avanzada de IBM Developer Kit para Java 243
Consideraciones sobre el rendimiento de la recogida de basura de Java 243
Consideraciones sobre el rendimiento de la llamada a métodos nativos Java 244
Consideraciones sobre el rendimiento de la incorporación de métodos Java 244
Consideraciones sobre el rendimiento de las excepciones Java 244
Herramientas de rendimiento de rastreo de llamadas Java 245
Herramientas de rendimiento de perfilado Java 245
Interfaz del perfilador de la máquina virtual Java 245
Recoger datos de rendimiento Java 246
La herramienta Performance Data Collector 247
La herramienta Java Performance Data Converter 247
Ejecutar Java Performance Data Converter 248
Mandatos y herramientas de IBM Developer Kit para Java 248
Herramientas Java soportadas por IBM Developer Kit para Java 248
Herramientas Java. 249
La herramienta ajar Java. 250
La herramienta appletviewer Java . . . 250
La herramienta extcheck Java 251
La herramienta idlj Java 251
La herramienta jar Java 251
La herramienta jarsigner Java 251
La herramienta javac Java 252
La herramienta javadoc Java 252
Cómo extraer archivos de ejemplo . . . 252
Herramientas Java. 252
La herramienta javah Java 253
La herramienta javap Java 253
La herramienta keytool Java 254
La herramienta native2ascii Java 254
La herramienta orbd Java 254
La herramienta policytool Java 254
La herramienta rmic Java 255
La herramienta rmid Java 255
La herramienta rmiregistry Java 255
La herramienta serialver Java 255
servertool Java 255
La herramienta tnameserv Java 256
El mandato java de Qshell 256
Mandatos CL soportados por Java 257
Consideraciones sobre el uso del mandato ANZJVM. 257
Depurar programas Java que se ejecutan en el servidor 258
Depurar programas Java desde una línea de mandatos de OS/400. 259
Depurar un programa Java 259
Depurar programas Java mediante la opción *DEBUG 260
Pantallas iniciales de depuración de programas Java. 260
Establecer puntos de interrupción . . . 262
Recorrer los programas Java que deben depurarse 263
Evaluar variables en programas Java . . 263
Depurar programas Java y programas de métodos nativos 264
Depurar un programa Java desde otra pantalla 265
Variable de entorno QIBM_CHILD_JOB_SNDINQMSG . . . 266
Depurar clases Java cargadas mediante un cargador de clases personalizadas 266
Depurar servlets 267
Arquitectura del depurador de la plataforma Java 267
Interfaz de depuración de la máquina virtual Java 268
Java Debug Wire Protocol 268
Interfaz de depuración Java 269
Depuración a máxima velocidad 269
Localizar fugas de memoria 269
Ejemplos de código para IBM Developer Kit para Java 270
Ejemplo: internacionalización de las fechas con la clase java.util.DateFormat 272
Ejemplo: internacionalización de las presentaciones numéricas con la clase java.util.NumberFormat 273
Ejemplo: internacionalización de los datos específicos de entorno nacional con la clase java.util.ResourceBundle. 274
Ejemplo: propiedad Access 275
Ejemplo: BLOB 277
Ejemplo: interfaz CallableStatement para IBM Developer Kit para Java 279

Ejemplo: eliminar valores de una tabla mediante el cursor de otra sentencia 279
Ejemplo: CLOB 281
Ejemplo: crear un UDBDataSource y enlazarlo con JNDI 283
Ejemplo: crear UDBDataSource y obtener un ID de usuario y una contraseña 283
Ejemplo: crear un UDBDataSourceBind y establecer las propiedades de DataSource . . . 284
Ejemplo: interfaz DatabaseMetaData para IBM Developer Kit para Java 285
Ejemplo: Datalink 285
Ejemplo: tipos Distinct 287
Ejemplo: intercalar sentencias SQL en la aplicación Java 288
Ejemplo: finalizar una transacción 291
Ejemplo: ID de usuario y contraseña no válidos 293
Ejemplo: JDBC 294
Ejemplo: varias conexiones que funcionan en una transacción. 298
Ejemplo: obtener un contexto inicial antes de enlazar UDBDataSource 300
Ejemplo: ParameterMetaData 301
Ejemplo: cambiar valores con una sentencia mediante el cursor de otra sentencia. 302
Ejemplo: interfaz ResultSet para IBM Developer Kit para Java 304
Ejemplo: sensibilidad de ResultSet 306
Ejemplo: ResultSets sensibles e insensibles. . . 308
Ejemplo: configurar una agrupación de conexiones con UDBDataSource y UDBConnectionPoolDataSource 310
Ejemplo: SQLException 310
Ejemplo: suspender y reanudar una transacción 311
Ejemplo: ResultSets suspendidos 314
Ejemplo: probar el rendimiento de una agrupación de conexiones 316
Ejemplo: probar el rendimiento de dos DataSources 317
Ejemplo: actualizar BLOB 318
Ejemplo: actualizar CLOB 319
Ejemplo: utilizar una conexión con varias transacciones 320
Ejemplo: utilizar BLOB 322
Ejemplo: utilizar CLOB 323
Ejemplo: utilizar JTA para manejar una transacción 324
Ejemplo: utilizar ResultSets de metadatos que tienen más de una columna 326
Ejemplo: Utilizar JDBC nativo y JDBC de IBM Toolbox para Java de forma concurrente . . . 328
Ejemplo: utilizar PreparedStatement para obtener un ResultSet 330
Ejemplo: utilizar el método executeUpdate de un objeto Statement 332
Ejemplos: HelloWorld para JAAS 333
HelloWorld.java 334
HWLoginModule.java 337
HWPrincipal.java 342
Ejemplo: SampleThreadSubjectLogin de JAAS 343
Ejemplo: programa cliente no JAAS de IBM JGSS 353
Ejemplo: programa servidor no JAAS de IBM JGSS 361
Ejemplo: programa cliente habilitado para JAAS de IBM JGSS 372
Ejemplo: programa servidor habilitado para JAAS de IBM JGSS 374
Ejemplo: llamar a un programa CL con java.lang.Runtime.exec() 375
Ejemplo: llamar a un mandato CL con java.lang.Runtime.exec() 376
Ejemplo: Clase para llamar a un mandato CL 377
Ejemplo: llamar a otro programa Java con java.lang.Runtime.exec() 377
Ejemplo: llamar a Java desde C 378
Ejemplo: llamar a Java desde RPG 378
Ejemplo: utilizar corrientes de entrada y de salida para la comunicación entre procesos . . 379
Ejemplo: API de invocación Java 380
Ejemplo: Utilizar la API de invocación Java 381
Ejemplo: método nativo IBM OS/400 PASE para Java 383
Ejecutar el ejemplo de método nativo OS/400 PASE para Java. 383
Ejemplos: utilizar la interfaz nativa Java para métodos nativos 383
Ejemplo: utilizar sockets para la comunicación entre procesos 388
Ejemplo: ejecutar Java Performance Data Converter 391
Ejemplos: cambiar el código Java para que utilice fábricas de sockets de cliente 392
Ejemplos: cambiar el código Java para que utilice fábricas de sockets de servidor 393
Ejemplos: cambiar el cliente Java para que utilice la capa de sockets segura 395
Ejemplos: cambiar el servidor Java para que utilice la capa de sockets segura 396
Resolución de problemas de IBM Developer Kit para Java 398
Limitaciones. 399
Buscar las anotaciones de trabajo para el análisis de problemas Java. 399
Recoger datos para el análisis de problemas de Java 400
Obtener soporte para IBM Developer Kit para Java 401
Información relacionada para IBM Developer Kit para Java 401

Apéndice. Avisos 403
Marcas registradas. 405
Términos y condiciones para bajar e imprimir publicaciones 405
Información de limitación de responsabilidad sobre el código 406

IBM Developer Kit para Java

IBM Developer Kit para Java está optimizado para el uso en un entorno de servidor iSeries. Utiliza la compatibilidad de la programación y las interfaces de usuario Java, para que el usuario pueda desarrollar sus propias aplicaciones para el servidor iSeries.

IBM Developer Kit para Java le permite crear y ejecutar programas Java en el servidor iSeries. IBM Developer Kit para Java es una implementación compatible de Sun Microsystems, Inc. Java Technology, y por consiguiente debe estar familiarizado con su documentación de JDK (Java Development Kit). Para facilitarle el trabajo con esa información y con la nuestra, le proporcionamos enlaces con la información de Sun Microsystems, Inc.

Si, por alguna razón, nuestros enlaces con la documentación de Java Development Kit de Sun Microsystems, Inc. no funcionan, consulte la documentación HTML correspondiente a la información que necesita. Puede encontrar esta información en la World Wide Web en The Source for Java Technology java.sun.com

.

Si desea obtener más detalles sobre cómo se utiliza IBM Developer Kit para Java, seleccione cualquiera de estos temas:

Novedades de V5R3
Resalta las actualizaciones más recientes de la información y los productos.

Imprimir este tema
Proporciona detalles sobre cómo bajar un archivo PDF imprimible o un paquete con formato zip de los archivos HTML de IBM Developer Kit para Java.

Instalar y configurar
Proporciona información acerca de la instalación, la configuración, cómo crear y ejecutar programas Java Hello World sencillos, bajar e instalar y compatibilidad entre releases.

Personalización
Proporciona instrucciones para personalizar la configuración del huso horario, las propiedades del sistema y la vía de acceso de clases en el servidor.

Compatibilidad
Proporciona información acerca de la compatibilidad de los archivos de clase Java entre releases.

Acceso a base de datos
Describe cómo IBM Developer Kit para Java permite a los programas Java acceder a los archivos de base de datos de iSeries.

Java con otros lenguajes de programación
Muestra cómo llamar a código escrito en lenguajes distintos de Java mediante JNDI (Java Native Interface), java.lang.Runtime.exec(), comunicación entre procesos y la API de invocación Java.

Plataforma Java
Describe el entorno para desarrollar y gestionar applets y aplicaciones Java, que consta del lenguaje Java, los paquetes Java y la máquina virtual Java.

Temas avanzados
Proporciona instrucciones para ejecutar Java en un trabajo de proceso por lotes y describe las autorizaciones de archivo Java necesarias en el sistema de archivos integrado para visualizar, ejecutar o depurar un programa Java.

Ejecutar en un sistema principal sin una GUI
Contiene información para configurar y ejecutar programas Java con NAWT (Native Abstract Window Toolkit).

Seguridad
Proporciona detalles sobre la autorización adoptada y describe cómo se puede utilizar SSL para hacer que las corrientes de sockets sean seguras en la aplicación Java.

Rendimiento
Proporciona información sobre la manera de ajustar el rendimiento de Java.

Mandatos y herramientas
Proporciona detalles sobre la manera de utilizar los mandatos y las herramientas Java.

Depurar
Explica cómo depurar los programas Java.

Ejemplos de código
Enlaza directamente con todos los ejemplos de código de esta información.

Resolución de problemas
Le muestra cómo localizar los archivos de anotaciones del trabajo y a recoger datos para analizar los programas Java. En este tema también se facilita información sobre los arreglos temporales del programa (PTF) y la manera de obtener soporte técnico para IBM Developer Kit para Java.

Información relacionada
Enlaza directamente con toda la información de consulta de Javadoc y API.

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

Novedades de V5R3 para IBM Developer Kit para Java

Este tema señala los cambios de IBM Developer Kit para Java en V5R3. Se señalan los cambios específicos de Java 2 Software Development Kit (J2SDK), Standard Edition, versión 1.4. Las actualizaciones posteriores al lanzamiento general de V5R3 aparecen al final de la lista siguiente.

Iniciación

- Se ha añadido nueva información de programa bajo licencia a Instalar IBM Developer Kit para Java.

- Soporte de múltiples JDK contiene información acerca de cada JDK soportado por IBM.
- Utilice Native Abstract Windowing Toolkit (NAWT) para obtener prestaciones gráficas completas con las aplicaciones y servlets Java.

Personalización
- Existen nuevas Propiedades del sistema, incluido el uso de la variable de entorno a nivel de trabajo QIBM_JAVA_PROPERTIES_FILE para señalar a un archivo de propiedades específico.
- Se ha añadido nueva información de Configuración de huso horario.
- Se han añadido nuevos valores File.encoding y los identificadores de juegos de caracteres codificados (CCSID) de iSeries con mayor coindicencia.

Acceso a base de datos
- Se han añadido nuevas propiedades de JVM para la sección JDBC.
- Se ha añadido el procedimiento SQLJ.REFRESH_CLASSES a la sección Procedimientos SQLJ.
- Se han añadido las secciones Procedimientos almacenados Java y Funciones escalares definidas por usuario Java.

Java con otros lenguajes de programación
- La máquina virtual Java (JVM) ahora da soporte al uso de métodos nativos del modelo de almacenamiento en teraespacio
- Se ha añadido nuevo soporte de la API de invocación Java.
- Se ha actualizado el uso de java.lang.Runtime.exec() para procesar distintas clases de mandatos.

Ejecutar en un sistema principal sin GUI
- Consulte el tema Native Abstract Windowing Toolkit (NAWT) para conocer las actualizaciones.

Seguridad
- Se ha añadido un nuevo Proveedor JSSE y nuevas propiedades

Rendimiento
- Consulte el tema Optimización para conocer las actualizaciones sobre el uso de antememoria para los cargadores de clases de usuario.

Mandatos y herramientas
- Se ha añadido la herramienta orbd Java.
- Se ha añadido la “servertool Java” en la página 255.

Depuración
- Consulte el tema Depuración para conocer las actualizaciones, incluida la depuración a máxima velocidad.

Ejemplos de código
- Se han añadido más ejemplos de código.

Imprimir este tema
- Imprimir este tema contiene un PDF de la información de IBM Developer Kit para Java.

Consulta
- Se ha añadido la sección Información relacionada de IBM Developer Kit para Java, que contiene información de consulta de Javadoc y API.

Cómo visualizar las novedades o cambios

En esta información se utiliza lo siguiente a modo de ayuda para indicar dónde se han realizado cambios técnicos:

- La imagen que señala el lugar en el que empieza la información nueva o cambiada.
- La imagen que señala el lugar en el que acaba la información nueva o cambiada.

Para buscar otras informaciones acerca de los novedades o cambios de este release, consulte el Memorándum para usuarios

.

Imprimir este tema

Para ver o bajar la versión PDF, seleccione IBM Developer Kit para Java (aproximadamente 2382 KB).

Guardar archivos PDF

Para guardar un PDF en la estación de trabajo con el fin de verlo o imprimirlo:

1. Con el botón derecho del ratón pulse sobre el PDF en el navegador (con el botón derecho, pulse el enlace anterior).
2. Pulse **Guardar objetivo como...** si está utilizando Internet Explorer. Pulse **Guardar enlace como...** si está utilizando Netscape Communicator.
3. Desplácese hasta el directorio en el que desea guardar el PDF.
4. Pulse **Guardar**.

Bajar Adobe Acrobat Reader

Si necesita Adobe Acrobat Reader para ver o imprimir estos PDF, puede bajar una copia desde el sitio Web de Adobe (www.adobe.com/products/acrobat/readstep.html)

.

Instalar y configurar IBM Developer Kit para Java

Si todavía no ha utilizado IBM Developer Kit para Java, siga estos pasos para instalarlo, configurarlo y practicar con la ejecución de un sencillo programa Java Hello World.

1. Si ya está familiarizado con la información de IBM Developer Kit para Java, consulte Novedades, donde hallará enlaces con la información y las actualizaciones más recientes del producto.
2. Instale IBM Developer Kit para Java.
3. Personalice el servidor iSeries.
4. Si esta información es nueva para usted y todavía no ha utilizado IBM Developer Kit para Java, consulte la sección Ejecutar el primer programa Java Hello World. En este tema se ilustran dos métodos de ejecución de un programa Java Hello World sencillo con IBM Developer Kit para Java. Es una forma práctica de ver si ha instalado correctamente IBM Developer Kit para Java.
5. Ahora ya está preparado para crear, compilar y ejecutar su propio programa Java Hello World. Si desea saber cuál es el procedimiento que debe seguir, consulte la sección Crear, compilar y ejecutar un programa Java Hello World.
6. Si está interesado en crear más aplicaciones Java propias, lea los temas siguientes:

- En la sección Crear y editar archivos fuente Java se muestran tres maneras diferentes de crear y editar archivos fuente Java.
- Bajar e instalar paquetes Java en un servidor iSeries sirve de ayuda para utilizar los paquetes Java de una forma más eficiente. En este tema se facilita información detallada referente a los paquetes con interfaz gráfica de usuario (GUI), el sistema de archivos integrado y la sensibilidad a mayúsculas y minúsculas, así como el manejo de archivos ZIP y JAR.
- Compatibilidad de release a release proporciona información acerca de la compatibilidad de un release a otro.

Instalar IBM Developer Kit para Java

Instalar IBM Developer Kit para Java le permite crear y ejecutar programas Java en el servidor iSeries.

Para la V5R3, se envía el programa bajo licencia 5722-JV1 con los CD del sistema, por lo que se preinstala JV1. Entre el mandato Ir a programa bajo licencia (GO LICPGM) y seleccione la opción 10 (Visualizar). Si no ve listado este programa bajo licencia, lleve a cabo los pasos siguientes:

1. Entre el mandato GO LICPGM en la línea de mandatos.
2. Seleccione la opción 11 (Instalar programa bajo licencia).
3. Elija la opción 1 (Instalar) para el programa bajo licencia (LP) 5722-JV1 *BASE y seleccione la opción que coincida con el programa Java Development Kit (JDK) que desea instalar. Si la opción que desea instalar no se visualiza en la lista, puede añadirla a la misma especificando la opción 1 (Instalar) en el campo de opción. Entre 5722JV1 en el campo programa bajo licencia y el número de opción en el campo opción de producto.

   Nota: se puede instalar más de una opción a la vez.

Una vez que haya instalado IBM Developer Kit para Java en el servidor iSeries, puede optar por personalizar el sistema.

Consulte la sección Ejecutar el primer programa Java Hello World para obtener información acerca de cómo empezar a trabajar con IBM Developer Kit para Java.

Instalar un programa bajo licencia con el mandato Restaurar programa bajo licencia

Los programas que aparecen en la pantalla *Instalar programas bajo licencia* son aquellos que están soportados por la instalación de LICPGM cuando el servidor era nuevo. Ocasionalmente, quedan disponibles programas nuevos que no aparecían en la lista como programas bajo licencia en el servidor. Si este es el caso del programa que desea instalar, debe utilizar el mandato Restaurar programa bajo licencia (RSTLICPGM) para instalarlo.

Para instalar un programa bajo licencia con el mandato Restaurar programa bajo licencia (RSTLICPGM), siga estos pasos:

1. Coloque la cinta o el CD-ROM que contiene el programa bajo licencia en la unidad adecuada.
2. En la línea de mandatos de iSeries, escriba:

   RSTLICPGM/p>

   y pulse la tecla Intro.

   Aparece la pantalla *Restaurar programa bajo licencia (RSTLICPGM)*.
3. En el campo *Producto*, escriba el número de ID del programa bajo licencia que desea instalar.
4. En el campo *Dispositivo*, especifique el dispositivo de instalación.

Nota: si instala desde una unidad de cintas, el ID de dispositivo está generalmente en el formato **TAP*xx***, donde ***xx*** es un número, como **01**.

5. Conserve los valores por omisión para los demás parámetros de la pantalla *Restaurar programa bajo licencia*. Pulse la tecla Intro.
6. Aparecen más parámetros. Conserve también estos valores por omisión. Pulse la tecla Intro. El programa empieza a instalarse.

Cuando el programa bajo licencia haya terminado de instalarse, aparecerá de nuevo la pantalla *Restaurar programa bajo licencia*.

Soporte para varios Java 2 Software Development Kits

El servidor iSeries da soporte a múltiples versiones de Java Development Kit (JDK) y Java 2 Software Development Kit (J2SDK), Standard Edition.

Nota: En esta documentación, dependiendo del contexto, el término JDK hace referencia a cualquier versión soportada de JDK y J2SDK. Normalmente, el contexto en el que aparece JDK incluye una referencia a la versión y número de release específicos.

El servidor iSeries da soporte al uso de múltiples JDK simultáneamente, pero solamente mediante múltiples máquinas virtuales Java. Cada una de las máquinas virtuales Java ejecuta el JDK que se haya especificado para ella.

Localice el JDK que usted utiliza, o que desea utilizar, y seleccione la opción correspondiente para instalarlo. Puede instalar más de un JDK a la vez. La propiedad java.version del sistema determina qué JDK hay que ejecutar. Una vez que esté en marcha una máquina virtual Java, el hecho de cambiar la propiedad java.version del sistema no afecta para nada al JDK.

Nota:

En la V5R3, las siguientes opciones ya no están disponibles: Opción 1 (JDK 1.1.6), Opción 2 (JDK 1.1.7), Opción 3 (JDK 1.2.2) y Opción 4 (JDK 1.1.8). La siguiente tabla lista los J2SDK soportados para este release.

Opción	JDK	java.home	java.version
5	1.3	/QIBM/ProdData/Java400/jdk13/	1.3
6	1.4	/QIBM/ProdData/Java400/jdk14/	1.4

El JDK por omisión elegido en este entorno de múltiples JDK depende de qué opciones de 5722-JV1 se han instalado. La tabla siguiente ofrece algunos ejemplos.

Instale	Entre	Resultado
Opción 5 (1.3)	`java Hello`	Se ejecuta J2SDK, Standard Edition, versión 1.3.
 Opción 6 (1.4)	`java Hello`	Se ejecuta J2SDK, Standard Edition, versión 1.4. 
Opción 5 (1.3) y Opción 6 (1.4)	`java Hello`	 Se ejecuta J2SDK, Standard Edition, versión 1.4. 

Nota: si instala un solo JDK, el JDK por omisión es el que ha instalado. Si instala más de un JDK, el orden de prioridad siguiente determina cuál es el JDK por omisión:

1. Opción 6 (1.4)
2. Opción 5 (1.3)

Instalar ampliaciones de IBM Developer Kit para Java

Las ampliaciones son paquetes de clases Java que pueden utilizarse para ampliar las funciones de la plataforma central. Las ampliaciones se empaquetan en uno o más archivos ZIP o JAR y se cargan en la máquina virtual Java mediante un cargador de clases de ampliación.

El mecanismo de ampliación permite que la máquina virtual Java utilice las clases de ampliación de la misma forma que la máquina virtual utiliza las clases del sistema. El mecanismo de ampliación también proporciona una manera de recuperar las ampliaciones a partir de los URL (localizadores universales de recursos) cuando aún no están instaladas en J2SDK, versión 1.2 o superior, o en Java 2 Runtime Environment, Standard Edition, versión 1.2 y posteriores.

Con el servidor iSeries se suministran algunos archivos JAR para ampliaciones. Si desea instalar una de estas ampliaciones, entre este mandato:

```
ADDLNK OBJ('/QIBM/ProdData/Java400/ext/extensionToInstall.jar')
     NEWLNK('/QIBM/UserData/Java400/ext/extensionToInstall.jar')
     LNKTYPE(*SYMBOLIC)
```

Donde `extensionToInstall.jar` es el nombre del archivo ZIP o JAR que contiene la ampliación que desea instalar.

Nota: los archivos JAR de ampliaciones no suministradas por IBM pueden colocarse en el directorio /QIBM/UserData/Java400/ext.

Al crear un enlace o añadir un archivo a una ampliación del directorio /QIBM/UserData/Java400/ext, la lista de archivos en la que busca el cargador de clases de ampliación pasa a ser la lista de *todas las máquinas virtuales Java que se ejecutan en el servidor iSeries*. Si no quiere que los cargadores de clases de ampliación correspondientes a las otras máquinas virtuales Java del servidor iSeries resulten afectados, pero aún así desea crear un enlace a una ampliación o bien instalar una ampliación no suministrada por IBM junto con el servidor iSeries, siga estos pasos:

1. Cree un directorio para instalar las ampliaciones.
   Utilice el mandato Crear directorio (MKDIR) desde la línea de mandatos de iSeries o el mandato `mkdir` desde el intérprete de Qshell.
2. Coloque el archivo JAR de ampliación en el directorio que ha creado.
3. Añada el directorio nuevo a la propiedad java.ext.dirs.
   Puede añadir el directorio nuevo a la propiedad java.ext.dirs utilizando el campo PROP del mandato JAVA desde la línea de mandatos de iSeries.

Si el nombre del directorio nuevo es /home/username/ext, el nombre del archivo de ampliación es extensionToInstall.jar y el nombre del programa Java es Hello, los mandatos que especifique deberán ser los siguientes:

```
MKDIR DIR('/home/username/ext')

CPY OBJ('/productA/extensionToInstall.jar') TODIR('/home/username/ext') o
copie el archivo en /home/username/ext utilizando FTP (protocolo de transferencia de archivos).

JAVA Hello PROP((java.ext.dirs '/home/username/ext'))
```

Bajar e instalar paquetes Java

Para bajar, instalar y utilizar paquetes Java de una forma más efectiva en un servidor iSeries, consulte las siguientes secciones:

- Paquetes con interfaces gráficas de usuario (page 8)
- La sensibilidad a las mayúsculas y minúsculas y el sistema de archivos integrado (page 8)
- Manejo de archivos ZIP y JAR (page 8)
- Infraestructura de ampliaciones Java (page 9)

Paquetes con interfaces gráficas de usuario

Los programas Java que se utilizan con una interfaz gráfica de usuario (GUI) requieren el uso de un dispositivo de presentación con posibilidades de visualización gráfica. Por ejemplo, se puede utilizar un PC, una estación de trabajo técnica o una máquina de red. Puede utilizar Native Abstract Windowing Toolkit (NAWT) para proporcionar a las aplicaciones y servlets Java la posibilidad de utilizar las funciones de gráficos de AWT (Abstract Windowing Toolkit) de Java 2 Software Development Kit (J2SDK), Standard Edition. Para obtener más información, consulte el apartado Native Abstract Windowing Toolkit (NAWT).

La sensibilidad a las mayúsculas y minúsculas y el sistema de archivos integrado

El sistema de archivos integrado proporciona sistemas de archivos sensibles a las mayúsculas y minúsculas y también otros no sensibles a las mayúsculas y minúsculas por lo que a los nombres de archivo se refiere. QOpenSys es un ejemplo de sistema de archivos sensible a las mayúsculas y minúsculas dentro del sistema de archivos integrado. El sistema de archivos root, '/', es un ejemplo de sistema de archivos no sensible a las mayúsculas y minúsculas. Para obtener más información, consulte el tema Sistema de archivos integrado.

Aunque un JAR o una clase puede encontrarse en un sistema de archivos no sensible a mayúsculas y minúsculas, Java sigue siendo un lenguaje sensible a mayúsculas y minúsculas. Mientras que `wrklnk '/home/Hello.class'` y `wrklnk '/home/hello.class'` generan los mismos resultados, `JAVA CLASS(Hello)` y `JAVA CLASS(hello)` llaman a clases distintas.

Manejo de archivos ZIP y JAR

Los archivos ZIP y JAR contienen un conjunto de clases Java. Cuando se utiliza el mandato Crear programa Java (CRTJVAPGM) en uno de estos archivos, se verifican las clases, estas se convierten a un formato máquina interno y, si se ha especificado así, se transforman en código máquina de iSeries. Los archivos ZIP y JAR pueden recibir el mismo trato que cualquier otro archivo de clase individual. Si se asocia un formato máquina interno a uno de estos archivos, dicho formato permanece asociado al archivo. En las ejecuciones futuras y con objeto de mejorar el rendimiento, se utilizará el formato máquina interno en lugar del archivo de clase. Si no está seguro de si existe un programa Java actual asociado al archivo de clase o al archivo JAR, utilice el mandato Visualizar programa Java (DSPJVAPGM) para visualizar información acerca del programa Java en el servidor iSeries.

En los releases anteriores de IBM Developer Kit para Java, era necesario volver a crear un programa Java si se cambiaba de algún modo el archivo JAR o ZIP, debido a que el programa Java conectado no hubiera podido utilizarse. Esto ya no es así. En muchos casos, si se cambia un archivo JAR o ZIP, el programa Java sigue siendo válido y no es necesario volver a crearlo. Si se efectúan cambios parciales, como al actualizar un solo archivo de clase dentro de un archivo JAR, solo es necesario volver a crear los archivos de clase afectados que se encuentran dentro del archivo JAR.

Los programas Java permanecen conectados al archivo JAR después de realizarse los cambios más habituales en el archivo JAR. Por ejemplo, estos programas Java permanecen conectados al archivo JAR después de:

- Cambiar o crear de nuevo un archivo JAR con la "La herramienta ajar Java" en la página 250.
- Cambiar o crear de nuevo un archivo JAR con la "La herramienta jar Java" en la página 251.
- Sustituir un archivo JAR con el mandato COPY de OS/400 o el programa de utilidad `cp` de Qshell.

Si accede a un archivo JAR del sistema de archivos integrado por medio de iSeries Access para Windows o desde una unidad correlacionada de un PC, estos programas Java permanecen conectados al archivo JAR después de:

- Arrastrar y soltar otro archivo JAR dentro del archivo JAR del sistema de archivos integrado existente.
- Cambiar o crear de nuevo el archivo JAR del sistema de archivos integrado con la "La herramienta jar Java" en la página 251.
- Sustituir el archivo JAR del sistema de archivos integrado utilizando el mandato copy de PC.

Cuando se cambia o sustituye un archivo JAR, el programa Java conectado a él ya no es actual.

Existe un único caso en el que los programas Java no permanecen conectados al archivo JAR. Los programas Java conectados se destruyen si se utiliza el protocolo de transferencia de archivos (FTP) para sustituir el archivo JAR. Esto ocurre, por ejemplo, si se utiliza el mandato `put` de FTP para sustituir el archivo JAR.

En el apartado Rendimiento de ejecución Java hallará información más detallada sobre las características de rendimiento de los archivos JAR.

Infraestructura de ampliaciones Java

En J2SDK, las ampliaciones son paquetes de clases Java que pueden utilizarse para ampliar las funciones de la plataforma central. Una ampliación o aplicación está empaquetada en uno o más archivos JAR. El mecanismo de ampliación permite que la máquina virtual Java utilice las clases de ampliación de la misma forma que la máquina virtual utiliza las clases del sistema. El mecanismo de ampliación también proporciona una forma de recuperar las ampliaciones a partir de los URL especificados cuando aún no están instaladas en J2SDK o en Java 2 Runtime Environment, Standard Edition.

En la sección Instalar ampliaciones para IBM Developer Kit para Java hallará información sobre cómo instalar las ampliaciones.

Ejecutar el primer programa Java Hello World

Para ejecutar el programa Java Hello World, puede hacerlo de cualquiera de las maneras siguientes:

1. Puede, simplemente, ejecutar el programa Java Hello World que se entrega con IBM Developer Kit para Java.

   Para ejecutar el programa que se incluye, lleve a cabo los siguientes pasos:

   a. Compruebe que está instalado el programa IBM Developer Kit para Java; para ello, entre el mandato Ir a programa bajo licencia (GO LICPGM). A continuación, seleccione la opción 10 (Visualizar programas bajo licencia instalados). Verifique que en la lista figuran como instalados el programa bajo licencia 5722-JV1 *BASE y al menos una de las opciones.

   b. Especifique `java Hello` en la línea de mandatos del menú principal de iSeries. Pulse Intro para ejecutar el programa Java Hello World.

   c. Si IBM Developer Kit para Java se ha instalado correctamente, aparecerá `Hello World` en la pantalla de la shell Java. Pulse F3 (Salir) o F12 (Salir) para volver a la pantalla de entrada de mandato.

   d. Si no se ejecuta la clase Hello World, compruebe si la instalación se ha realizado satisfactoriamente o consulte la sección Obtener soporte técnico para IBM Developer Kit para Java para obtener información de servicio.

2. También puede ejecutar su propio programa Java Hello. Si desea obtener más información sobre cómo puede crear su propio programa Java Hello, consulte la sección Crear, compilar y ejecutar un programa Java Hello World.

Correlacionar una unidad de red con el servidor iSeries

Para correlacionar una unidad de red con el servidor iSeries, asegúrese de tener instalado iSeries Access para Windows en el servidor y en la estación de trabajo. Para obtener más información acerca de cómo instalar y configurar iSeries Access para Windows, consulte la sección Instalación de iSeries Access para Windows.

Debe tener configurada una conexión para el servidor iSeries para poder correlacionar una unidad de red.

Para correlacionar una unidad de red, siga estos pasos:

1. Abra el Explorador de Windows:
   a. Con el botón derecho del ratón, pulse el botón **Inicio** de la barra de tareas de Windows.
   b. Pulse **Explorar** en el menú.
2. Seleccione **Correlacionar unidad de red** en el menú **Herramientas**.
3. Seleccione la unidad que desea utilizar para conectarse con el servidor iSeries.
4. Escriba el nombre de vía de acceso al servidor. Por ejemplo:

   MISERVIDOR

   donde ***MISERVIDOR*** es el nombre del servidor iSeries.
5. Marque el recuadro **Reconectar al iniciar la sesión** si está en blanco.
6. Pulse **Aceptar** para finalizar.

La unidad correlacionada aparece ahora en la sección **Todas las carpetas** del Explorador de Windows.

Crear un directorio en el servidor iSeries

Debe crear un directorio en el servidor iSeries en el que pueda guardar las aplicaciones Java. Existen dos formas de hacerlo:

- Crear un directorio mediante iSeries Navigator
  Elija esta opción si tiene instalado iSeries Access para Windows. Si tiene previsto utilizar iSeries Navigator para compilar, optimizar y ejecutar el programa Java, debe seleccionar esta opción para asegurarse de que el programa se guarda en la ubicación correcta para realizar estas operaciones.
- Crear un directorio mediante la línea de entrada de mandatos
  Elija esta opción si no tiene instalado iSeries Access para Windows.

Para obtener información acerca de iSeries Navigator, incluyendo información de instalación, consulte la sección Iniciación a iSeries Navigator.

Crear, compilar y ejecutar un programa Java HelloWorld

La tarea de crear un programa Java Hello World sencillo constituye un excelente punto de partida para familiarizarse con IBM Developer Kit para Java.

Para crear, compilar y ejecutar su propio programa Java Hello World, lleve a cabo los siguientes pasos:

1. Correlacione una unidad de red con el servidor iSeries.
2. Cree un directorio en el servidor iSeries para las aplicaciones Java.

3. Cree el archivo fuente como archivo de texto ASCII (American Standard Code for Information Interchange) en el sistema de archivos integrado. Puede utilizar un producto del IDE (entorno de desarrollo integrado) o un editor de texto, como por ejemplo el Bloc de notas de Windows para codificar la aplicación Java.
   a. Dé al archivo de texto el nombre `HelloWorld.java`. Si desea obtener más información sobre la manera de crear y editar el archivo, consulte la sección Crear y editar archivos fuente Java.
   b. Asegúrese de que el archivo contiene el código fuente siguiente:

```
class HelloWorld {
public static void main (String args[]) {
          System.out.println("Hello World");
     }
}
```

4. Compile el archivo fuente.
   a. Entre el mandato Trabajar con variable de entorno (WRKENVVAR) para comprobar la variable de entorno CLASSPATH. Si no existe, añádala y establézcala en '.' (el directorio actual). Si existe, asegúrese de que '.' encabeza la lista de vías de acceso. Para obtener detalles acerca de la variable de entorno CLASSPATH, consulte el apartado Vía de acceso de clases Java.
   b. Especifique el mandato Arrancar Qshell (STRQSH) para iniciar el intérprete de Qshell.
   c. Utilice el mandato cd (cambiar de directorio) para pasar del directorio actual al directorio del sistema de archivos integrado que contiene el archivo `HelloWorld.java`.
   d. Entre `javac` seguido del nombre del archivo tal y como lo haya guardado en el disco. Por ejemplo, entre `javac HelloWorld.java`.
5. Establezca las autorizaciones de archivo del archivo de clase del sistema de archivos integrado.
6. Optimice la aplicación Java.
   a. En la línea de *entrada de mandatos QSH*, escriba:

```
system "CRTJVAPGM '/midir/miclase.class' OPTIMIZE(20)"
```

      donde *midir* es el nombre de vía de acceso del directorio donde se guarda la aplicación Java y *miclase* es el nombre de la aplicación java compilada.

      Nota: puede especificar un nivel de optimización de 40 como máximo. Un nivel de optimización 40 aumenta la eficiencia de la aplicación Java, pero también limita las posibilidades de depuración. En los primeros estadios de desarrollo de una aplicación Java, puede que desee establecer el nivel de optimización en 20 para poder depurar más fácilmente la aplicación. Consulte el mandato CRTJVAPGM y el parámetro OPTIMIZE para obtener más información.
   b. Pulse la tecla **Intro**.

      Aparece un mensaje que indica que se ha creado un programa Java para la clase.
7. Ejecute el archivo de clase.
   a. Asegúrese de que la vía de acceso de clases Java está configurada correctamente.
   b. En la línea de mandatos de Qshell, escriba `java` seguido de `HelloWorld` para ejecutar `HelloWorld.class` con la máquina virtual Java. Por ejemplo, especifique `java HelloWorld`. También puede utilizar el mandato Ejecutar Java (RUNJVA) en el servidor iSeries para ejecutar `HelloWorld.class`.
   c. Si se ha especificado todo correctamente, se mostrará `"Hello World"` en la pantalla. Aparece la solicitud de la shell (por omisión, un signo $), indicando que Qshell está preparado para otro mandato.
   d. Pulse F3 (Salir) o F12 (Desconectar) para volver a la pantalla de entrada de mandato.

También puede compilar, optimizar y ejecutar fácilmente la aplicación Java mediante iSeries Navigator, una interfaz gráfica de usuario para realizar tareas en el servidor iSeries. Para obtener instrucciones, consulte el apartado Trabajar con aplicaciones Java mediante iSeries Navigator. Para obtener más información acerca de iSeries Navigator, incluyendo información de instalación, consulte la sección Iniciación a iSeries Navigator.

Crear y editar archivos fuente Java

Puede crear y editar archivos fuente Java de diversas formas:

- "Con iSeries Access para Windows".
- "En una estación de trabajo".
- "Con EDTF".
- "Con el programa de utilidad para entrada del fuente".

Con iSeries Access para Windows

Los archivos fuente Java son archivos de texto ASCII (American Standard Code for Information Interchange) del sistema de archivos integrado de servidores iSeries.

Puede crear y editar un archivo fuente Java con iSeries Access para Windows y un editor basado en estación de trabajo.

En una estación de trabajo

Se puede crear un archivo fuente Java en una estación de trabajo. A continuación, hay que transferirlo al sistema de archivos integrado utilizando para ello el protocolo de transferencia de archivos (FTP).

Para crear y editar archivos fuente Java en una estación de trabajo:

1. Cree el archivo ASCII en la estación de trabajo con el editor que prefiera.
2. Conéctese al servidor iSeries con FTP.
3. Transfiera el archivo fuente al directorio del sistema de archivos integrado como archivo binario, para que así conserve el formato ASCII.

Con EDTF

El mandato CL EDTF le permite editar archivos de cualquier sistema de archivos. EDTF es un editor similar al programa de utilidad para entrada del fuente (SEU) para editar archivos continuos o archivos de base de datos. Hallará información en El mandato CL EDTF.

Con el programa de utilidad para entrada del fuente

Puede crear un archivo fuente Java como archivo de texto si emplea el programa de utilidad para entrada del fuente (SEU).

Para crear el archivo fuente Java como archivo de texto mediante SEU, siga estos pasos:

1. Cree un miembro de archivo fuente con SEU.
2. Utilice el mandato Copiar en archivo continuo (CPYTOSTMF) para copiar el miembro de archivo fuente en un archivo continuo del sistema de archivos integrado y convertir al mismo tiempo los datos a formato ASCII.

Si ha de realizar cambios en el código fuente, cambie el miembro de base de datos con SEU y copie de nuevo el archivo.

Para obtener información sobre cómo almacenar archivos, consulte la sección Archivos del sistema de archivos integrado.

Personalizar el servidor iSeries para IBM Developer Kit para Java

Tras instalar IBM Developer Kit para Java en el servidor iSeries, puede personalizar el servidor. Para obtener más información sobre las posibles personalizaciones, consulte la siguiente información:

Vía de acceso de clases
Aprenda a personalizar la manera en que la JVM busca una clase concreta.

Propiedades del sistema Java
Averigüe cómo personalizar las propiedades del sistema Java, las cuales determinan el entorno en que se ejecutan los programas Java en el servidor.

Internacionalización
Lea sobre la personalización de las aplicaciones Java para una zona específica del mundo configurando el huso horario, utilizando entornos locales de Java y codificando datos de tipo carácter.

Vía de acceso de clases Java

La máquina virtual Java utiliza la vía de acceso de clases Java para buscar las clases durante la ejecución. Los mandatos y las herramientas Java la utilizan también para localizar las clases. La vía de acceso de clases por omisión del sistema, la variable de entorno CLASSPATH y el parámetro de mandato de vía de acceso de clases determinan en qué directorios se realiza la búsqueda cuando se desea hallar una clase determinada.

En Java 2 Software Development Kit (J2SDK), Standard Edition, la propiedad java.ext.dirs determina la vía de acceso de clases para las ampliaciones que se cargan. Consulte la sección Instalar ampliaciones para IBM Developer Kit para Java para obtener más información.

La vía de acceso de clases de rutina de carga por omisión está definida por el sistema y no debe cambiarse. En el servidor, la vía de acceso de clases de rutina de carga por omisión especifica dónde se encuentran las clases que forman parte de IBM Developer Kit, de Native Abstract Window Toolkit (NAWT) y otras clases de sistemas.

Para hallar cualquier otra clase en el sistema, debe especificar la vía de acceso de clases en la que debe realizarse la búsqueda; para ello, utilice la variable de entorno CLASSPATH o el parámetro de vía de acceso de clases. El parámetro de vía de acceso de clases que se utilice en una herramienta o en un mandato prevalecerá sobre el valor especificado en la variable de entorno CLASSPATH.

Para trabajar con la variable de entorno CLASSPATH, utilice el mandato Trabajar con variable de entorno (WRKENVVAR). Desde la pantalla WRKENVVAR, se puede añadir o cambiar la variable de entorno CLASSPATH. Los mandatos Añadir variable de entorno (ADDENVVAR) y Cambiar variable de entorno (CHGENVVAR) añaden y cambian, respectivamente, la variable de entorno CLASSPATH.

El valor de la variable de entorno CLASSPATH es una lista de nombres de vías de acceso, separados por el signo de dos puntos (:), en las que se busca una clase determinada. Un nombre de vía de acceso es una secuencia de cero o más nombres de directorio. Estos nombres de directorio van seguidos del nombre del directorio, el archivo ZIP o el archivo JAR en el que se ha de realizar la búsqueda en el sistema de archivos integrado. Los componentes del nombre de vía de acceso van separados por medio del carácter barra inclinada (/). Utilice un punto (.) para indicar cuál es el directorio de trabajo actual.

Para establecer la variable CLASSPATH del entorno Qshell, puede emplear el programa de utilidad de exportación que está disponible al utilizar el intérprete de Qshell.

Estos mandatos añaden la variable CLASSPATH al entorno Qshell y la establecen en el valor `".:/myclasses.zip:/Product/classes."`

- El mandato siguiente establece la variable CLASSPATH del entorno Qshell:

```
export -s CLASSPATH=.:/myclasses.zip:/Product/classes
```

- Este mandato establece la variable CLASSPATH desde la línea de mandatos:

```
ADDENVVAR ENVVAR(CLASSPATH) VALUE(".:/myclasses.zip:/Product/classes")
```

J2SDK busca primero en la vía de acceso de clases de rutina de carga, a continuación en los directorios de ampliación y finalmente en la vía de acceso de clases. El orden de búsqueda de J2SDK, utilizando el ejemplo anterior, es:

1. La vía de acceso de clases de rutina de carga, que se encuentra en la propiedad sun.boot.class.path,
2. Los directorios de ampliación, que se encuentran en la propiedad java.ext.dirs,
3. El directorio de trabajo actual,
4. El archivo myclasses.zip que se encuentra en el sistema de archivos "root" (/),
5. El directorio de clases del directorio Product que hay en el sistema de archivos "root" (/).

Al entrar en el entorno Qshell, la variable CLASSPATH queda establecida en la variable de entorno. El parámetro de vía de acceso de clases especifica una lista de nombres de vía de acceso. La sintaxis es idéntica a la de la variable de entorno CLASSPATH. Las herramientas y mandatos siguientes disponen de un parámetro de vía de acceso de clases:

- El mandato `java` de Qshell
- La herramienta `javac`
- La herramienta `javah`
- La herramienta `javap`
- La herramienta `javadoc`
- La herramienta `rmic`
- El mandato Ejecutar Java (RUNJVA)

Para obtener más información acerca de estos mandatos, consulte la sección Mandatos y herramientas de IBM Developer Kit para Java. Si utiliza el parámetro de vía de acceso de clases con cualquiera de estos mandatos o herramientas, el parámetro hará caso omiso de la variable de entorno CLASSPATH.

Puede alterar temporalmente la variable de entorno CLASSPATH utilizando la propiedad java.class.path. Puede cambiar la propiedad java.class.path, así como otras propiedades, utilizando el archivo SystemDefault.properties. Los valores del archivo SystemDefault.properties alteran temporalmente la variable de entorno CLASSPATH. Para obtener información acerca del archivo SystemDefault.properties, consulte la sección Archivo SystemDefault.properties.

En J2SDK, la opción -Xbootclasspath también afecta a los directorios en que el sistema busca al buscar clases. Utilizar -Xbootclasspath/a:*path* añade la *path* a la vía de acceso de clases de rutina de carga por omisión, /p:*path* coloca *path* antes de la vía de acceso de clases de rutina de carga, y :*path* sustituye a la vía de acceso de clases de rutina de carga por *path*.

Nota: tenga cuidado al especificar -Xbootclasspath porque pueden producirse resultados imprevistos si no se encuentra una clase del sistema o si esta se sustituye de forma incorrecta por una clase definida por el usuario. Por lo tanto, debería permitir que la búsqueda de clases se realice primero en la vía de acceso de clases por omisión del sistema, antes que en una vía de acceso de clases especificada por el usuario.

En Propiedades Java del sistema hallará más información sobre la manera de determinar el entorno en el que se ejecutan los programas Java.

Para obtener más información, consulte los apartads Las API de programas y mandatos CL o Sistema de archivos integrado.

Propiedades del sistema Java

Las propiedades del sistema Java determinan cuál es el entorno en el que se ejecutan los programas Java. Son parecidas a los valores del sistema o las variables de entorno de OS/400.

Al iniciar una instancia de una máquina virtual Java (JVM) se establecen los valores para las propiedades del sistema que afectan a esa JVM.

Puede elegir utilizar los valores por omisión para las propiedades del sistema Java o bien puede especificar valores para ellas utilizando los siguientes métodos:

- Añadir parámetros a la línea de mandatos (o a la API de invocación de la interfaz Java nativa (JNI)) al iniciar el programa Java
- 

  Utilizar la variable de entorno de nivel de trabajo QIBM_JAVA_PROPERTIES_FILE para señalar a un archivo de propiedades específico. Por ejemplo:

```
     ADDENVVAR ENVVAR(QIBM_JAVA_PROPERTIES_FILE)
VALUE(/qibm/userdata/java400/mySystem.properties)
```

- Crear un archivo SystemDefault.properties que se crea en el directorio user.home
- Utilizar el archivo /qibm/userdata/java400/SystemDefalut.properties

OS/400 y la JVM determinan los valores para las propiedades del sistema Java utilizando el siguiente orden de preferencia:

1. Línea de mandatos o API de invocación de JNI
2. Variable de entorno QIBM_JAVA_PROPERTIES_FILE
3. Archivo user.home SystemDefault.properties
4. /QIBM/UserData/Java400/SystemDefault.properties
5. Valores de propiedades del sistema por omisión

Para obtener más información, consulte las siguientes páginas:

Lista de propiedades del sistema Java

Archivo SystemDefault.properties

Archivo SystemDefault.properties

El archivo SystemDefault.properties es un archivo de propiedades estándar de Java que le permite especificar propiedades por omisión del entorno Java.

El archivo SystemDefault.properties que se encuentra en el directorio inicial tiene prioridad sobre el archivoSystemDefault.properties que se encuentra en el directorio /QIBM/UserData/Java400.

Las propiedades que establezca en el archivo /YourUserHome/SystemDefault.properties afectarán solamente a las siguientes máquinas virtuales Java específicas:

- Las JVM que inicie sin especificar una propiedad user.home distinta
- Las JVM que inicien otros usuarios especificando la propiedad `user.home = /YourUserHome/`

Ejemplo: archivo SystemDefault.properties: En el siguiente ejemplo se establecen varias propiedades Java:

```
    #Los comentarios empiezan por el signo de almohadilla
    #Utilice J2SDK 1.4
    java.version=1.4
    #Esto establece una propiedad especial
myown.propname=6
```

Para obtener más información sobre las propiedades del sistema, consulte las páginas siguientes:

Propiedades del sistema Java

Lista de propiedades del sistema Java

Lista de propiedades del sistema Java

Las propiedades Java del sistema determinan cuál es el entorno en el que se ejecutan los programas Java. Son parecidas a los valores del sistema o a las variables de entorno de OS/400.

Al iniciar una máquina virtual Java (JVM) se establecen las propiedades del sistema para esa instancia de la JVM. Para obtener más información sobre cómo especificar valores para propiedades del sistema Java, consulte las siguientes páginas:

Propiedades del sistema Java

Archivo SystemDefault.properties

Para obtener más información sobre las propiedades del sistema Java, consulte Propiedades del sistema Java Secure Socket Extension (JSSE).

La tabla siguiente enumera las propiedades del sistema Java para las versiones soportadas de Java 2 Software Development Kit (J2SDK), Standard Edition:

- J2SDK, versión 1.3
- J2SDK, versión 1.4

Para cada propiedad, la tabla lista el nombre de la propiedad y los valores por omisión aplicables o bien una breve descripción. La tabla indica qué propiedades del sistema tienen valores distintos en las distintas versiones de J2SDK. Cuando la columna que enumera los valores por omisión no indica versiones distintas de J2SDK, todas las versiones soportadas de J2SDK utilizan ese valor por omisión.

Propiedades del sistema	Valor por omisión o descripción
awt.toolkit	 sun.awt.motif.MToolkit awt.toolkit quedará sin establecer a menos que os400.awt.native=true o java.awt.headless=true 

Propiedades del sistema	Valor por omisión o descripción
file.encoding	 ISO8859_1 (valor por omisión) Correlaciona el identificador de juego de caracteres codificado (CCSID) del trabajo OS/400 con el CCSID ASCII ISO correspondiente. Asimismo, establece el valor de file.encoding en el valor Java que representa el CCSID ASCII ISO. Consulte la sección Valores de file.encoding y CCSID de iSeries para obtener una tabla que muestra la relación entre los valores posibles de file.encoding y el CCSID de OS/400 que más se aproxima.
file.encoding.pkg	sun.io
file.separator	/ (barra inclinada)
 java.awt.headless	J2SDK v1.4 = false (valor por omisión) J2SDK v1.3 = Esta propiedad no está disponible al ejecutar J2SDK v.1.3. Esta propiedad especifica si la API de Abstract Windowing Toolkit (AWT) opera en modalidad headless o no. El valor por omisión de false hace que la función completa de AWT solamente esté disponible al habilitar AWT estableciendo os400.awt.native como true. Establecer esta propiedad como true da soporte a la modalidad AWT headless y también fuerza explícitamente que os400.awt.native sea true. 
java.class.path	. (punto) (valor por omisión) Designa la vía de acceso que OS/400 utiliza para localizar clases. Toma por omisión la CLASSPATH especificada por el usuario.
java.class.version	 J2SDK v1.3 = 47.0 J2SDK v1.4 = 48.0 (valor por omisión) 
java.compiler	 jitc_de (valor por omisión) Especifica si el código se compila con el compilador Just-In-Time (JIT) (jitc) o tanto con el compilador JIT como con el proceso directo (jitc_de).
java.ext.dirs	 J2SDK v1.3 = /QIBM/ProdData/Java400/jdk13/lib/ext: /QIBM/UserData/Java400/ext J2SDK v1.4 = /QIBM/ProdData/OS400/Java400/jdk/lib/ext: /QIBM/ProdData/Java400/jdk14/lib/ext: /QIBM/UserData/Java400/ext (valor por omisión) 

Propiedades del sistema	Valor por omisión o descripción
java.home	 J2SDK v1.3 = /QIBM/Prodata/Java400/jdk13 J2SDK v1.4 =/QIBM/ProdData/Java400/jdk14 (valor por omisión) Consulte la sección Soporte para varios Java Development Kits (JDK) para obtener detalles.
java.library.path	/QSYS.LIB/QSHELL.LIB:/QSYS.LIB/QGPL.LIB: /QSYS.LIB/QTEMP.LIB:/QSYS.LIB/QDEVELOP.LIB: /QSYS.LIB/QBLDSYS.LIB:/QSYS.LIB/QBLDSYSR.LIB (valor por omisión) Lista de bibliotecas de OS/400
 java.net.preferIPv4Stack	true (valor por omisión) false (no's) En las máquinas de pila dual, las propiedades del sistema se proporcionan para establecer la pila de protocolo preferidad (IPv4 o IPv6), así como los tipos de familias de direcciones preferidas (inet4 o inet6). La pila IPv6 es la preferida por omisión, ya que en una máquina de pila dual el socket IPv6 puede comunicarse con iguales IPv4 y IPv6. Este valor puede cambiarse con esta propiedad. java.net.preferIPv4Stack es específico de J2SDK v1.4. Para obtener más información, consulte Protocolo IPv6.
 java.net.preferIPv6Addresses	true false (no's) (valor por omisión) Aunque IPv6 está disponible en el sistema operativo, la preferencia por omisión es preferir una dirección correlacionada con IPv4 antes que una dirección IPv6. Esta propiedad controla si se utilizan direcciones IPv6 (true) o IPv4 (false). java.net.preferIPv4Stack es específico de J2SDK v1.4. Para obtener más información, consulte Protocolo IPv6.
java.policy	 J2SDK v1.3 = /QIBM/ProdData/Java400/jdk13/lib/security/java.policy J2SDK v1.4 = /QIBM/ProdData/OS400/Java400/jdk/lib/security/java.policy (valor por omisión)
java.specification.name	 Especificación API plataforma Java (valor por omisión) Especificación de lenguaje Java
java.specification.vendor	Sun Microsystems, Inc.
java.specification.version	 J2SDK v1.3 = 1.3 J2SDK v1.4 = 1.4 (valor por omisión)
java.use.policy	true
java.vendor	IBM Corporation

Propiedades del sistema	Valor por omisión o descripción
java.vendor.url	http://www.ibm.com
java.version	 1.3.1 1.4.2. (valor por omisión)  Determina qué versión de J2SDK desea ejecutar. Instalar una única versión de J2SDK convierte a esa versión en la versión por omisión. Especificar una versión que no está instalada da como resultado un mensaje de error. Si no se puede especificar una versión, se utiliza la versión más reciente de J2SDK como valor por omisión. **Nota:** Se ignora java.version si se coloca en el archivo SystemDefault.properties y se intenta utilizar la interfaz Java nativa (JNI). Para obtener más información, consulte la sección Soporte para múltiples J2SDK.
java.vm.name	VM clásica
java.vm.specification.name	Especificación de máquina virtual Java
java.vm.specification.vendor	Sun Microsystems, Inc.
java.vm.specification.version	1.0
java.vm.vendor	IBM Corporation
java.vm.version	 J2SDK v1.3 = 1.3 J2SDK v1.4 = 1.4 (valor por omisión) 
line.separator	\n
os.arch	PowerPC
os.name	OS/400
os.version	 V5R3M0 (valor por omisión)  Obtiene el nivel de release de OS/400 a partir de la interfaz de programas de aplicación (API) Recuperar información de producto.
 os400.awt.native	Controla si la API de Abstract Windowing Toolkit (AWT) está soportada o no. Los valores válidos son true y false. El valor por omisión es false a menos que se establezca java.awt.headless=true, en cuyo caso os400.awt.native es true implícitamente. 
os400.certificateContainer	Manda al soporte SSL (capa de sockets segura) que utilice el contenedor de certificados especificado para el programa Java que se ha iniciado y la propiedad que se ha indicado. Si especifica la propiedad os400.secureApplication del sistema, se hace caso omiso de esta propiedad del sistema. Por ejemplo, especifique `-Dos400.certificateContainer=/home/username/mykeyfile.kdb` o cualquier otro archivo de claves del sistema de archivos integrado.

Propiedades del sistema	Valor por omisión o descripción
os400.certificateLabel	Puede especificar esta propiedad del sistema junto con la propiedad os400.certificateContainer del sistema. Esta propiedad le permite seleccionar qué certificado del contenedor especificado desea que la capa de sockets segura (SSL) utilice. Por ejemplo, entre `-Dos400.certificateLabel=myCert`, donde myCert es el nombre de etiqueta que usted asigna al certificado por medio del Gestor de certificados digitales (DCM) al crear o importar el certificado.
os400.child.stdio.convert	Controla la conversión de datos para stdin, stdout y stderr en Java. La conversión de datos entre datos ASCII y datos EBCDIC (Extended Binary Coded Decimal Interchange Code) se produce por omisión en la máquina virtual Java. Utilizar esta propiedad para activar y desactivar estas conversiones también afecta a los procesos hijo que este proceso inicie utilizando el método runtime.exec(). Consulte los valores por omisión.
os400.class.path.security.check	20 (valor por omisión) Valores válidos: 0: Sin comprobación de seguridad 10: equivalente a RUNJVA CHKPATH(*IGNORE) 20: equivalente a RUNJVA CHKPATH(*WARN) 30: equivalente a RUNJVA CHKPATH(*SECURE)
os400.class.path.tools	0 (valor por omisión) Valores válidos: 0: No hay herramientas Sun en la propiedad java.class.path 1: Añade el archivo de herramietas específico de J2SDK como prefijo de la propiedad java.class.path En J2SDK v1.3, la vía de acceso a tools.jar es /QIBM/ProdData/Java400/jdk13/lib/ En J2SDK v1.3, la vía de acceso a tools.jar es /QIBM/ProdData/OS400/Java400/jdk/lib/
os400.create.type	interpret (valor por omisión) Valores válidos: interpret: equivalente a RUNJVA OPTIMIZE(*INTERPRET) e INTERPRET(*OPTIMIZE), o INTERPRET(*YES) direct: Otherwise
os400.define.class.cache.file	valor por omisión = null. Especifica el nombre de un archivo JAR o ZIP. Consulte el apartado ″Utilizar antememoria para cargadores de clases de usuario″ en Consideraciones sobre el rendimiento de Java.
os400.define.class.cache.hours	valor por omisión = 768 valor decimal máximo = 9999. Especifica un valor decimal. Consulte el apartado ″Utilizar antememoria para cargadores de clases de usuario″ en Consideraciones sobre el rendimiento de Java.
os400.define.class.cache.maxpgms	valor por omisión = 5000 valor decimal máximo = 40000 Especifica un valor decimal. Consulte el apartado ″Utilizar antememoria para cargadores de clases de usuario″ en Consideraciones sobre el rendimiento de Java.
os400.defineClass.optLevel	0
 os400.display.properties	Si se establece este valor como ’true’, se imprimirán todas las propiedades de la máquina virtual Java en salida estándar. No se reconocen otros valores. 

Propiedades del sistema	Valor por omisión o descripción
os400.enbpfrcol	0 (valor por omisión) Valores válidos: 0: equivalente a CRTJVAPGM ENBPFRCOL(*NONE) 1: equivalente a CRTJVAPGM ENBPFRCOL(*ENTRYEXIT) 7: equivalente a CRTJVAPGM ENBPFRCOL(*FULL) Para un valor no cero, JIT genera eventos *JVAENTRY, *JVAEXIT, *JVAPRECALL y *JVAPOSTCALL.
os400.exception.trace	Esta propiedad se utiliza sólo para la depuración.Si se especifica esta propiedad, las excepciones más recientes se envían a la salida estándar cuando se sale de la VM.
os400.file.create.auth, os400.dir.create.auth	Estas propiedades especifican las autorizaciones asignadas a los archivos y directorios. Si se especifican las propiedades sin valores o con valores no soportados, la autorización de uso público es *NONE. Puede especificar os400.file.create.auth=RWX o os400.dir.create.auth=RWX, donde R=lectura, W=escritura y X=ejecución. Cualquier combinación de estas autorizaciones es válida.
os400.file.io.mode	Convierte el CCSID del archivo si es diferente del valor de file.encoding al especificar TEXT, en lugar del valor por omisión, que es BINARY.
 os400.gc.heap.size.init	Una alternativa a utilizar -Xms (establecer el tamaño de GC inicial). Se recomienda que los clientes continúen utilizando -Xms a menos que no tengan otra opción ya que esta propiedad es específica de OS/400. Esta propiedad se introdujo principalmente para los usuarios puedan especificar un tamaño de GC inicial en el archivo SystemDefault.properties. **Nota:** Utilice esta propiedad con cuidado, ya que prevalecerá sobre -Xms si se especifica. El valor debe ser un entero en tamaño de kilobytes y sin comas. 
 os400.gc.heap.size.max	Una alternativa a utilizar -Xmx (establecer el tamaño de GC máximo). Se recomienda que los clientes continúen utilizando -Xmx a menos que no tengan otra opción ya que esta propiedad es específica de OS/400. Esta propiedad se introdujo principalmente para los usuarios puedan especificar un tamaño de GC máximo en el archivo SystemDefault.properties. **Nota:** Utilice esta propiedad con cuidado, ya que prevalecerá sobre -Xmx si se especifica. El valor debe ser un entero en tamaño de kilobytes y sin comas. 
os400.interpret	0 (valor por omisión) Valores válidos: 0: equivalente a CRTJVAPGM INTERPRET(*NO) 1: equivalente a CRTJVAPGM INTERPRET(*YES)
os400.jit.mmi.threshold	Establece el número de veces que un método se ejecuta utilizando el MMI(Mixed-Mode Interpeter) antes de que OS/400 utilice el compilador JIT para compilar el método en instrucciones de máquina nativa. Normalmente no deberá cambiar el valor por omisión de 2000. • Un valor de cero inhabilita MMI y compila métodos cuando se les llama por primera vez. • Los valores inferiores al valor por omisión tienden a prolongar el tiempo de arranque y degradar el rendimiento general. • Los valores superiores al valor por omisión degradan el rendimiento inicialmente hasta que se alcanza el umbral, tendiendo entonces a mejorar el rendimiento de ejecución general.

Propiedades del sistema	Valor por omisión o descripción
os400.optimization	0 (valor por omisión) Valores válidos: 0: equivalente a CRTJVAPGM OPTIMIZE(*INTERPRET) 10: equivalente a CRTJVAPGM OPTIMIZE(10) 20: equivalente a CRTJVAPGM OPTIMIZE(20) 30: equivalente a CRTJVAPGM OPTIMIZE(30) 40: equivalente a CRTJVAPGM OPTIMIZE(40)
os400.pool.size	Define cuánto espacio (en kilobytes) se debe establecer como disponible para cada agrupación de almacenamiento dinámico en el almacenamiento dinámico local de hebras.
os400.run.mode	jitc_de (valor por omisión) Valores válidos: interpret: equivalente a RUNJVA OPTIMIZE(*INTERPRET) y INTERPRET(*OPTIMIZE), o INTERPRET(*YES) program_create_type jitc_de: Otherwise
 os400.run.verbose	Si se establece este valor como 'true', se imprimirá la carga de clases verbose como salida estándar. No se reconocen otros valores. Se consigue lo mismo que especificando -verbose en QSHELL u OPTION(*VERBOSE) en los mandatos CL, excepto que esta propiedad funciona en el archivo SystemDefault.properties. 
os400.runtime.exec	EXEC (valor por omisión) Valores válidos: EXEC = Invocar funciones mediante runtime.exec() utilizando la interfaz EXEC. QSHELL = Invocar funciones mediante runtime.exec() utilizando el intérprete Qshell. Para obtener más información, consulte Utilizar java.lang.Runtime.exec()
os400.secureApplication	Asocia el programa Java que se inicia al utilizar este propiedad del sistema (os400.secureApplication) con el nombre de aplicación segura registrada. Para ver los nombres de las aplicaciones seguras registradas, utilice el Gestor de certificados digitales (DCM).
os400.security.properties	Permite tener pleno control de qué archivo java.security se utiliza. Si se especifica esta propiedad, J2SDK no utilizará ningún otro archivo java.security, incluido el valor por omisión java.security específico de J2SDK.
os400.stderr	Permite correlacionar stderr con un archivo o un socket. Consulte los valores por omisión.
os400.stdin	Permite correlacionar stdin con un archivo o un socket. Consulte los valores por omisión.
os400.stdin.allowed	valor por omisión = 1 Especifica si stdin está permitido (1) o no (0). Si el llamador está ejecutando un trabajo de proceso por lotes, stdin no debe estar permitido.
os400.stdio.convert	Permite el control de la conversión de datos para stdin, stdout y stderr en Java. La conversión de datos se produce por omisión en la máquina virtual Java para convertir datos ASCII a o desde EBCDIC. Puede activar o desactivar estas conversiones con esta propiedad, que afecta al programa Java actual. Consulte los valores por omisión.
os400.stdout	Permite correlacionar stdout con un archivo o un socket. Consulte los valores por omisión.
os400.verify.checks.disable	65535 (valor por omisión) Este valor de propiedad del sistema es una serie que representa la suma de uno o más valores numéricos. Hallará una lista de estos valores en la sección Valores numéricos de os400.verify.checks.disable.

Propiedades del sistema	Valor por omisión o descripción
 os400.xrun.option	Esta propiedad del sistema permite utilizar la opción Qshell -Xrun especificando una propiedad. Puede utilizarla para especificar un programa agente para ejecutarse durante el inicio de la JVM. Durante el inicio se llama a la función JVM_OnLoad.
path.separator	: (dos puntos)
sun.boot.class.path	 Lista todos los archivos necesarios para el cargador de clases de arranque por omisión. No cambie este valor.
user.dir	Directorio de trabajo actual que utiliza la API getcwd.
user.home	Recupera el directorio de trabajo inicial utilizando la API Obtener (getpwnam). Puede colocar un archivo SystemDefault.properties en la vía de acceso user.home para alterar temporalmente las propiedades por omisión en /QIBM/UserData/Java400/SystemDefault.properties. Puede personalizar el servidor iSeries para especificar un conjunto propio de valores de propiedad por omisión.
user.language	La máquina virtual Java utiliza esta propiedad del sistema para leer el valor de LANGID del trabajo y lo emplea para localizar el idioma correspondiente.
user.name	La máquina virtual Java utiliza esta propiedad del sistema para recuperar el verdadero nombre del perfil de usuario desde la sección de seguridad (Security.UserName) de TCB (Trusted Computing Base).
user.region	La máquina virtual Java utiliza esta propiedad del sistema para leer el valor de CNTRYID del trabajo y lo emplea para determinar la región del usuario.
user.timezone	Universal Time Coordinate (UTC) (valor por omisión) La máquina virtual Java utiliza esta propiedad del sistema para obtener el nombre de huso horario mediante la API QlgRetrieveLocalInformation. La JVM se dirige primero al objeto QLOCALE del sistema. Si no se encuentra, entonces la JVM se dirigirá al valor del sistema QTIMZON. Si el valor del sistema QTIMZON contiene un objeto QTIMZON no reconocido, la JVM utiliza UTC como valor por omisión de user.timezone.

Internacionalización

Puede personalizar sus programas Java para una zona específica del mundo creando programas Java internacionalizados. Utilizando el huso horario, los entornos locales y la codificación de caracteres, puede asegurarse de que el programa Java refleja la hora, lugar e idioma correctos.

Para obtener más información, consulte lo siguiente:

Huso horario
Aprenda a configurar el huso horario en el servidor de forma que los programas Java sensibles al huso horario utilicen la hora correcta.

Entornos locales Java
Utilice la lista de entornos locales Java para ayudar a asegurar que los programas Java proporcionan soporte para el idioma, los datos culturales o caracteres específicos de una zona geográfica.

Codificación de caracteres
Lea sobre cómo los programas Java pueden convertir datos en distintos formatos, permitiendo a las aplicaciones transferir y utilizar información de muchas clases de juegos de caracteres internacionales.

Ejemplos
Revise ejemplos que pueden ayudarle a utilizar el huso horario, los entornos locales y la codificación de caracteres para crear un programa Java internacionalizado.

Para obtener más información sobre la internacionalización, consulte lo siguiente:

- Globalización OS/400
- Documentación sobre internacionalización de Sun Microsystems, Inc.

Configuración del huso horario

Cuando tenga programas Java que son sensibles al huso horario, deberá configurar el huso horario en el servidor de forma que los programas Java utilicen la hora correcta.

Para determinar la hora local correctamente, la máquina virtual Java (JVM) requiere que establezca el valor del sistema OS/400 QUTCOFFSET y la información de hora en el parámetro de usuario LOCALE para el usuario o trabajo actual:

•

La JVM determina la UTC (Coordinated Universal Time) correcta al comparar el valor de QUTCOFFSET con la hora local para el sistema

- La JVM devuelve la hora local correcta al sistema utilizando la propiedad del sistema Java user.timezone.

Nota: A partir de la V5R3, una alternativa a establecer QUTCOFFSET y LOCALE es utilizar el valor del sistema QTIMZON. La JVM se dirige primero al objeto QLOCALE del sistema. Si no se encuentra, entonces la JVM se dirigirá al valor del sistema QTIMZON. Si el valor del sistema QTIMZON contiene un objeto QTIMZON no reconocido, la JVM utiliza UTC como valor por omisión de user.timezone.

QUTCOFFSET y user.timezone: El valor del sistema OS/400 QUTCOFFSET representa el desplazamiento de UTC (Coordinated Universal Time) para el sistema. QUTCOFFSET especifica la diferencia de hora entre UTC (u Hora Media de Greeenwich) y la hora actual del sistema. El valor por omisión para QUTCOFFSET es cero (+00:00).

El valor QUTCOFFSET permite a la JVM determinar el valor correcto para la hora local. Por ejemplo, el valor para QUTCOFFSET para especificar el Horario Estándar Central de EEUU (CST) es -6:00. Para especificar el horario de verano (CDT), QUTCOFFSET tiene un valor de -5:00.

La propiedad del sistema Java user.timezone utiliza la hora UTC como valor por omisión. A menos que especifique un valor distinto, la JVM reconoce la hora UTC como la hora actual.

Para obtener más información sobre QUTCOFFSET y las propiedades del sistema Java, consulte las siguientes páginas:

Valor del sistema OS/400: QUTCOFFSET

Propiedades del sistema Java

LOCALE: El parámetro LOCALE en un perfil de usuario especifica el objeto *LOCALE que debe utilizarse para la variable de entorno LANG. No confunda el objeto OS/400 *LOCALE con los entornos locales Java.

Al establecer debidamente la información de entorno local, la JVM puede establecer la propiedad user.timezone en el huso horario correcto. Puede establecer la propiedad user.timezone para alterar temporalmente el valor por omisión suministrado por el objeto *LOCALE.

Para obtener más información sobre cómo utilizar entornos locales y establecer propiedades del sistema Java, consulte las siguientes páginas:

Entornos locales

Propiedades del sistema Java

La categoría LC_TOD define reglas para el cambio de horario de verano e información del huso horario para un entorno local.

Nota: Para utilizar el cambio de horario de verano, debe ajustar el valor del sistema QUTCOFFSET con el desplazamiento correcto.

El ejemplo siguiente muestra la información de la categoría LC_TOD que debe incluir en el objeto de entorno local para poder configurar el huso horario correcto para Java:

```
     LC_TOD

     % TZDIFF es el número de minutos de diferencia desde UTC (o GMT)
tzdiff   -300
     % Nombre de huso horario (es el valor que se pasaría a
     % la JVM como propiedad user.timezone.)
tname    "<C><S><T>"
     % Recuerde ajustar el valor de QUTCOFFSET al utilizar
     % el cambio de horario de verano (DST)
     % Nombre utilizado para DST.
dstname   "<C><D><T>"
     % En esta parte de EE. UU., el DST empieza el primer domingo
     % de abril a las 2 de la mañana
dststart 4,1,1,7200
% En esta zona de EE. UU., DST termina el último domingo de octubre.
dstend   10,-1,1,7200
% Diferencia en segundos.
dstshift 3600

END LC_TOD
```

La categoría LC_TOD del entorno local contiene el campo tname, que debe establecer con el mismo valor que el huso horario.

Para series de husos horarios válidos, consulte la información de consulta de Javadoc para la clase java.util.TimeZone. Para obtener más información sobre cómo trabajar con entornos locales, consulte las siguientes páginas:

Trabajar con entornos locales

Información de consulta de Javadoc TimeZone

Codificaciones de caracteres de Java

Internamente, la máquina virtual Java (JVM) siempre trabaja con datos en formato Unicode. Sin embargo, todos los datos transferidos a la JVM o desde ella tienen un formato que se corresponde con la propiedad file.encoding. Los datos que entran en la JVM para lectura se convierten de file.encoding a Unicode, y los datos que salen de la JVM se convierten de Unicode a file.encoding.

Los archivos de datos de los programas Java se almacenan en el sistema de archivos integrado. Los archivos del sistema de archivos integrado están marcados con un identificador de juego de caracteres (CCSID) que identifica la codificación de caracteres de los datos que hay en el archivo. En la tabla Valores de file.encoding y CCSID de iSeries hallará una descripción de la correlación establecida entre la propiedad file.encoding y el CCSID del servidor iSeries.

Cuando un programa Java lee datos, estos deberían tener la codificación de caracteres que se corresponde con el valor de la propiedad file.encoding. Cuando un programa Java escribe datos en un archivo, los datos se escriben en una codificación de caracteres en correspondencia con el valor de file.encoding. Esto es igualmente aplicable a los archivos de código fuente (archivos .java) procesados por el mandato javac y a los datos que se envían y reciben a través de los sockets del protocolo de control de transmisión/protocolo Internet (TCP/IP) mediante el paquete .net.

Los datos que se leen o escriben en System.in, System.out y System.err se manejan de distinta forma que los datos que se leen o escriben en otros orígenes cuando están asignados a la entrada estándar (stdin), a la salida estándar (stdout) y a la salida de error estándar (stderr). Puesto que, normalmente, la entrada estándar, la salida estándar y la salida de errores estándar están conectadas a dispositivos EBCDIC en el servidor iSeries, la JVM realiza en los datos una conversión para que pasen de tener la codificación de caracteres normal file.encoding a tener un CCSID que coincida con el CCSID del trabajo de iSeries. Cuando System.in, System.out o System.err se redirigen a un archivo o a un socket y no se dirigen a la entrada estándar, a la salida estándar ni a la salida de error estándar, esta conversión de datos adicional no se realiza y los datos siguen teniendo la codificación de caracteres correspondiente a file.encoding.

Cuando es necesario leer o escribir datos en un programa Java utilizando una codificación de caracteres distinta de file.encoding, el programa puede emplear las clases Java de E/S java.io.InputStreamReader, java.io.FileReader, java.io.OutputStreamReader y java.io.FileWriter. Estas clases Java permiten especificar un valor de file.encoding que tiene prioridad sobre la propiedad file.encoding por omisión que utiliza en ese momento la JVM.

Los datos destinados a o procedentes de la base de datos DB2/400, por medio de las API de JDBC, se convierten a o desde el CCSID de la base de datos de iSeries.

Los datos transferidos a o desde otros programas por medio de la interfaz Java nativa (JNI) no se convierten.

Para obtener más información acerca de la internacionalización, consulte la sección Globalización de OS/400.

También puede consultar la documentación relativa a internacionalización de Sun Microsystems, Inc. para obtener más información.

Valores de file.encoding y CCSID de iSeries: Esta tabla muestra la relación que existe entre los valores posibles de file.encoding y el identificador de juego de caracteres (CCSID) de iSeries que más se aproxima.

Para obtener más información sobre el soporte de file.encoding, consulte Codificaciones soportadas de Sun Microsystems, Inc.

file.encoding	CCSID	Descripción
ASCII	367	Código estándar americano para intercambio de información
Big5	950	BIG-5 para chino tradicional ASCII de 8 bits
Big5_HKSCS	950	Big5_HKSCS
Big5_Solaris	950	Big5 con siete correlaciones de caracteres ideográficos Hanzi adicionales para el entorno local Solaris zh_TW.BIG5
CNS11643	964	Juego de caracteres nacionales para chino tradicional
Cp037	037	EBCDIC de IBM para EE. UU., Canadá, los Países Bajos, ...
Cp273	273	EBCDIC de IBM para Alemania y Austria
Cp277	277	EBCDIC de IBM para Dinamarca y Noruega
Cp278	278	EBCDIC de IBM para Finlandia y Suecia
Cp280	280	EBCDIC de IBM para Italia
Cp284	284	EBCDIC de IBM para España y América Latina
Cp285	285	EBCDIC de IBM para el Reino Unido
Cp297	297	EBCDIC de IBM para Francia
Cp420	420	Árabe EBCDIC de IBM
Cp424	424	Hebreo EBCDIC de IBM
Cp437	437	PC de EE. UU. ASCII de 8 bits
Cp500	500	Internacional EBCDIC de IBM
Cp737	737	Griego MS-DOS ASCII de 8 bits
Cp775	775	Báltico MS-DOS ASCII de 8 bits
Cp838	838	EBCDIC de IBM para Tailandia
Cp850	850	Multinacional Latin-1 ASCII de 8 bits
Cp852	852	Latin-2 ASCII de 8 bits
Cp855	855	Cirílico ASCII de 8 bits
Cp856	0	Hebreo ASCII de 8 bits
Cp857	857	Latin-5 ASCII de 8 bits
Cp860	860	ASCII de 8 bits para Portugal
Cp861	861	ASCII de 8 bits para Islandia
Cp862	862	Hebreo ASCII de 8 bits
Cp863	863	ASCII de 8 bits para Canadá

file.encoding	CCSID	Descripción
Cp864	864	Árabe ASCII de 8 bits
Cp865	865	ASCII de 8 bits para Dinamarca y Noruega
Cp866	866	Cirílico ASCII de 8 bits
Cp868	868	Urdu ASCII de 8 bits
Cp869	869	Griego ASCII de 8 bits
Cp870	870	Latin-2 EBCDIC de IBM
Cp871	871	EBCDIC de IBM para Islandia
Cp874	874	ASCII de 8 bits para Tailandia
Cp875	875	Griego EBCDIC de IBM
Cp918	918	Urdu EBCDIC de IBM
Cp921	921	Báltico ASCII de 8 bits
Cp922	922	ASCII de 8 bits para Estonia
Cp930	930	Japonés katakana ampliado EBCDIC de IBM
Cp933	933	Coreano EBCDIC de IBM
Cp935	935	Chino simplificado EBCDIC de IBM
Cp937	937	Chino tradicional EBCDIC de IBM
Cp939	939	Japonés latino ampliado EBCDIC de IBM
Cp942	942	Japonés ASCII de 8 bits
 Cp942C	942	Variante de Cp942 
Cp943	943	Datos mixtos de PC japonés para el entorno abierto
Cp943C	943	Datos mixtos de PC japonés para el entorno abierto
Cp948	948	Chino tradicional IBM ASCII de 8 bits
Cp949	944	KSC5601 para coreano ASCII de 8 bits
 Cp949C	949	Variante de Cp949 
Cp950	950	BIG-5 para chino tradicional ASCII de 8 bits
Cp964	964	Chino tradicional EUC
Cp970	970	Coreano EUC
Cp1006	1006	Urdu de 8 bits ISO
Cp1025	1025	Cirílico EBCDIC de IBM
Cp1026	1026	EBCDIC de IBM para Turquía
Cp1046	1046	Árabe ASCII de 8 bits
Cp1097	1097	Persa EBCDIC de IBM

file.encoding	CCSID	Descripción
Cp1098	1098	Persa ASCII de 8 bits
Cp1112	1112	Báltico EBCDIC de IBM
Cp1122	1122	EBCDIC de IBM para Estonia
Cp1123	1123	EBCDIC de IBM para Ucrania
Cp1124	0	8 bits ISO para Ucrania
 Cp1140	1140	Variante de Cp037 con carácter Euro 
 Cp1141	1141	Variante de Cp273 con carácter Euro 
 Cp1142	1142	Variante de Cp277 con carácter Euro 
 Cp1143	1143	Variante de Cp278 con carácter Euro 
 Cp1144	1144	Variante de Cp280 con carácter Euro 
 Cp1145	1145	Variante de Cp284 con carácter Euro 
 Cp1146	1146	Variante de Cp285 con carácter Euro 
 Cp1147	1147	Variante de Cp297 con carácter Euro 
 Cp1148	1148	Variante de Cp500 con carácter Euro 
 Cp1149	1149	Variante de Cp871 con carácter Euro 
Cp1250	1250	Latin-2 MS-Win
Cp1251	1251	Cirílico MS-Win
Cp1252	1252	Latin-1 MS-Win
Cp1253	1253	Griego MS-Win
Cp1254	1254	Turco MS-Win
Cp1255	1255	Hebreo MS-Win
Cp1256	1256	Árabe MS-Win
Cp1257	1257	Báltico MS-Win
Cp1258	1251	Ruso MS-Win
Cp1381	1381	GB para chino simplificado ASCII de 8 bits
Cp1383	1383	Chino simplificado EUC

file.encoding	CCSID	Descripción
Cp33722	33722	Japonés EUC
EUC_CN	1383	EUC para chino simplificado
EUC_JP	 5050 	EUC para japonés
 EUC_JP_LINUX	0	JISX 0201, 0208 , codificación EUC Japonés 
EUC_KR	970	EUC para coreano
EUC_TW	964	EUC para chino tradicional
 GB2312	1381	GB para chino simplificado ASCII de 8 bits 
GB18030	 1392 	Chino simplificado, estándar PRC
GBK	1386	Nuevo ASCII 9 de 8 bits para chino simplificado
 ISCII91	806	Codificación ISCII91 de scripts Indic
 ISO2022CN	965	ISO 2022 CN, Chino (sólo conversión a Unicode) 
 ISO2022_CN_CNS	965	CNS11643 en formato ISO 2022 CN, Chino tradicional (sólo conversión de Unicode)
 ISO2022_CN_GB	1383	GB2312 en formato ISO 2022 CN, Chino simplificado (sólo conversión de Unicode)
ISO2022CN_CNS	 965 	ASCII de 7 bits para chino tradicional
ISO2022CN_GB	 1383 	ASCII de 7 bits para chino simplificado
ISO2022JP	5054	ASCII de 7 bits para japonés

file.encoding	CCSID	Descripción
ISO2022KR	25546	ASCII de 7 bits para coreano
ISO8859_1	819	ISO 8859-1 Alfabeto Latino Núm. 1
ISO8859_2	912	ISO 8859-2 ISO Latin-2
ISO8859_3	» 0 «	ISO 8859-3 ISO Latin-3
ISO8859_4	914	ISO 8859-4 ISO Latin-4
ISO8859_5	915	ISO 8859-5 ISO Latin-5
ISO8859_6	1089	ISO 8859-6 ISO Latin-6 (árabe)
ISO8859_7	813	ISO 8859-7 ISO Latin-7 (griego/latino)
ISO8859_8	916	ISO 8859-8 ISO Latin-8 (hebreo)
ISO8859_9	920	ISO 8859-9 ISO Latin-9 (ECMA-128, Turquía)
» ISO8859_13	0	Alfabeto Latino Núm. 7 «
» ISO8859_15	923	ISO8859_15 «
» ISO8859_15_FDIS	923	ISO 8859-15, Alfabeto Latino Núm. 9 «
» ISO-8859-15	923	ISO 8859-15, Alfabeto Latino Núm. 9 «
JIS0201	897	JIS X0201
JIS0208	» 5052 «	JIS X0208
JIS0212	» 0 «	JIS X0212
» JISAutoDetect	0	Detecta y convierte de Shift-JIS, EUC-JP, ISO 2022 JP (sólo conversión a Unicode) «
Johab	» 0 «	Codificación (completa) Hangul para coreano.

file.encoding	CCSID	Descripción
K018_R	878	Cirílico
KSC5601	949	Coreano ASCII de 8 bits
MacArabic	1256	Árabe Macintosh
MacCentralEurope	1282	Latin-2 Macintosh
MacCroatian	1284	Croata Macintosh
MacCyrillic	1283	Cirílico Macintosh
MacDingbat	0	Dingbat Macintosh
MacGreek	1280	Griego Macintosh
MacHebrew	1255	Hebreo Macintosh
MacIceland	1286	Islandia Macintosh
MacRoman	0	Romano Macintosh
MacRomania	1285	Rumanía Macintosh
MacSymbol	0	Símbolo Macintosh
MacThai	0	Tailandés Macintosh
MacTurkish	1281	Turco Macintosh
MacUkraine	1283	Ucrania Macintosh
MS874	874	MS-Win para Tailandia

file.encoding	CCSID	Descripción
MS932	943	Japonés Windows
MS936	936	Chino simplificado Windows
MS949	949	Coreano Windows
MS950	950	Chino tradicional Windows
MS950_HKSCS	NA	Chino tradicional Windows con extensiones de Hong Kong, Región Administrativa Especial de China
SJIS	932	Japonés ASCII de 8 bits
TIS620	874	TIS 620
US-ASCII	367	Código estándar americano para intercambio de información
UTF8	1208	UTF-8 (CCSID 1208 de IBM, que no está disponible todavía en el servidor iSeries)
UTF-16	1200	Formato de transformación UCS de dieciséis bits, orden de bytes identificado por una marca de orden de bytes opcional
UTF-16BE	1200	Formato de transformación Unicode de dieciséis bits, orden de bytes de gran memoria numérica
UTF-16LE	1200	Formato de transformación Unicode de dieciséis bits, orden de bytes de pequeña memoria numérica
UTF-8	1208	Formato de transformación UCS de ocho bits
Unicode	13488	UNICODE, UCS-2
UnicodeBig	13488	Idéntico a Unicode
UnicodeBigUnmarked		Unicode sin marca de orden de bytes
UnicodeLittle		Unicode con orden de bytes little-endian

file.encoding	CCSID	Descripción
UnicodeLittleUnmarked		UnicodeLittle sin marca de orden de bytes

En Valores de file.encoding por omisión hallará los valores por omisión.

Valores de file.encoding por omisión: Esta tabla muestra cómo se establece el valor de file.encoding tomando como base el identificador de juego de caracteres (CCSID) de iSeries cuando se inicia la máquina virtual Java.

CCSID iSeries	File.encoding por omisión	Descripción
37	ISO8859_1	Inglés para EE. UU., Canadá, Nueva Zelanda y Australia; portugués para Portugal y Brasil; y holandés para los Países Bajos
256	ISO8859_1	Internacional nº 1
273	ISO8859_1	Alemán/Alemania, alemán/Austria
277	ISO8859_1	Danés/Dinamarca, noruego/Noruega, noruego/Noruega, NY
278	ISO8859_1	Finlandés/Finlandia
280	ISO8859_1	Italiano/Italia
284	ISO8859_1	Catalán/España, español/España
285	ISO8859_1	Inglés/Gran Bretaña, inglés/Irlanda
290	Cp943C	Parte SBCS del CCSID mixto EBCDIC japonés (CCSID 5026)
297	ISO8859_1	Francés/Francia
420	Cp1046	Árabe/Egipto
423	ISO8859_7	Grecia
424	ISO8859_8	Hebreo/Israel
500	ISO8859_1	Alemán/Suiza, francés/Bélgica, francés/Canadá, francés/Suiza
833	Cp970	Parte SBCS del CCSID mixto EBCDIC coreano (CCSID 933)
836	Cp1383	Parte SBCS del CCSID mixto EBCDIC para chino simplificado (CCSID 935)
838	TIS620	Tailandés
870	ISO8859_2	Checo/República Checa, croata/Croacia, húngaro/Hungría, polaco/Polonia
871	ISO8859_1	Islandés/Islandia
875	ISO8859_7	Griego/Grecia
880	ISO8859_5	Bulgaria (ISO 8859_5)
905	ISO8859_9	Ampliado de Turquía
918	Cp868	Urdu
930	Cp943C	Japonés EBCDIC mixto (similar al CCSID 5026)

CCSID iSeries	File.encoding por omisión	Descripción
933	Cp970	Coreano/Corea
935	Cp1383	Chino simplificado
937	Cp950	Chino tradicional
939	Cp943C	Japonés EBCDIC mixto (similar al CCSID 5035)
1025	ISO8859_5	Bielorruso/Bielorrusia, búlgaro/Bulgaria, macedonio/Macedonia, ruso/Rusia
1026	ISO8859_9	Turco/Turquía
1027	Cp943C	Parte SBCS del CCSID mixto EBCDIC japonés (CCSID 5035)
1097	Cp1098	Persa
1112	Cp921	Lituano/Lituania, letón/Letonia, báltico
1388	GBK	CCSID mixto EBCDIC para chino simplificado (se incluye GBK)
5026	Cp943C	CCSID mixto EBCDIC japonés (katakana ampliado)
5035	Cp943C	CCSID mixto EBCDIC japonés (latino ampliado)
8612	Cp1046	Árabe (grafías básicas únicamente) (o ASCII 420 y 8859_6)
9030	Cp874	Tailandés (SBCS ampliado de sistema principal)
13124	GBK	Parte SBCS del CCSID mixto EBCDIC para chino simplificado (se incluye GBK)
28709	Cp948	Parte SBCS del CCSID mixto EBCDIC para chino tradicional (CCSID 937)

Ejemplos: Crear un programa Java internacionalizado

Si necesita personalizar un programa Java(TM) para una región concreta del mundo, puede crear un programa Java internacionalizado con los Entornos nacionales Java.

Crear un programa Java internacionalizado implica diversas tareas:

1. Aísle los datos y el código sensibles al entorno nacional. Por ejemplo, las series, las fechas y los números del programa.
2. Establezca u obtenga el entorno nacional con la clase Locale.
3. Formatee las fechas y los números con el fin de especificar un entorno nacional si no se quiere utilizar el entorno nacional por omisión.
4. Cree paquetes de recursos para manejar las series y demás datos sensibles al entorno nacional.

Revise los siguientes ejemplos, que ofrecen maneras de ayudarle a completar las tareas necesarias para crear un programa Java internacionalizado:

- Ejemplo: Internacionalización de las fechas utilizando la clase java.util.DateFormat
- Ejemplo: Internacionalización de la presentación numérica utilizando la clase java.util.NumberFormat
- Ejemplo: Internacionalización de los datos específicos de entorno nacional utilizando la clase java.util.ResourceBundle

Para obtener más información sobre la internacionalización, consulte lo siguiente:

- Globalización OS/400
- Documentación sobre internacionalización de Sun Microsystems, Inc.

Compatibilidad entre releases

Los archivos de clase Java son compatibles con los releases posteriores (JDK 1.1.x a 1.2.x a 1.3.x a 1.4.x) siempre y cuando no utilicen algunas características cuyo soporte ha sido eliminado o modificado por Sun (consulte la documentación de Sun). Consulte The Source for Java Technology java.sun.com

para obtener información acerca de la disponibilidad entre releases.

Cuando en un servidor iSeries se optimizan los programas Java mediante el mandato Crear programa Java (CRTJVAPGM), se conecta un programa Java (JVAPGM) al archivo de clase. La estructura interna de estos JVAPGM ha variado en V4R4. Esto implica que los JVAPGM creados antes de V4R4 no son válidos en V4R4 ni en los releases posteriores. Debe volver a crear los JVAPGM o el sistema creará automáticamente un JVAPGM con el mismo nivel de optimización que el que tenía antes. Sin embargo, es aconsejable emitir manualmente un mandato CRTJVAPGM, especialmente en el caso de los archivos JAR o ZIP. Así obtendrá la mejor optimización con el menor tamaño de programa.

Para obtener el mejor rendimiento en el nivel de optimización 40, es aconsejable emitir el mandato CRTJVAPGM en cada release de OS/400 o en cada cambio de versión de JDK. Esto es especialmente aplicable si se utiliza el recurso JDKVER en CRTJVAPGM, ya que ello provocaría la incorporación de los métodos de JDK de Sun en JVAPGM. Esta estrategia puede ser muy ventajosa para el rendimiento. No obstante, si en releases ulteriores se hiciesen cambios en JDK que invalidasen las incorporaciones mencionadas, los programas podrían ejecutarse en realidad con mayor lentitud que en las optimizaciones inferiores. Ello se debe a que sería necesario ejecutar código de caso especial para obtener un funcionamiento adecuado.

Consulte el apartado Rendimiento de ejecución Java para obtener información más detallada sobre el rendimiento.

Acceso a base de datos con IBM Developer Kit para Java

Con IBM Developer Kit para Java, los programas Java pueden acceder a los archivos de base de datos de tres formas:

- En la sección Controlador JDBC se explica cómo el controlador JDBC de IBM Developer Kit para Java permite a los programas Java acceder a los archivos de base de datos.
- En la sección Soporte SQLJ se explica cómo IBM Developer Kit para Java permite utilizar sentencias SQL intercaladas en la aplicación Java.
- La sección Rutinas SQL Java describe cómo puede utilizar procedimientos almacenados Java y funciones definidas por usuario Java para acceder a programas Java.

Acceder a la base de datos de iSeries con el controlador JDBC de IBM Developer Kit para Java

El controlador JDBC de IBM Developer Kit para Java, denominado también controlador "nativo", proporciona acceso programático a los archivos de base de datos de iSeries. Si se emplea la API de JDBC (Java Database Connectivity), las aplicaciones escritas en el lenguaje Java pueden acceder a las funciones de base de datos de JDBC con lenguaje de consulta estructurada (SQL) intercalado, ejecutar sentencias SQL, recuperar resultados y propagar los cambios de nuevo a la base de datos. La API de JDBC también puede utilizarse para interactuar con varios orígenes de datos en un entorno distribuido heterogéneo.

La CLI (Command Language Interface) SQL99, en la que se basa la API de JDBC, es la base de ODBC. JDBC proporciona una correlación natural y de fácil utilización desde el lenguaje de programación Java a las abstracciones y conceptos definidos en el estándar SQL.

Para utilizar el controlador JDBC, consulte lo siguiente:

Iniciación a JDBC
Puede seguir el ejercicio de aprendizaje consistente en escribir un programa JDBC y ejecutarlo en el servidor iSeries.

Conexiones
Un programa de aplicación puede tener varias conexiones simultáneas. Puede representar una conexión con un origen de datos en JDBC mediante un objeto Connection. Es a través de objetos Connection como se crean objetos Statement para procesar sentencias SQL en la base de datos.

Propiedades de la JVM
Algunos valores utilizados por el controlador JDBC nativo no pueden establecerse utilizando una propiedad de conexión. Estos valores deben establecerse para la JVM en la que se ejecuta el controlador JDBC nativo.

DatabaseMetaData
La interfaz DatabaseMetaData la utilizan principalmente los servidores de aplicaciones y las herramientas para determinar cómo hay que interaccionar con un origen de datos dado. Las aplicaciones también pueden servirse de los métodos de DatabaseMetaData para obtener información sobre un origen de datos específico.

Excepciones
El lenguaje Java utiliza excepciones para proporcionar posibilidades de manejo de errores para sus programas. Una excepción es un evento que se produce cuando se ejecuta el programa de forma que interrumpe el flujo normal de instrucciones.

Transacciones
Una transacción es una unidad lógica de trabajo. Las transacciones se utilizan para proporcionar integridad de los datos, una semántica correcta de la aplicación y una vista coherente de los datos durante el acceso concurrente. Todos los controladores compatibles con JDBC deben dar soporte a las transacciones.

Tipos de sentencia
La interfaz Statement y sus subclases PreparedStatement y CallableStatement se utilizan para procesar mandatos SQL en la base de datos. Las sentencias SQL provocan la generación de objetos ResultSet.

ResultSets
La interfaz ResultSet proporciona acceso a los resultados generados al ejecutar consultas. Los datos de un ResultSet pueden considerarse como una tabla con un número específico de columnas y un número específico de filas. Por omisión, las filas de la tabla se recuperan por orden. Dentro de una fila, se puede acceder a los valores de columna en cualquier orden.

Agrupación de objetos JDBC
Debido a que la creación de muchos objetos utilizados en JDBC es costosa, como por ejemplo objetos Connection, Statement y ResultSet, pueden obtenerse ventajas significativas de rendimiento

utilizando la agrupación de objetos JDBC. Mediante la agrupación de objetos, puede reutilizar estos objetos en lugar de crearlos cada vez que los necesita.

Actualizaciones por lotes
El soporte de actualización por lotes permite pasar muchas actualizaciones de la base de datos como una sola transacción entre el programa de usuario y la base de datos. Las actualizaciones por lotes pueden mejorar significativamente el rendimiento cuando deben realizarse muchas actualizaciones simultáneamente.

Tipos de datos avanzados
Existen varios tipos de datos nuevos denominados tipos de datos SQL3, que se suministran en la base de datos de iSeries. Los tipos de datos SQL3 proporcionan un enorme nivel de flexibilidad. Son perfectos para almacenar objetos Java serializados, documentos XML (Extensible Markup Language) y datos multimedia, como por ejemplo, canciones, imágenes de producto, fotografías de empleados y clips de vídeo. Los tipos de datos SQL3 son los siguientes:

- Tipos distintivos (distinct)
- Objetos grandes, como por ejemplo objetos grandes de tipo binario, objetos grandes de tipo carácter y objetos grandes de tipo carácter de doble byte
- Enlaces de datos (Datalinks)

RowSets
La especificación RowSet está diseñada más como infraestructura que como implementación real. Las interfaces RowSet definen un conjunto de funciones centrales que comparten todos los RowSets.

Transacciones distribuidas
La API de transacción Java (JTA) tiene soporte para transacciones complejas. También proporciona soporte para desasociar transacciones de objetos Connection. JTA y JDBC funcionan conjuntamente para desasociar transacciones de objetos Connection, lo cual permite que una sola conexión trabaje en varias transacciones simultáneamente. A la inversa, permite que varias conexiones trabajen en una sola transacción.

Consejos sobre rendimiento
Con estos consejos sobre rendimientos, podrá obtener el mejor rendimiento posible de las aplicaciones JDBC.

Para obtener más información acerca de JDBC, consulte la documentación relativa a JDBC

de Sun Microsystems, Inc.

Para obtener más información sobre el controlador JDBC nativo de iSeries, consulte Preguntas más frecuentes sobre el controlador JDBC nativo de iSeries

.

Iniciación a JDBC

El controlador Java Database Connectivity (JDBC) suministrado con Developer Kit para Java se denomina controlador JDBC de Developer Kit para Java. Este controlador también se conoce comúnmente como controlador JDBC nativo.

Para seleccionar el controlador JDBC que se ajuste a sus necesidades, tenga en cuenta las siguientes sugerencias:

- Los programas que se ejecutan directamente en un servidor en el que reside la base de datos deben utilizar el controlador JDBC nativo a efectos de rendimiento. Esto incluye la mayoría de las soluciones de servlets y JavaServer Pages (JSP) y las aplicaciones escritas para ejecutarse localmente en un servidor iSeries.
- Los programas que deben conectarse a un servidor iSeries remoto utilizan el controlador JDBC de IBM Toolbox para Java. El controlador JDBC de IBM Toolbox para Java es una implementación robusta de JDBC y se proporciona como parte de IBM Toolbox para Java. Aún siendo Java puro, el controlador JDBC de IBM Toolbox para Java es fácil de configurar para los clientes y requiere poca configuración del servidor.
- Los programas que se ejecutan en un servidor iSeries y necesitan conectarse a una base de datos remota no iSeries utilizan el controlador JDBC nativo y deben configurar una conexión DRDA (Distributed Relational Database Architecture) con ese servidor remoto.

Para empezar a trabajar con JDBC, consulte los siguientes apartados:

Tipos de controladores JDBC
Este tema define los tipos de controladores JDBC. Los tipos de controladores se definen en categorías según la tecnología utilizada para conectarse a la base de datos.

Requisitos
Este tema indica los requisitos necesarios para acceder a los siguientes elementos:

- JDBC central
- Paquete opcional de JDBC 2.0
- API de transacción Java (JTA)

Ejercicio de aprendizaje de JDBC
Este es un primer paso importante para aprender a escribir un programa JDBC y ejecutarlo en un servidor iSeries con el controlador JDBC nativo.

Tipos de controladores JDBC: Este tema define los tipos de controladores JDBC (Java Database Connectivity). Los tipos de controladores se definen en categorías según la tecnología utilizada para conectarse a la base de datos. Los proveedores de controladores JDBC utilizan estos tipos para describir cómo operan sus productos. Algunos tipos de controladores JDBC son más adecuados que otros para algunas aplicaciones.

Tipo 1: Los controladores de tipo 1 son controladores ″puente″. Utilizan otra tecnología, como por ejemplo, ODBC (Open Database Connectivity), para comunicarse con la base de datos. Esto representa una ventaja, ya que existen controladores ODBC para muchas plataformas RDBMS (sistemas de gestión de bases de datos relacionales). La interfaz Java nativa (JNI) se utiliza para llamar a las funciones ODBC desde el controlador JDBC.

Un controlador de tipo 1 debe tener el controlador puente instalado y configurado para poder utilizar JDBC con él. Esto puede representar un grave inconveniente para una aplicación de producción. Los controladores de tipo 1 no pueden utilizarse en un applet, ya que los applets no pueden cargar código nativo.

Tipo 2: Los controladores de tipo 2 utilizan una API nativa para comunicarse con un sistema de base de datos. Se utilizan métodos nativos Java para llamar a las funciones de la API que realizan las operaciones de base de datos. Los controladores de tipo 2 son generalmente más rápidos que los controladores de tipo 1.

Los controladores de tipo 2 necesitan tener instalado y configurado código binario nativo para funcionar. Un controlador de tipo 2 siempre utiliza JNI. Los controladores de tipo 2 no pueden utilizarse en un

applet, ya que los applets no pueden cargar código nativo. Un controlador JDBC de tipo 2 puede requerir la instalación de algún software de red DBMS (sistema de gestión de bases de datos).

El controlador JDBC de Developer Kit para Java es un controlador JDBC de tipo 2.

Tipo 3: Estos controladores utilizan un protocolo de red y middleware para comunicarse con un servidor. A continuación, el servidor convierte el protocolo a llamadas de función DBMS específicas de DBMS.

Los controladores de tipo 3 son la solución JDBC más flexible, ya que no requieren ningún código binario nativo en el cliente. Un controlador de tipo 3 no necesita ninguna instalación de cliente.

Tipo 4: Un controlador de tipo 4 utiliza Java para implementar un protocolo de red de proveedor DBMS. Puesto que los protocolos son generalmente de propiedad, los proveedores DBMS son generalmente las únicas empresas que suministran un controlador JDBC de tipo 4.

Los controladores de tipo 4 son todos ellos controladores Java. Esto significa que no existe ninguna instalación ni configuración de cliente. Sin embargo, un controlador de tipo 4 puede no ser adecuado para algunas aplicaciones si el protocolo subyacente no maneja adecuadamente cuestiones tales como la seguridad y la conectividad de red.

El controlador JDBC de IBM Toolbox para Java es un controlador JDBC de tipo 4, lo cual indica que la API es un controlador de protocolo de red Java puro.

Requisitos de JDBC: Antes de escribir y desplegar las aplicaciones JDBC, puede que sea necesario instalar los siguientes elementos:

- "JDBC central"
- "Paquete opcional de JDBC 2.0"
- "API de transacción Java" en la página 41

JDBC central:

Todo el soporte para el acceso del JDBC (Java™ Database Connectivity) central a la base de datos local está incorporado y preinstalado. Para poder establecer una conexión JDBC puede ser necesaria una mínima labor de configuración. El CCSID (identificador de juego de caracteres codificado)del trabajo en el que se está ejecutando la JVM (máquina virtual Java) debe configurarse para que se ejecute en un valor distinto de 65535. Esto puede realizarse cambiando el valor del sistema QCCSID, el perfil de usuario del trabajo de la JVM o mediante cualquier otro procedimiento que cambie el CCSID del trabajo.

Paquete opcional de JDBC 2.0: Si necesita utilizar las clases del paquete opcional de JDBC 2.0, debe incluir el archivo jdbc2_0-stdext.jar en la vía de acceso de clases. Este archivo de archivado Java (JAR) contiene todas las interfaces estándar necesarias para escribir la aplicación de forma que utilice el paquete opcional de JDBC 2.0. Para añadir el archivo JAR a la vía de acceso de clases de ampliaciones, cree un enlace simbólico desde el directorio de ampliaciones UserData a la ubicación del archivo JAR. Sólo es necesario realizar esta operación una vez; el archivo JAR del paquete opcional de JDBC 2.0 siempre está disponible para las aplicaciones durante la ejecución. Utilice el siguiente mandato para añadir el paquete opcional a la vía de acceso de clases de ampliaciones:

```
    ADDLNK OBJ('/QIBM/ProdData/OS400/Java400/ext/jdbc2_0-stdext.jar')
NEWLNK('/QIBM/UserData/Java400/ext/jdbc2_0-stdext.jar')
```

Nota: Este requisito sólo afecta a J2SDK 1.3. Dado que J2SDK 1.4 es el primer release con soporte JDBC 3.0, la totalidad de JDBC (es decir, el JDBC central y el paquete opcional) se traslada al archivo JAR de la ejecución básica de J2SDK que el programa siempre busca.

API de transacción Java: Si necesita utilizar la API de transacción Java (JTA) en la aplicación, debe incluir el archivo jta-spec1_0_1.jar en la vía de acceso de clases. Este archivo JAR contiene todas las interfaces estándar necesarias para escribir la aplicación de forma que utilice JTA. Para añadir el archivo JAR a la vía de acceso de clases de ampliaciones, cree un enlace simbólico desde el directorio de ampliaciones UserData a la ubicación del archivo JAR. Esta operación se realiza una sola vez y, una vez completada, el archivo JAR de JTA siempre está disponible para la aplicación durante la ejecución. Utilice el siguiente mandato para añadir JTA a la vía de acceso de clases de ampliaciones:

```
    ADDLNK OBJ('/QIBM/ProdData/OS400/Java400/ext/jta-spec1_0_1.jar')
NEWLNK('/QIBM/UserData/Java400/ext/jta-spec1_0_1.jar')
```

Compatibilidad de JDBC: El controlador JDBC nativo es compatible con todas las especificaciones JDBC relevantes. El nivel de compatibilidad del controlador JDBC no depende del release de OS/400, sino del release de JDK que utilice. A continuación se ofrece una lista del nivel de compatibilidad del controlador JDBC nativo para las diversas versiones de JDK:

Release de J2SDK	**Nivel de compatibilidad del controlador JDBC**
JDK 1.1	Esta versión de JDK es compatible con JDBC 1.0.
JDK 1.2	Esta versión de JDK es compatible con JDBC 2.0 y soporta el paquete opcional de JDBC 2.1.
JDK 1.3	Esta versión de JDK es compatible con JDBC 2.0 y soporta el paquete opcional de JDBC 2.1 (no existen cambios relacionados con JDBC en JDK 1.3).
JDK 1.4	Esta versión de JDK es compatible con JDBC 3.0, pero el paquete opcional de JDBC ya no existe (actualmente, su soporte forma parte de JDK central).

Ejercicio de aprendizaje de JDBC: A continuación se ofrece un ejercicio de aprendizaje para practicar la escritura de un programa JDBC (Java Database Connectivity) y su ejecución en un servidor iSeries con el controlador JDBC nativo. Está diseñado para mostrarle los pasos básicos necesarios para que el programa ejecute JDBC.

El programa de ejemplo crea una tabla y la llena con algunos datos. El programa procesa una consulta para obtener esos datos de la base de datos y visualizarlos en la pantalla.

Ejecutar el programa de ejemplo: Para ejecutar el programa de ejemplo, lleve a cabo los siguientes pasos:

1. Copie el programa en la estación de trabajo.
   a. Copie el programa de ejemplo y péguelo en un archivo de la estación de trabajo.
   b. Guarde el archivo con el mismo nombre que la clase pública suministrada y con la extensión .java. En este caso, el nombre del archivo debe ser BasicJDBC.java en la estación de trabajo local.
2. Transfiera el archivo desde la estación de trabajo al servidor iSeries. Desde un indicador de mandatos, especifique los siguientes mandatos:

```
    ftp <nombre servidor iSeries>
    <Especifique el ID de usuario>
    <Especifique la contraseña>
cd /home/cujo
put BasicJDBC.java
quit
```

   Para que estos mandatos funcionen, debe tener un directorio en el que colocar el archivo. En el ejemplo, la ubicación es /home/cujo, pero puede utilizar la ubicación que desee.

   Nota: es posible que los mandatos FTP mencionados anteriormente sean diferentes en función de la configuración del servidor, pero deben ser parecidos. La forma de transferir el archivo al servidor iSeries no tiene importancia, siempre y cuando se transfiera al sistema de archivos integrado. Herramientas tales como VisualAge para Java pueden automatizar por completo este proceso.

3. Asegúrese de establecer la vía de acceso de clases en el directorio en el que ha colocado el archivo, para que los mandatos Java encuentren el archivo al ejecutarlos. Desde una línea de mandatos CL, puede utilizar el mandato WRKENVVAR para ver las variables de entorno establecidas para el perfil de usuario.
   - Si observa una variable de entorno denominada CLASSPATH, debe asegurarse de que la ubicación en la que ha colocado el archivo .java se encuentra en la serie de directorios indicados allí, o añádala si la ubicación no se ha especificado.
   - Si no existe ninguna variable de entorno CLASSPATH, debe añadir una. Esta operación puede realizarse con el siguiente mandato:

     ```
     ADDENVVAR ENVVAR(CLASSPATH) VALUE('/home/cujo:/QIBM/ProdData/Java400/jdk13/lib/tools.jar')
     ```

   Nota: para compilar código Java desde el mandato CL, debe incluir el archivo tools.jar. Este archivo JAR incluye el mandato javac.
4. Compile el archivo Java en un archivo de clase.
   Especifique el siguiente mandato desde la línea de mandatos CL:

   ```
   java class(com.sun.tools.javac.Main) prop(BasicJDBC)
java BasicJDBC
   ```

   También puede compilar el archivo Java desde QSH:

   ```
   cd /home/cujo
javac BasicJDBC.java
   ```

   QSH se asegura automáticamente de que el archivo tools.jar pueda encontrarse. En consecuencia, no es necesario añadirlo a la vía de acceso de clases. El directorio actual también se encuentra en la vía de acceso de clases. Al emitir el mandato cambiar directorio (cd), también se encuentra el archivo BasicJDBC.java.

   Nota: también puede compilar el archivo en la estación de trabajo y utilizar FTP para enviar el archivo de clase al servidor iSeries en modalidad binaria. Este es un ejemplo de la capacidad de Java para ejecutarse en cualquier plataforma. Ejecute el programa Java mediante el siguiente mandato desde la línea de mandatos CL o desde QSH:

   ```
   java BasicJDBC
   ```

   La salida es la siguiente:

   ```
   ----------------------
   | 1 | Frank Johnson  |
   |                    |
   | 2 | Neil Schwartz  |
   |                    |
   | 3 | Ben Rodman     |
   |                    |
   | 4 | Dan Gloore     |
   ----------------------
   Se devuelven 4 filas.
   Salida completada.
   Programa Java completado.
   ```

Consultas: Para obtener más información acerca de Java y JDBC, consulte los siguientes recursos:

- Sitio Web externo del controlador JDBC nativo

- Sitio Web externo del controlador JDBC de IBM Toolbox para Java

- Página JDBC de Sun

- Foro de Java/JDBC para iSeries y usuarios de iSeries

- Dirección de correo electrónico de IBM JDBC

Utilizar JNDI para los ejemplos: Los DataSources trabajan mano a mano con JNDI (Java Naming and Directory Interface). JNDI es una capa de abstracción Java para servicios de directorio, del mismo modo que JDBC (Java Database Connectivity) es una capa de abstracción para bases de datos. JNDI se utiliza con mayor frecuencia con LDAP (Lightweight Directory Access Protocol), pero también puede utilizarse con COS (CORBA Object Services), el registro RMI (Remote Method Invocation) de Java o el sistema de archivos subyacente. Esta utilización variada se lleva a cabo por medio de los diversos proveedores de servicios de directorio que convierten las peticiones JNDI comunes en peticiones de servicio de directorio específicas.

Java 2 SDK, v 1.3 incluye tres suministradores de servicio: el suministrador de servicio LDAP, el suministrador de servicio de denominación COS y el suministrador de servicio de registro RMI.

Nota: Tenga en cuenta que utilizar RMI puede resultar una tarea compleja. Antes de elegir RMI como solución, asegúrese de comprender las ramificaciones que puede conllevar. La página siguiente es un buen lugar para empezar a valorar RMI:

Invocación de método remoto (RMI) de Java

Los ejemplos de DataSource se han diseñado utilizando el proveedor de servicio del sistema de archivos JNDI. Si desea ejecutar los ejemplos suministrados, debe existir un proveedor de servicio JNDI.

Siga estas instrucciones para configurar el entorno para el proveedor de servicio del sistema de archivos:

1. Baje el soporte JNDI del sistema de archivos del sitio de JNDI

   de Sun Microsystems.
2. Transfiera (utilizando FTP u otro mecanismo) fscontext.jar y providerutil.jar al sistema y colóquelos en /QIBM/UserData/Java400/ext. Este es el directorio de extensiones, y los archivos JAR que coloque en él se encontrarán automáticamente cuando ejecute la aplicación (es decir, no es necesario especificarlos en la vía de acceso de clases).

Una vez que tiene soporte de un proveedor de servicio para JNDI, debe configurar la información de contexto para las aplicaciones. Esta acción puede realizarse colocando la información necesaria en un archivo SystemDefault.properties. Existen varios lugares del sistema en los que puede especificar propiedades por omisión, pero lo mejor es crear un archivo de texto denominado SystemDefault.properties en el directorio local (es decir, en /home/).

Para crear un archivo, utilice las siguientes líneas o añádalas al archivo existente:

```
# Valores de entorno necesarios para JNDI.
java.naming.factory.initial=com.sun.jndi.fscontext.RefFSContextFactory
java.naming.provider.url=file:/DataSources/jdbc
```

Estas líneas especifican que el proveedor de servicio del sistema de archivos maneja las peticiones JNDI y que /DataSources/jdbc es el directorio raíz para las tareas que utilizan JNDI. Puede cambiar esta ubicación, pero el directorio que especifique debe existir. La ubicación que especifique será el lugar de enlace y despliegue de los DataSources de ejemplo.

Conexiones

El objeto Connection representa una conexión con un origen de datos en Java Database Connectivity (JDBC). Los objetos Statement se crean a través de objetos Connection para procesar sentencias SQL en la

base de datos. Un programa de aplicación puede tener varias conexiones simultáneas. Estos objetos Connection puede conectarse todos a la misma base de datos o a bases de datos diferentes.

La obtención de una conexión en JDBC puede realizarse de dos maneras:
- Mediante la clase DriverManager.
- Utilizando DataSources.

Es preferible utilizar DataSources para obtener una conexión, ya que mejora la portabilidad y la capacidad de mantenimiento de las aplicaciones. También permite que una aplicación utilice de forma transparente las agrupaciones de conexiones y sentencias y las transacciones distribuidas.

Para obtener detalles acerca de la obtención de conexiones, consulte las siguientes secciones:

DriverManager
DriverManager es una clase estática que gestiona el conjunto de controladores JDBC disponibles para que los utilice una aplicación.

Propiedades de conexión
En la tabla figuran las propiedades válidas de conexión para el controlador JDBC, sus valores y descripciones.

Utilizar DataSources con UDBDataSource
Puede desplegar un DataSource con la clase UDBDataSource configurándolo de forma que tenga propiedades específicas y, a continuación, enlazándolo con algún servicio de directorio mediante la utilización de JNDI (Java Naming and Directory Interface).

Propiedades de DataSource
En la tabla figuran las propiedades válidas de DataSource, sus valores y descripciones.

Otras implementaciones de DataSource
Existen otras implementaciones de la interfaz DataSource suministradas con el controlador JDBC nativo. Sólo existen como puente hasta que se adopten UDBDataSource y sus funciones relacionadas.

Una vez que se ha obtenido una conexión, puede utilizarse para realizar las siguientes tareas de JDBC:
- "Crear sentencias" en la página 81 para interactuar con la base de datos.
- Controlar transacciones en la base de datos.
- Recuperar metadatos acerca de la base de datos.

DriverManager: DriverManager es una clase estática en Java 2 Software Development Kit (J2SDK). DriverManager gestiona el conjunto de controladores Java Database Connectivity (JDBC) que están disponibles para que los utilice una aplicación. Las aplicaciones pueden utilizar varios controladores JDBC simultáneamente si es necesario. Cada aplicación especifica un controlador JDBC mediante la utilización de un URL (Localizador universal de recursos). Pasando un URL de un controlador JDBC específico a DriverManager, la aplicación informa a DriverManager acerca del tipo de conexión JDBC que debe devolverse a la aplicación.

Para poder realizar esta operación, DriverManager debe estar al corriente del los controladores JDBC disponibles para que pueda distribuir las conexiones. Efectuando una llamada al método Class.forName, carga una clase en la máquina virtual Java (JVM) que se está ejecutando en función del nombre de serie que se pasa en el método. A continuación figura un ejemplo del método class.forName utilizado para cargar el controlador JDBC nativo:

Ejemplo: cargar el controlador JDBC nativo

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
// Cargar el controlador JDBC nativo en DriverManager para hacerlo
// disponible para peticiones getConnection.

Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
```

Los controladores JDBC están diseñados para informar a DriverManager acerca de sí mismos automáticamente cuando se carga su clase de implementación de controlador. Una vez que se ha procesado la línea de código mencionada anteriormente, el controlador JDBC nativo está disponible para la DriverManager con la que debe trabajar. La línea de código siguiente solicita un objeto Connection que utiliza el URL de JDBC nativo:

Ejemplo: solicitar un objeto Connection

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
// Obtener una conexión que utiliza el controlador JDBC nativo.

Connection c = DriverManager.getConnection("jdbc:db2:*local");
```

La forma más sencilla de URL JDBC es una lista de tres valores separados mediante dos puntos. El primer valor de la lista representa el protocolo, que es siempre `jdbc` para los URL JDBC. El segundo valor es el subprotocolo y se utiliza `db2` o `db2iSeries` para especificar el controlador JDBC nativo. El tercer valor es el nombre de sistema para establecer la conexión con un sistema específico. Existen dos valores especiales que pueden utilizarse para conectarse con la base de datos local. Son *LOCAL y localhost (ambos son sensibles a mayúsculas y minúsculas). También puede suministrarse un nombre de sistema específico, de la forma siguiente:

```
Connection c =
  DriverManager.getConnection("jdbc:db2:rchasmop");
```

Así se crea una conexión con el sistema rchasmop. Si el sistema al que intenta conectarse es un sistema remoto (por ejemplo, a través de Distributed Relational Database Architecture), debe utilizarse el nombre de sistema del directorio de bases de datos relacionales.

Notas:

- Si no se especifica lo contrario, el ID de usuario y la contraseña utilizados actualmente para iniciar la sesión también se utilizan para establecer la conexión con la base de datos.
-

  El controlador IBM DB2 JDBC Universal también utiliza el subprotocolo db2. Para asegurarse de que el controlador JDBC nativo puede manejar el URL, las aplicaciones deben utilizar el URL jdbc:db2iSeries:xxxx en lugar del URL jdbc:db2:xxxx. Si la aplicación no desea que el controlador nativo acepte URLS con el subprotocolo db2, la aplicación deberá cargar la clase com.ibm.db2.jdbc.app.DB2iSeriesDriver, en lugar de com.ibm.db2.jdbc.app.DB2Driver. Al cargarse esta clase, el controlador nativo ya no tiene que manejar los URL que contienen el subprotocolo db2.

Propiedades: El método DriverManager.getConnection toma un URL de una sola serie indicado anteriormente, y sólo es uno de los métodos de DriverManager destinado a obtener un objeto Connection. También existe otra versión del método DriverManager.getConnection que toma un ID de usuario y una contraseña. A continuación figura un ejemplo de esta versión:

Ejemplo: método DriverManager.getConnection que toma un ID de usuario y una contraseña

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
// Obtener una conexión que utiliza el controlador JDBC nativo.

Connection c = DriverManager.getConnection("jdbc:db2:*local", "cujo", "newtiger");
```

La línea de código intenta conectarse con la base de datos local como usuario cujo con la contraseña newtiger sin importar quién ejecuta la aplicación. También existe una versión del método DriverManager.getConnection que toma un objeto java.util.Properties que permite una mayor personalización. A continuación se ofrece un ejemplo:

Ejemplo: método DriverManager.getConnection que toma un objeto java.util.Properties

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
// Obtener una conexión que utiliza el controlador JDBC nativo.

Properties prop = new java.util.Properties();
prop.put("user", "cujo");
prop.put("password","newtiger");
Connection c = DriverManager.getConnection("jdbc:db2:*local", prop);
```

El código es funcionalmente equivalente a la versión mencionada anteriormente que ha pasado el ID de usuario y la contraseña como parámetros.

Consulte las Propiedades de Connection para obtener una lista completa de las propiedades de conexión del controlador JDBC nativo.

Propiedades de URL: Otra forma de especificar propiedades es colocarlas en una lista del propio objeto URL. Cada propiedad de la lista está separada mediante un signo de punto y coma, y la lista debe tener el formato `nombre propiedad=valor propiedad`. Sólo existe un método abreviado que no cambia significativamente la forma en que se realiza el proceso, como muestra el ejemplo siguiente:

Ejemplo: especificar propiedades de URL

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
// Obtener una conexión que utiliza el controlador JDBC nativo.

Connection c = DriverManager.getConnection("jdbc:db2:*local;user=cujo;password=newtiger");
```

De nuevo, el código es funcionalmente equivalente a los ejemplos mencionados anteriormente.

Si se especifica un valor de propiedad tanto en un objeto de propiedades como en el objeto URL, la versión de URL tiene preferencia sobre la versión del objeto de propiedades. A continuación se ofrece un ejemplo:

Ejemplo: propiedades de URL

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
// Obtener una conexión que utiliza el controlador JDBC nativo.
Properties prop = new java.util.Properties();
prop.put("user", "someone");
prop.put("password","something");
Connection c = DriverManager.getConnection("jdbc:db2:*local;user=cujo;password=newtiger",
prop);
```

El ejemplo utiliza el ID de usuario y la contraseña de la serie de URL en lugar de la versión del objeto Properties. Termina siendo funcionalmente equivalente al código mencionado anteriormente.

Consulte los siguientes ejemplos para obtener más información:

- Utilizar JDBC nativo y JDBC de IBM Toolbox para Java de forma concurrente
- Propiedad Access
- ID de usuario y contraseña no válidos

Propiedades de conexión: En esta tabla figuran las propiedades válidas de conexión para el controlador JDBC, los valores que tienen y sus descripciones:

Propiedad	Valores	Significado
access (acceso)	all, read call, read only	Este valor permite restringir el tipo de operaciones que se pueden realizar con una determinada conexión. El valor por omisión es ″all″, que básicamente significa que la conexión tiene pleno acceso a la API de JDBC. El valor ″read call″ (llamada de lectura) solo permite que la conexión haga consultas y llame a procedimientos almacenados. Se impide todo intento de actualizar la base de datos con una sentencia SQL. El valor solo de lectura ″read only″ permite restringir una conexión para que solo pueda hacer consultas. Se impiden las llamadas a procedimientos almacenados y las sentencias de actualización.
 auto commit (compromiso automático)	true, false	Este valor se utiliza para establecer el compromiso automático de la conexión. El valor por omisión es true a menos que se haya establecido la propiedad de aislamiento de transacción con un valor distino a ninguno. En ese caso, el valor por omisión es false. 
batch style (estilo de lotes)	2.0, 2.1	La especificación JDBC 2.1 define un segundo método para manejar las excepciones de una actualización por lotes. El controlador puede ajustarse a cualquiera de ellos. El valor por omisión es trabajar según lo definido en la especificación JDBC 2.0.

Propiedad	Valores	Significado
block size (tamaño de bloque)	0, 8, 16, 32, 64, 128, 256, 512	Este es el número de filas que se extraen de una sola vez para un conjunto de resultados. En un proceso habitual sólo hacia adelante de un conjunto de resultados, se obtiene un bloque de este tamaño. Entonces no es necesario acceder a la base de datos ya que la aplicación procesa cada fila. La base de datos sólo solicitará otro bloque de datos cuando se haya llegado al final del bloque. Este valor solo se utiliza si la propiedad de habilitado para bloques (blocking enabled) se establece como true. Establecer la propiedad de tamaño de bloque en 0 tiene el mismo efecto que establecer la propiedad de habilitado para bloques como false. El valor por omisión es utilizar la agrupación en bloques con un tamaño de bloque de 32. Actualmente, esta decisión es completamente arbitraria, por lo que el valor por omisión podría cambiar en el futuro. Actualmente, la agrupación en bloques no se utiliza en los conjuntos de resultados desplazables.
blocking enabled (habilitado para bloques)	true, false	Este valor se utiliza para determinar si la conexión debe utilizar o no la agrupación en bloques en la recuperación de filas de conjuntos de resultados. La agrupación en bloques puede aumentar notablemente el rendimiento al procesar conjuntos de resultados. Por omisión, esta propiedad está establecida en true.

Propiedad	Valores	Significado
cursor hold (retención de cursor)	true, false	Este valor especifica si los conjuntos de resultados deben permanecer abiertos cuando se compromete una transacción. El valor true indica que una aplicación puede acceder a sus conjuntos de resultados abiertos una vez llamado el compromiso. El valor false indica que el compromiso cierra los cursores abiertos en la conexión. Por omisión, esta propiedad está establecida en true. Este valor de propiedad funciona como valor por omisión para todos los conjuntos de resultados establecidos para la conexión. Con el soporte de retención de cursor añadido en JDBC 3.0, este valor por omisión se sustituye simplemente si una aplicación especifica posteriormente una capacidad de retención diferente.
data truncation (truncamiento de datos)	true, false	Este valor especifica si el truncamiento de datos de tipo carácter debe generar avisos y excepciones (true) o si los datos deben truncarse de forma silenciosa (false). Si el valor por omisión es true, debe aceptarse el truncamiento de datos en los campos de caracteres.
date format (formato de fecha)	juliano, mdy, dmy, ymd, usa, iso, eur, jis	Esta propiedad permite cambiar el formato de las fechas.
date separator (separador de fecha)	/(barra inclinada), -(guión), .(punto), ,(coma), blanco	Esta propiedad permite cambiar el separador de fecha. Sólo es válida en combinación con algunos de los valores de dateFormat (según las normas del sistema).
separador decimal	.(punto), ,(coma)	Esta propiedad permite cambiar el separador de decimal.

Propiedad	Valores	Significado
hacer proceso de escape	true, false	Esta propiedad establece un distintivo que indica si las sentencias bajo la conexión deben hacer o no un proceso de escape. La utilización del proceso de escape es una manera de codificar las sentencias SQL para que sean genéricas y similares para todas las plataformas, pero luego la base de datos lee las cláusulas de escape y sustituye la debida versión específica del sistema para el usuario. Es una propiedad valiosa, salvo que implica hacer un trabajo adicional en el sistema. Si el usuario sabe que sólo va a utilizar sentencias SQL que ya contienen sintaxis SQL iSeries válida, es preferible establecer este valor en ″false″ para aumentar el rendimiento. El valor por omisión de esta propiedad es ″true″, ya que debe estar en conformidad con la especificación JDBC (es decir, el proceso de escape está activo por omisión). Este valor se ha añadido debido a una deficiencia de la especificación JDBC. Sólo se puede establecer que el proceso de escape está desactivado en la clase Statement. Eso funciona correctamente si se trata de sentencias simples. Basta con crear la sentencia, desactivar el proceso de escape y empezar a ejecutar las sentencias. Sin embargo, en el caso de las sentencias preparadas y de las sentencias invocables, este esquema no funciona. Se suministra la sentencia SQL en el momento de construir la sentencia preparada o la sentencia invocable y ésta no cambia después de ello. Así que la sentencia queda preparada en primer lugar y el hecho de cambiar el proceso de escape más adelante no tiene ningún significado. Gracias a esta propiedad de conexión, existe un modo de soslayar la actividad general adicional.
errores	basic, full	Esta propiedad permite devolver el texto de errores de segundo nivel de todo el sistema en mensajes de objeto SQLException. El valor por omisión es basic, que devuelve sólo el texto de mensaje estándar.

Propiedad	Valores	Significado
bibliotecas	Una lista de bibliotecas separadas mediante espacios. (Una lista de bibliotecas también puede separarse mediante signos de dos puntos o comas).	Esta propiedad permite colocar una lista de bibliotecas en la lista de bibliotecas del trabajo servidor o establecer una lista de bibliotecas específica. La propiedad de denominación, ″naming″, afecta al funcionamiento de esta propiedad. En el caso por omisión, en que ″naming″ está establecida en sql, JDBC funciona como ODBC. La lista de bibliotecas no tiene ningún efecto sobre el proceso que efectúa la conexión. Existe una biblioteca por omisión para todas las tablas no calificadas. Por omisión, la biblioteca tiene el mismo nombre que el perfil de usuario al que está conectado. Si se especifica la propiedad ″libraries″, la primera biblioteca de la lista pasa a ser la biblioteca por omisión. Si se especifica una biblioteca por omisión en el URL de conexión (como en jdbc:db2:*local/mibilioteca), se altera temporalmente cualquier valor de esta propiedad. Si ″naming″ se establece en ″system″, cada una de las bibliotecas especificadas para esta propiedad se añade a la parte del usuario de la lista de bibliotecas y se busca en la lista de bibliotecas para resolver las referencias de tabla no calificadas.

Propiedad	Valores	Significado
umbral de lob	Cualquier valor por debajo de 500000	Esta propiedad indica al controlador que coloque los valores reales en el almacenamiento de conjunto de resultados en lugar de localizadores de columnas lob si la columna lob es inferior al tamaño del umbral. Esta propiedad actúa sobre el tamaño de columna, no sobre el tamaño de los datos lob propiamente. Por ejemplo, si la columna lob está definida para contener hasta 1 MB para cada lob, pero todos los valores de columna están por debajo de 500 MB, se siguen utilizando localizadores. Tenga en cuenta que el límite de tamaño se establece de forma que permita extraer los bloques de datos sin el riesgo de que los bloques de datos crezcan más allá de los 16 MB de máximo de tamaño de asignación. En conjuntos de resultados mayores, sigue siendo fácil sobrepasar este límite, lo cual provoca anomalías en las extracciones. Debe tener cuidado en la forma en que la propiedad block size y esta propiedad interactúan con el tamaño de un bloque de datos. El valor por omisión es 0. Siempre se utilizan localizadores para datos lob.
 precisión máxima	31, 63	Este valor especifica la precisión máxima (longitud) que se devuelve para los tipos de datos de resultados. El valor por omisión es 31.
 escala máxima	0-63	Este valor especifica la escala máxima (número de posiciones decimales a la derecha de la coma decimal) que se devuelve para los tipos de datos de resultados. El valor puede ir de 0 a la precisión máxima. El valor por omisión es 31.

Propiedad	Valores	Significado
 escala de división mínima	0-9	Este valor especifica la escala de división mínima (número de posiciones decimales a la derecha de la coma decimal) que se devuelve para los tipos de datos intermedios y de resultados. El valor puede ir de 0 a 9, sin sobrepasar la escala máxima. Si se especifica 0, no se utiliza la escala de división mínima. El valor por omisión es 0. 
naming (denominación)	sql, system	Esta propiedad permite utilizar la sintaxis de denominación tradicional de iSeries o la sintaxis de denominación estándar de SQL. La denominación del sistema significa que utilizará un carácter /(barra inclinada) para separar los valores de colección y de tabla, y la denominación sql significa que utilizará un carácter .(punto) para separar los valores. El establecimiento de este valor tiene ramificaciones que afectan también a cuál es la biblioteca por omisión. Consulte la propiedad libraries (page 51) para obtener más información. El valor por omisión es utilizar la denominación sql.
contraseña	cualquier valor	Esta propiedad prevé la especificación de una contraseña para la conexión. Esta propiedad no funciona correctamente si no se especifica también la propiedad de usuario, "user". Estas propiedades permiten establecer conexiones con la base de datos en los casos en que el usuario no coincida con el que está ejecutando el trabajo de iSeries. Especificar las propiedades de usuario y contraseña tiene el mismo efecto que utilizar el método de conexión con la firma getConnection(String url, String userId, String password).

Propiedad	Valores	Significado
prefetch (preextraer)	true, false	Esta propiedad especifica si el controlador debe extraer los primeros datos de un conjunto de resultados inmediatamente después del proceso o esperar a que se soliciten los datos. Si el valor por omisión es true, deben preextraerse los datos. En las aplicaciones que utilizan el controlador JDBC nativo, esto rara vez representa un problema. La propiedad existe principalmente para uso interno con procedimientos almacenados Java y funciones definidas por usuario en las que es importante que el motor de bases de datos no extraiga ningún dato de los conjuntos de resultados en nombre del usuario antes de que éste lo solicite.
reutilizar objetos	true, false	Esta propiedad especifica si el controlador debe intentar reutilizar algunos tipos de objetos después de que el usuario los haya cerrado. Esto representa una mejora en el rendimiento. El valor por omisión es true.
rastreo de servidor	Una representación de serie de un entero	Esta propiedad habilita el rastreo del trabajo servidor JDBC. Si el rastreo del servidor está habilitado, el rastreo se inicia cuando el cliente se conecta al servidor y finaliza cuando termina la conexión. Los datos de rastreo se recogen en archivos en spool en el servidor. Pueden activarse varios niveles de rastreos del servidor combinados, añadiendo las constantes y pasando esa suma en el método set. **Nota:** el personal de soporte utiliza habitualmente esta propiedad y sus valores no se describen con más detalle.
formato de hora	hms, usa, iso, eur, jis	Esta propiedad permite cambiar el formato de los valores de hora.
separador de hora	:(dos puntos), .(punto), ,(coma), blanco	Esta propiedad permite cambiar el separador de hora. Sólo es válida en combinación con algunos de los valores de timeFormat (según las normas del sistema).

Propiedad	Valores	Significado
trace (rastreo)	true, false	Esta propiedad prevé la activación del rastreo de la conexión. Se puede utilizar como una simple ayuda para la depuración. Está previsto ampliar esta característica en el futuro. Consulte D2. El controlador JDBC ha lanzado una excepción. ¿Qué debo hacer? si desea más información sobre la depuración. El valor por omisión es ″false″, que corresponde a no utilizar el rastreo.
transaction isolation (aislamiento de transacciones)	none, read committed, read uncommitted, repeatable read, serializable	Esta propiedad permite al usuario establecer el nivel de aislamiento de transacción para la conexión. No hay ninguna diferencia entre establecer esta propiedad en un nivel concreto y especificar un nivel en el método setTransactionIsolation() de la interfaz Connection. El valor por omisión de esta propiedad es ″none″, El valor por omisión de esta propiedad es ″none″, ya que JDBC toma por omisión la modalidad de compromiso automático.
translate binary (convertir binario)	true, false	Esta propiedad puede utilizarse para obligar al controlador JDBC a que trate los valores de datos de tipo binary y varbinary como si fuesen valores de datos de tipo char y varchar. El valor por omisión de esta propiedad es ″false″, es decir, no tratar los datos de tipo binario como si fuesen datos de tipo carácter.
 translate hex (convertir hex)	binario, carácter	Este valor se utiliza para seleccionar el tipo de datos utilizado por las constantes hex en expresiones SQL. El valor binario indica que las constantes hex utilizarán el tipo de datos BINARY, que es una novedad de la V5R3. El valor carácter indica que las constantes hex utilizarán el tipo de datos CHARACTER FOR BIT DATA. El valor por omisión es carácter.

Propiedad	Valores	Significado
use block insert (utilizar inserción de bloques)	true, false	Esta propiedad permite al controlador JDBC nativo colocarse en modalidad de inserción de bloques para insertar bloques de datos en la base de datos. Esta es una versión optimizada de la actualización por lotes. Esta modalidad optimizada sólo puede utilizarse en aplicaciones que garanticen no transgredir determinadas restricciones del sistema ni producir anomalías de inserción de datos, pudiendo dañar los datos. Las aplicaciones que activen esta propiedad sólo deben conectarse al sistema local al intentar realizar actualizaciones por lotes. No deben utilizar DRDA para establecer conexiones remotas, ya que la inserción por bloques no puede gestionarse a través de DRDA. Las aplicaciones también deben asegurarse de que PreparedStatements con una sentencia SQL insert y una cláusula values indican todos los parámetros de valores de inserción. No se permiten contantes en la lista de valores. Este es un requisito del motor de inserción por bloques del sistema. El valor por omisión es false.
user (usuario)	cualquier valor	Esta propiedad permite especificar un ID de usuario para la conexión. Esta propiedad no funciona correctamente si no se especifica también la propiedad de contraseña, "password". Estas propiedades permiten establecer conexiones con la base de datos en los casos en que el usuario no coincida con el que está ejecutando el trabajo de iSeries. Especificar las propiedades de usuario y contraseña tiene el mismo efecto que utilizar el método de conexión con la firma getConnection(String url, String userId, String password).

Utilizar DataSources con UDBDataSource: Las interfaces DataSource se han diseñado para permitir una flexibilidad adicional en la utilización de controladores JDBC (Java Database Connectivity). La utilización de DataSources puede dividirse en dos fases:

- **Despliegue**
  El despliegue es una fase de configuración que se produce antes de que una aplicación JDBC se ejecute

realmente. El despliegue implica generalmente configurar un DataSource de forma que tenga propiedades específicas y, a continuación, enlazarlo con un servicio de directorio mediante la utilización de JNDI (Java Naming and Directory Interface). El servicio de directorio es generalmente LDAP (Lightweight Directory Access Protocol), pero también pueden utilizarse otros, como por ejemplo Servicios de objeto CORBA (Common Object Request Broker Architecture, RMI (Java Remote Method Invocation) o el sistema de archivos subyacente.

- **Utilización**
  Al desasociar el despliegue de la utilización de ejecución de DataSource, muchas aplicaciones pueden reutilizar la configuración de DataSource. Cambiando alguno de los aspectos del despliegue, todas las aplicaciones que utilizan ese DataSource recogen los cambios automáticamente.

Nota: Tenga en cuenta que utilizar RMI puede resultar una tarea compleja. Antes de elegir RMI como solución, asegúrese de comprender las ramificaciones que puede conllevar. La página siguiente es un buen lugar para empezar a valorar RMI:

Llamada de método remoto (RMI) de Java

Una de las ventajas de los DataSources es que permiten a los controladores JDBC efectuar el trabajo en nombre de la aplicación sin influir directamente sobre el proceso de desarrollo de la misma. Para obtener más información, consulte las secciones Agrupación de conexiones, Agrupación de sentencias y Transacciones distribuidas.

UDBDataSourceBind: El programa UDBDataSourceBind es un ejemplo de cómo crear un UDBDataSource y enlazarlo con JNDI. Este programa realiza todas las tareas básicas solicitadas. Es decir, crea una instancia de un objeto UDBDataSource, establece las propiedades de este objeto, recupera un contexto JNDI y enlaza el objeto con un nombre del contexto JNDI.

El código de despliegue es específico del proveedor. La aplicación debe importar la implementación específica de DataSource con la que desea trabajar. En la lista de importación, se importa la clase UDBDataSource calificada por paquete. La parte menos conocida de esta aplicación es el trabajo que se realiza con JNDI (por ejemplo, la recuperación del objeto Context y la llamada a bind). Para obtener más información, consulte JNDI

de Sun Microsystems, Inc.

Una vez que este programa se ha ejecutado y completado satisfactoriamente, existe una entrada nueva en un servicio de directorio JNDI denominada SimpleDS. Esta entrada se encuentra en la ubicación especificada por el contexto JNDI. Ahora, se despliega la implementación de DataSource. Un programa de aplicación puede utilizar este DataSource para recuperar conexiones de base de datos y trabajo relacionado con JDBC.

UDBDataSourceUse: El programa UDBDataSourceUse es un ejemplo de aplicación JDBC que utiliza la aplicación desplegada anteriormente.

La aplicación JDBC obtiene un contexto inicial al igual que hizo antes enlazando el UDBDataSource en el ejemplo anterior. A continuación, se utiliza el método lookup en ese contexto para devolver un objeto de tipo DataSource para que lo utilice la aplicación.

Nota: la aplicación de ejecución sólo está interesada en los métodos de la interfaz DataSource, y por tanto no es necesario que esté al corriente de la clase de implementación. Esto hace que la aplicación sea portable.

Suponga que UDBDataSourceUse es una aplicación compleja que ejecuta una operación de grandes dimensiones dentro de la empresa. En la empresa existen una docena o más de aplicaciones similares de

grandes dimensiones. Es necesario cambiar el nombre de uno de los sistemas de la red. Ejecutando una herramienta de despliegue y cambiando una sola propiedad de UDBDataSource, es posible conseguir este comportamiento nuevo en todas las aplicaciones sin cambiar el código de las mismas. Una de las ventajas de los DataSources es que permiten consolidar la información de configuración del sistema. Otra de las ventajas principales es que permiten a los controladores implementar funciones invisibles para la aplicación, como por ejemplo agrupación de conexiones, agrupación de sentencias y soporte para transacciones distribuidas.

Después de analizar detenidamente UDBDataSourceBind y UDBDataSourceUse, quizá se haya preguntado cómo es posible que el objeto DataSource sepa lo que debe hacer. No existe ningún código que especifique un sistema, un ID de usuario o una contraseña en ninguno de estos programas. La clase UDBDataSource tiene valores por omisión para todas las propiedades; por omisión, se conecta al servidor iSeries local con el perfil de usuario y la contraseña de la aplicación que se ejecuta. Si deseara asegurarse de que, en lugar de ello, la conexión se ha efectuado con el perfil de usuario cujo, podría hacerlo de dos maneras:

- Estableciendo el ID de usuario y la contraseña como propiedades de DataSource. Consulte el Ejemplo: crear un UDBDataSourceBind y establecer las propiedades de DataSource para saber cómo utilizar esta técnica.
- Utilizando el método getConnection de DataSource, que toma un ID de usuario y una contraseña durante la ejecución . Consulte el Ejemplo: crear un UDBDataSource y obtener un ID de usuario y una contraseña para saber cómo utilizar esta técnica.

Existen diversas propiedades que pueden especificarse para UDBDataSource, al igual que existen propiedades que pueden especificarse para las conexiones creadas con DriverManager. Consulte la sección Propiedades de DataSource para obtener una lista de las propiedades soportadas para el controlador JDBC nativo.

Aunque estas listas son similares, no es seguro que lo sean en releases futuros. Es aconsejable iniciar la codificación en la interfaz DataSource.

Nota: el controlador JDBC nativo también tiene otras dos implementaciones de DataSource, pero no es aconsejable su uso directo.

- DB2DataSource
- DB2StdDataSource

Propiedades de DataSource: Esta tabla contiene las propiedades de origen de datos válidas, sus valores y descripciones:

Método Set (tipo de datos)	Valores	Descripción
setAccess (String)	″all″, ″read call″, ″read only″	Esta propiedad puede utilizarse para restringir el tipo de operaciones que se pueden realizar con una determinada conexión. El valor por omisión es ″all″, que básicamente significa que la conexión tiene pleno acceso a la API de JDBC (Java Database Connectivity). El valor ″read call″ sólo permite que la conexión realice consultas y llame a procedimientos almacenados. Cualquier intento de actualizar la base de datos con una sentencia SQL provoca una SQLEXception. El valor ″read only″ restringe la conexión para que sólo pueda realizar consultas. Cualquier intento de procesar una llamada a procedimiento almacenado o sentencias de actualización provoca una SQLException.
setBatchStyle (String)	″2.0″, ″2.1″	La especificación JDBC 2.1 define un segundo método para manejar las excepciones de una actualización por lotes. El controlador puede ajustarse a cualquiera de ellos. El valor por omisión es trabajar según lo definido en la especificación JDBC 2.0.
setUseBlocking (boolean)	″true″, ″false″	Esta propiedad se utiliza para determinar si la conexión debe utilizar o no la agrupación en bloques en la recuperación de filas de conjuntos de resultados. La agrupación en bloques puede aumentar notablemente el rendimiento al procesar conjuntos de resultados. Por omisión, esta propiedad está establecida en true.

Método Set (tipo de datos)	Valores	Descripción
setBlockSize (int)	″0″, ″8″, ″16″, ″32″, ″64″, ″128″, ″256″, ″512″	Esta propiedad indica el número de filas que se extraen de una sola vez para un conjunto de resultados. En el proceso habitual sólo hacia adelante de un conjunto de resultados, se obtiene un bloque de este tamaño si la base de datos tiene suficientes filas para satisfacer la consulta. Sólo cuando se haya llegado al final del bloque en el almacenamiento interno del controlador JDBC se enviará otra petición de bloque de datos a la base de datos. Este valor sólo se utiliza si la propiedad useBlocking se establece en true. Consulte la sección acerca de setUseBlocking (page 59) para obtener más información. El valor ″0″ para la propiedad block size equivale a llamar a setUseBlocking(false). El valor por omisión es utilizar la agrupación en bloques de tamaño ″32″. Esta decisión es completamente arbitraria, por lo que el valor por omisión podría cambiar en futuros releases. Actualmente, la agrupación en bloques no se utiliza en los conjuntos de resultados desplazables. La utilización de la agrupación por bloques afecta al grado de sensibilidad de cursor de la aplicación del usuario. Un cursor sensible observa los cambios efectuados por otras sentencias SQL. Sin embargo, debido al almacenamiento intermedio de datos, los cambios sólo se detectan cuando es necesario extraer datos de la base de datos.

Método Set (tipo de datos)	Valores	Descripción
setCursorHold (boolean)	″true″, ″false″	Esta propiedad especifica si los conjuntos de resultados deben permanecer abiertos cuando se compromete una transacción. El valor true indica que una aplicación puede acceder a sus conjuntos de resultados abiertos una vez llamado el compromiso. El valor false indica que el compromiso cierra los cursores abiertos en la conexión. Por omisión, esta propiedad está establecida en true. Esta propiedad funciona como valor por omisión para todos los conjuntos de resultados establecidos para la conexión. Con el soporte de retención de cursor añadido en JDBC 3.0 (consulte la sección Características de ResultSet para obtener detalles), este valor por omisión se sustituye simplemente si una aplicación especifica posteriormente un soporte de cursor diferente.
setDataTruncation (boolean)	″true″, ″false″	Esta propiedad especifica lo siguiente: • Si el truncamiento de datos de tipo carácter debe generar avisos y excepciones (true) • Si los datos deben truncarse de forma silenciosa (false). Consulte la sección acerca de DataTruncation para obtener más detalles.
setDatabaseName (String)	Cualquier nombre	Esta propiedad especifica la base de datos a la que DataSource intenta conectarse. El valor por omisión es *LOCAL. El nombre de base de datos debe existir en el directorio de bases de datos relacionales del sistema que ejecuta la aplicación o ser el valor especial *LOCAL o localhost para especificar el sistema local.
setDataSourceName (String)	Cualquier nombre	Esta propiedad permite pasar un nombre JNDI (Java Naming and Directory Interface) ConnectionPoolDataSource para dar soporte a la agrupación de conexiones.
setDateFormat (String)	″julian″, ″mdy″, ″dmy″, ″ymd″, ″usa″, ″iso″, ″eur″, ″jis″	Esta propiedad permite cambiar el formato de las fechas.

Método Set (tipo de datos)	Valores	Descripción
setDateSeparator (String)	"/", "-", ".", ",", "b"	Esta propiedad permite cambiar el separador de fecha. Sólo es válida en combinación con algunos de los valores de dateFormat (según las normas del sistema).
setDecimalSeparator (String)	".", ","	Esta propiedad permite cambiar el separador decimal.
setDescription (String)	Cualquier nombre	Esta propiedad permite establecer el texto descriptivo de este objeto DataSource.
setDoEscapeProcessing (boolean)	"true", "false"	Esta propiedad especifica si se realiza proceso de escape en las sentencias SQL. El valor por omisión de esta propiedad es true.
setFullErrors (boolean)	"true", "false"	Esta propiedad permite devolver el texto de errores de segundo nivel de todo el sistema en mensajes de objeto SQLException. El valor por omisión es false.
setLibraries (String)	Una lista de bibliotecas separadas mediante espacios	Esta propiedad permite colocar una lista de bibliotecas en la lista de bibliotecas del trabajo servidor. Esta propiedad sólo se utiliza cuando se utiliza setSystemNaming(true).
setLobThreshold (int)	Cualquier valor por debajo de 500000	Esta propiedad indica al controlador que coloque los valores reales en lugar de localizadores LOB (Locator OBject) si la columna LOB es inferior al tamaño del umbral.
setLoginTimeout (int)	Cualquier valor	Esta propiedad se pasa por alto actualmente; está prevista para uso futuro.
setNetworkProtocol (int)	Cualquier valor	Esta propiedad se pasa por alto actualmente; está prevista para uso futuro.
setPassword (String)	Cualquier serie	Esta propiedad prevé la especificación de una contraseña para la conexión. Se pasa por alto si no se establece un ID de usuario.
setPortNumber (int)	Cualquier valor	Esta propiedad se pasa por alto actualmente; está prevista para uso futuro.
setPrefetch (boolean)	"true", "false"	Esta propiedad especifica si el controlador debe extraer los primeros datos de un conjunto de resultados inmediatamente después del proceso o esperar a que se soliciten los datos. El valor por omisión es true.

Método Set (tipo de datos)	Valores	Descripción
setReuseObjects (boolean)	″true″, ″false″	Esta propiedad especifica si el controlador intenta reutilizar algunos tipos de objetos después de que el usuario los haya cerrado. Esto representa una mejora en el rendimiento. El valor por omisión es true.
setServerName (String)	Cualquier nombre	Esta propiedad se pasa por alto actualmente; está prevista para uso futuro.
setServerTraceCategories (int)	Una representación de serie de un entero	Esta propiedad habilita el rastreo del trabajo servidor JDBC. Si el rastreo del servidor está habilitado, el rastreo se inicia cuando el cliente se conecta al servidor y finaliza cuando termina la conexión. Los datos de rastreo se recogen en archivos en spool en el servidor. Pueden activarse varios niveles de rastreos del servidor combinados, añadiendo las constantes y pasando esa suma en el método set. **Nota:** el personal de soporte utiliza habitualmente esta propiedad y sus valores no se describen con más detalle.
setSystemNaming (boolean)	″true″, ″false″	Esta propiedad permite especificar si las colecciones y tablas deben separarse mediante un punto (denominación SQL) o mediante una barra inclinada (denominación del sistema). Esta propiedad también determina si se utiliza una biblioteca por omisión (denominación SQL) o si se utiliza la lista de bibliotecas (denominación del sistema). El valor por omisión es la denominación SQL.
setTimeFormat (String)	″hms″, ″usa″, ″iso″, ″eur″, ″jis″	Esta propiedad permite cambiar el formato de los valores de hora.
setTimeSeparator (String)	″:″, ″.″, ″,″, ″b″	Esta propiedad permite cambiar el separador de hora. Sólo es válida en combinación con algunos de los valores de timeFormat (según las normas del sistema).
setTrace (boolean)	″true″, ″false″	Esta propiedad puede habilitar un rastreo simple. El valor por omisión es false.
setTransactionIsolationLevel (String)	″none″, ″read committed″, ″read uncommitted″, ″repeatable read″, ″serializable″	Esta propiedad permite la especificación del nivel de aislamiento de transacción. El valor por omisión de esta propiedad es ″none″, ya que JDBC toma por omisión la modalidad de compromiso automático.

Método Set (tipo de datos)	Valores	Descripción
setTranslateBinary (Boolean)	″true″, ″false″	Esta propiedad puede utilizarse para obligar al controlador JDBC a que trate los valores de datos de tipo binary y varbinary como si fuesen valores de datos de tipo char y varchar. El valor por omisión de esta propiedad es false.
setUseBlockInsert (boolean)	″true″, ″false″	Esta propiedad permite al controlador JDBC nativo colocarse en modalidad de inserción de bloques para insertar bloques de datos en la base de datos. Esta es una versión optimizada de la actualización por lotes. Esta modalidad optimizada sólo puede utilizarse en aplicaciones que garanticen no transgredir determinadas restricciones del sistema ni producir anomalías de inserción de datos, pudiendo dañar los datos. Las aplicaciones que activen esta propiedad sólo deben conectarse al sistema local al intentar realizar actualizaciones por lotes. No utilizan DRDA para establecer conexiones remotas, ya que la inserción por bloques no puede gestionarse a través de DRDA. Las aplicaciones también deben asegurarse de que PreparedStatements con una sentencia SQL insert y una cláusula values indican todos los parámetros de valores de inserción. No se permiten contantes en la lista de valores. Este es un requisito del motor de inserción por bloques del sistema. El valor por omisión es false.
setUser (String)	cualquier valor	Esta propiedad permite establecer un ID de usuario para la obtención de conexiones. Esta propiedad requiere que se establezca también la propiedad password.

Otras implementaciones de DataSource: Existen dos implementaciones de la interfaz DataSource incluidas en el controlador JDBC nativo. Estas implementaciones de DataSource deben considerarse desestimadas. Aunque todavía pueden utilizarse, no está previsto someterlas a futuras mejoras; por ejemplo, no se añaden conexiones robustas ni agrupación de sentencias a estas implementaciones. Estas implementaciones existirán hasta que adopte la interfaz UDBDataSource y sus funciones relacionadas.

DB2DataSource: DB2DataSource era una implementación antigua de la interfaz DataSource y no se ajusta a la especificación completa (es decir, es anterior a la especificación). DB2DataSource existe actualmente sólo para permitir a los usuarios de WebSphere migrar a releases actuales, y no debe utilizarse si no es con este fin.

DB2StdDataSource: DB2StdDataSource es la versión revisada de la implementación de DB2DataSource que pasó a cumplir con la especificación cuando la especificación de paquetes opcionales de JDBC pasó a ser definitiva. La nueva versión se ha suministrado para no romper con el código ya escrito en la versión DB2DataSource.

Si ha escrito aplicaciones que utilizan estas implementaciones de DataSource, la migración a UDBDataSource es una tarea sin importancia, ya que todas las propiedades antiguas están soportadas. Es aconsejable migrar a UDBDataSource para beneficiarse de las funciones de las nuevas clases de UDBDataSource.

Propiedades de la JVM para JDBC

Algunos valores utilizados por el controlador JDBC nativo no pueden establecerse utilizando una propiedad de conexión. Estos valores deben establecerse para la JVM en la que se ejecuta el controlador JDBC nativo. Estos valores se utilizan para todas las conexiones creadas por el controlador JDBC nativo.

El controlador nativo reconoce las siguientes propiedades de la JVM:

Propiedad	Valores	Significado
jdbc.db2.job.sort.sequence	valor por omisión = *HEX	Establecer esta propiedad como verdadera provoca que el controlador JDBC nativo utilice la Secuencia de ordenación de trabajos del usuario que inicia el trabajo en lugar de utilizar el valor por omisión de *HEX. Establecerla con otro valor o dejarla en blanco provocará que JDBC continúe utilizando el valor por omisión de *HEX. Tenga en cuenta lo que esto significa. Cuando las conexiones JDBC se pasan en perfiles de usuario distintos en peticiones de conexión, la secuencia de ordenación del perfil de usuario que inicia el servidor se utiliza para todas las conexiones. Este es un atributo de entorno que se establece en el momento del inicio, no un atributo de conexión dinámica.
jdbc.db2.trace	1 o error = Rastrear información de error 2 o info = Rastrear información e información de error 3 o verbose = Rastrear verboso, información e información de error 4 o all o true = Rastrear toda la información posible	Esta propiedad activa el rastreo para el controlador JDBC. Deberá utilizarse al informar de un problema.

Propiedad	Valores	Significado
jdbc.db2.trace.config	stdout = Se envía información de rastreo a stdout (valor por omisión) usrtrc = Se envía información de rastreo a un rastreo de usuario. El mandato CL Volcar almacenamiento intermedio de rastreo de usuario (DMPUSRTRC) puede utilizarse para obtener la información de rastreo. file://<pathtofile> = Se envía información de rastreo a un archivo. Si el nombre de archivo contiene "%j", se sustituirá "%j" por el nombre de trabajo. Un ejemplo de <pathtofile> es /tmp/jdbc.%j.trace.txt.	Este propiedad se utiliza para especificar a dónde debe ir la salida del rastreo.

Interfaz DatabaseMetaData para IBM Developer Kit para Java

El controlador JDBC de IBM Developer Kit para Java implementa la interfaz DatabaseMetaData con objeto de proporcionar información acerca de sus orígenes de datos subyacentes. La utilizan principalmente los servidores de aplicaciones y las herramientas para determinar cómo hay que interaccionar con un origen de datos dado. Las aplicaciones también pueden servirse de los métodos de DatabaseMetaData para obtener información sobre un origen de datos, pero esto ocurre con menos frecuencia.

La interfaz DatabaseMetaData incluye alrededor de 150 métodos, que se pueden clasificar en categorías en función de los siguientes tipos de información que proporcionan:

- "Recuperar información general" en la página 67 acerca del origen de datos
- "Determinar el soporte de característica" en la página 67
- "Límites de origen de datos" en la página 67
- "Objetos SQL y sus atributos" en la página 67
- "Soporte de transacción" en la página 68 ofrecido por el origen de datos

La interfaz DatabaseMetaData también contiene alrededor de 40 campos, que son constantes empleadas como valores de retorno en los diversos métodos de DatabaseMetaData.

Consulte la sección "Cambios en JDBC 3.0" en la página 68 para obtener información acerca de los cambios efectuados en la interfaz DatabaseMetaData.

Crear un objeto DatabaseMetaData: Un objeto DatabaseMetaData se crea con el método getMetaData de Connection. Una vez creado el objeto, puede utilizarse para buscar dinámicamente información acerca del origen de datos subyacente. El ejemplo siguiente crea un objeto DatabaseMetaData y lo utiliza para determinar el número máximo de caracteres permitidos para un nombre de tabla:

Ejemplo: crear un objeto DatabaseMetaData

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
// con es un objeto Connection.
DatabaseMetaData dbmd = con.getMetadata();
int maxLen = dbmd.getMaxTableNameLength();
```

Recuperar información general: Algunos métodos de DatabaseMetaData se emplean para buscar dinámicamente información general acerca de un origen de datos, y también para obtener detalles sobre su implementación. Algunos de estos métodos son:

- getURL
- getUserName
- getDatabaseProductVersion, getDriverMajorVersion y getDriverMinorVersion
- getSchemaTerm, getCatalogTerm y getProcedureTerm
- nullsAreSortedHigh y nullsAreSortedLow
- usesLocalFiles y usesLocalFilePerTable
- getSQLKeywords

Determinar el soporte de característica: Hay un gran grupo de métodos de DatabaseMetaData que permiten determinar si una característica concreta (o un conjunto concreto de características) está soportada por el controlador o el origen de datos subyacente. Aparte de esto, existen métodos que describen el nivel de soporte que se proporciona. Algunos de los métodos que describen el soporte de características individuales son:

- supportsAlterTableWithDropColumn
- supportsBatchUpdates
- supportsTableCorrelationNames
- supportsPositionedDelete
- supportsFullOuterJoins
- supportsStoredProcedures
- supportsMixedCaseQuotedIdentifiers

Entre los métodos que describen un nivel de soporte de característica se incluyen los siguientes:

- supportsANSI92EntryLevelSQL
- supportsCoreSQLGrammar

Límites de origen de datos: Otro grupo de métodos proporciona los límites impuestos por un determinado origen de datos. Algunos de los métodos de esta categoría son:

- getMaxRowSize
- getMaxStatementLength
- getMaxTablesInSelect
- getMaxConnections
- getMaxCharLiteralLength
- getMaxColumnsInTable

Los métodos de este grupo devuelven el valor de límite como un entero (integer). Si el valor de retorno es cero, indica que no hay ningún límite o que el límite es desconocido.

Objetos SQL y sus atributos: Diversos métodos de DatabaseMetaData proporcionan información sobre los objetos SQL que llenan un determinado origen de datos. Estos métodos pueden determinar los atributos de los objetos SQL. Estos métodos también devuelven objetos ResultSet, en los que cada fila describe un objeto concreto. Por ejemplo, el método getUDTs devuelve un objeto ResultSet en el que hay una fila para cada tabla definida por usuario (UDT) que se haya definido en el origen de datos. Son ejemplos de esta categoría:

- getSchemas y getCatalogs
- getTables
- getPrimaryKeys
- getProcedures y getProcedureColumns

- getUDTs

Soporte de transacción: Hay un pequeño grupo de métodos que proporcionan información sobre la semántica de transacción soportada por el origen de datos. Son ejemplos de esta categoría:
- supportsMultipleTransactions
- getDefaultTransactionIsolation

En Ejemplo: interfaz DatabaseMetaData para IBM Developer Kit para Java hallará un ejemplo de cómo se utiliza la interfaz DatabaseMetaData.

Cambios en JDBC 3.0: En JDBC 3.0 existen cambios en los valores de retorno de algunos de los métodos. Los siguientes métodos se han actualizado en JDBC 3.0 para añadir campos a los ResultSets que devuelven.
- getTables
- getColumns
- getUDTs
- getSchemas

Nota: si está desarrollando una implementación mediante Java Development Kit (JDK) 1.4, puede que observe que se devuelve un determinado número de columnas al efectuar la prueba. El usuario escribe la aplicación y espera acceder a todas estas columnas. Sin embargo, si la aplicación se está diseñando para que también funcione en releases anteriores de JDK, ésta recibe una SQLException cuando intenta acceder a estos campos, que no existen en releases anterior de JDK. SafeGetUDTs es un ejemplo de cómo puede escribirse una aplicación que funcione en JDK 1.4, JDK 1.3 y en releases anteriores de JDK.

Excepciones

El lenguaje Java utiliza excepciones para proporcionar posibilidades de manejo de errores para sus programas. Una excepción es un evento que se produce cuando se ejecuta el programa de forma que interrumpe el flujo normal de instrucciones.

El sistema de ejecución Java y muchas clases de paquetes Java lanzan excepciones en algunas circunstancias utilizando la sentencia throw. Puede utilizar el mismo mecanismo para lanzar excepciones en los programas Java.

Para obtener más información acerca de las excepciones, consulte las siguientes secciones:

SQLException
La clase SQLException y sus subtipos proporcionan información acerca de los errores y avisos que se producen mientras se está accediendo a un origen de datos.

SQLWarning
Los métodos generan un objeto SQLWarning si provocan un aviso de acceso a base de datos. Los métodos de las siguientes interfaces pueden generar SQLWarnings:
- Connection
- Statement y sus subtipos, PreparedStatement y CallableStatement
- ResultSet

DataTruncation
DataTruncation es una subclase de SQLWarning. Aunque no se lanzan SQLWarnings, a veces se lanzan objetos DataTruncation y se conectan al igual que otros objetos SQLWarning.

"Truncamiento silencioso" en la página 73
El método de sentencia setMaxFieldSize permite especificar un tamaño máximo de campo para

cualquier columna. Si los datos se truncan debido a que el tamaño ha sobrepasado el valor de tamaño máximo de campo, no se informa de ningún aviso ni excepción.

SQLException: La clase SQLException y sus subtipos proporcionan información acerca de los errores y avisos que se producen mientras se está accediendo a un origen de datos.

A diferencia de la mayor parte de JDBC, que se define mediante interfaces, el soporte de excepciones se suministra en clases. La clase básica para las excepciones que se producen durante la ejecución de aplicaciones JDBC es SQLException. Todos los métodos de la API JDBC se declaran capaces de lanzar SQLExceptions. SQLException es una ampliación de java.lang.Exception y proporciona información adicional relacionada con las anomalías que se producen en un contexto de base de datos. Específicamente, en una SQLException está disponible la siguiente información:

- Texto descriptivo
- SQLState
- Código de error
- Una referencia a las demás excepciones que también se han producido

ExceptionExample es un programa que maneja adecuadamente la captura de una SQLException (esperada, en este caso), y el vuelco de toda la información que proporciona.

Nota: JDBC proporciona un mecanismo para encadenar las excepciones. Esto permite al controlador o a la base de datos informar de varios errores en una sola petición. Actualmente, no existen instancias en las que el controlador JDBC nativo realice esta acción. Sin embargo, esta información sólo se proporciona como referencia y no como indicación clara de que el controlador nunca realizará esta acción en el futuro.

Como se ha indicado, los objetos SQLException se lanzan cuando se producen errores. Esta afirmación es correcta, pero es incompleta. En la práctica, el controlador JDBC nativo rara vez lanza SQLExceptions reales. Lanza instancias de sus propias subclases SQLException. Esto permite al usuario determinar más información acerca de lo que realmente ha fallado, como se indica a continuación.

DB2Exception.java: Los objetos DB2Exception no se lanzan directamente. Esta clase básica se utiliza para contener las funciones que son comunes a todas las excepciones JDBC. Existen dos subclases de esta clase que son las excepciones estándar que lanza JDBC. Estas subclases son DB2DBException.java y DB2JDBCException.java. Las DB2DBExceptions son excepciones de las que se informa al usuario que provienen directamente de la base de datos. Las DB2JDBCExceptions se lanzan cuando el controlador JDBC detecta problemas por su cuenta. Dividir la jerarquía de clases de excepción de esta forma permite manejar los dos tipos de excepciones de forma distinta.

DB2DBException.java: Como se ha indicado, las DB2DBExceptions son excepciones que provienen directamente de la base de datos. Se detectan cuando el controlador JDBC efectúa una llamada a CLI y obtiene un código de retorno SQLERROR. En estos casos, se llama al SQLError de la función CLI para obtener el texto del mensaje, SQLState, y el código del proveedor. También se recupera y se devuelve al usuario el texto de sustitución del SQLMessage. La clase DatabaseException provoca un error que la base de datos reconoce e indica al controlador JDBC que cree el objeto de excepción para el mismo.

DB2JDBCException.java: Las DB2JDBCExceptions se generan para condiciones de error que provienen del propio controlador JDBC. El funcionamiento de esta clase de excepciones es fundamentalmente diferente; el propio controlador JDBC maneja la conversión de lenguaje del mensaje de la excepción y otras cuestiones que el sistema operativo y la base de datos manejan en el caso de las excepciones que se originan en la base de datos. Siempre que es posible, el controlador JDBC se ajusta a los SQLStates de la base de datos. El código de proveedor para las excepciones que lanza el controlador JDBC es siempre -99999. Las DB2DBExceptions reconocidas y devueltas por la capa CLI también tienen con frecuencia el código de error -99999. La clase JDBCException provoca un error que el controlador JDBC reconoce y crea la excepción por su cuenta. Durante el desarrollo de este release, se ha creado la siguiente salida. Observe

que, al principio de la pila, se encuentra DB2JDBCException. Esto indica que se está informando del error desde el controlador JDBC antes de efectuar la petición a la base de datos.

SQLWarning: Los métodos de las siguientes interfaces generan un objeto SQLWarning si provocan un aviso de acceso a base de datos:

- Connection
- Statement y sus subtipos, PreparedStatement y CallableStatement
- ResultSet

Cuando un método genera un objeto SQLWarning, no se informa al llamador de que se ha producido un aviso de acceso a base de datos. Debe llamarse al método getWarnings en el objeto adecuado para recuperar el objeto SQLWarning. Sin embargo, en algunas circunstancias puede que se lance la subclase DataTruncation de SQLWarning. Debe tenerse en cuenta que el controlador JDBC nativo opta por pasar por alto algunos avisos generados por la base de datos para aumentar la eficacia. Por ejemplo, el sistema genera un aviso cuando el usuario intenta recuperar datos más allá del final de un ResultSet mediante el método ResultSet.next. En este caso, el método next se define para devolver false en lugar de true, informando al usuario del error. Sería innecesario crear un objeto que avisara de nuevo, por lo que el aviso se pasa simplemente por alto.

Si se producen múltiples avisos de acceso a datos, los nuevos avisos se encadenan al primero y, para recuperarlos, se llama al método SQLWarning.getNextWarning. Si no hay más avisos en la cadena, el método getNextWarning devuelve `null`.

Los objetos SQLWarning subsiguientes se siguen añadiendo a la cadena hasta que se procesa la próxima sentencia o, en el caso de un objeto ResultSet, cuando vuelve a situarse el cursor. Como resultado, se eliminan todos los objetos SQLWarning de la cadena.

La utilización de objetos Connection, Statement y ResultSet puede provocar la generación de SQLWarnings. Los SQLWarnings son mensajes informativos que indican que, aunque una operación determinada se ha realizado satisfactoriamente, puede haber otra información sobre la que el usuario debe estar al corriente. Los SQLWarnings son una ampliación de la clase SQLException, pero no se lanzan. En lugar de ello, se conectan al objeto que provoca su generación. Cuando se genera un SQLWarning, no ocurre nada que indique a la aplicación que se ha generado el aviso. Las aplicaciones deben solicitar activamente la información de aviso.

Al igual que las SQLExceptions, los SQLWarnings pueden encadenarse entre sí. Puede llamar al método clearWarnings en un objeto Connection, Statement o ResultSet para borrar los avisos correspondientes a ese objeto.

Nota: al llamar al método clearWarnings no se borran todos los avisos. Sólo se borran los avisos asociados con un objeto determinado.

El controlador JDBC borra los objetos SQLWarning en momentos específicos si el usuario no los borra manualmente. Los objetos SQLWarning se borran cuando se realizan las siguientes acciones:

- En la interfaz Connection, los avisos se borran durante la creación de un objeto Statement, PreparedStatement o CallableStatement nuevo.
- En la interfaz Statement, los avisos se borran cuando se procesa la próxima sentencia (o cuando se procesa de nuevo la sentencia para PreparedStatements y CallableStatements).
- En la interfaz ResultSet, los avisos se borran cuando vuelve a situarse el cursor.

DataTruncation: DataTruncation es una subclase de SQLWarning. Aunque no se lanzan SQLWarnings, a veces se lanzan objetos DataTruncation y se conectan al igual que otros objetos SQLWarning. Los objetos DataTruncation proporcionan información adicional más allá de lo que devuelve un SQLWarning. La información disponible es la siguiente:

- El número de bytes de datos que se han transferido.
- El índice de columna o parámetro que se ha truncado.
- Si el índice es para un parámetro o para una columna de ResultSet.
- Si el truncamiento se ha producido al leer en la base de datos o al escribir en ella.
- La cantidad de datos que se han transferido realmente.

En algunos casos, la información puede descifrarse, pero se producen situaciones que no son completamente intuitivas. Por ejemplo, si se utiliza el método setFloat de PreparedStatement para insertar un valor en una columna que contiene valores enteros, puede producirse una DataTruncation debido a que float puede ser mayor que el valor mayor que la columna puede contener. En estas situaciones, las cuentas de bytes para el truncamiento no tienen sentido, pero es importante para el controlador proporcionar la información de truncamiento.

Informar de los métodos set() y update(): Existe una ligera diferencia entre los controladores JDBC. Algunos controladores, como por ejemplo los controladores JDBC nativos y de IBM Toolbox para Java capturan e informan de las situaciones de truncamiento de datos en el momento de establecer el parámetro. Esta acción se realiza en el método set de PreparedStatement o en el método update de ResultSet. Otros controladores informan del problema en el momento de procesar la sentencia, y se realiza mediante los métodos execute, executeQuery o updateRow.

No informar del problema en el momento de proporcionar datos incorrectos en lugar de hacerlo en el momento que el proceso ya no puede continuar ofrece un par de ventajas:
- La anomalía puede dirigirse a la aplicación cuando se produce un problema en lugar de dirigir el problema en el momento del proceso.
- Efectuando la comprobación al establecer los parámetros, el controlador JDBC puede asegurarse de que los valores que se pasan a la base de datos en el momento de procesar la sentencia son válidos. Esto permite a la base de datos optimizar su trabajo y el proceso puede realizarse con mayor rapidez.

Los métodos ResultSet.update() lanzan excepciones DataTruncation: En algunos releases anteriores, los métodos ResultSet.update() enviaban avisos cuando existían condiciones de truncamiento. Este caso se produce cuando el valor de datos va a insertarse en la base de datos. La especificación indica que los controladores JDBC deben lanzar excepciones en estos casos. Como resultado, el controlador JDBC funciona de esta forma.

No hay ninguna diferencia significativa entre manejar una función de actualización de ResultSet que recibe un error de truncamiento de datos y manejar un parámetro de sentencia preparada establecido para una sentencia update o insert que recibe un error. En ambos casos, el problema es idéntico; el usuario ha proporcionado datos que no caben donde se desea.

Los datos de tipo NUMERIC y DECIMAL se truncan a la derecha de una coma decimal de forma silenciosa. Así es como funciona JDBC para UDB NT y también SQL interactivo en un servidor iSeries.

Nota: No se redondea ningún valor cuando se produce un truncamiento de datos. Cualquier parte fraccionaria de un parámetro que no quepa en una columna NUMERIC o DECIMAL se pierde sin aviso.

A continuación se ofrecen ejemplos, suponiendo que el valor de la cláusula values es realmente un parámetro que se establece en una sentencia preparada:

```
create table cujosql.test (col1 numeric(4,2))
a) insert into cujosql.test values(22.22)  // funciona - inserta 22.22
b) insert into cujosql.test values(22.223) // funciona - inserta 22.22
c) insert into cujosql.test values(22.227) // funciona - inserta 22.22
d) insert into cujosql.test values(322.22) // falla - Error de conversión en la asignación a columna COL1.
```

Diferencia entre un aviso de truncamiento de datos y una excepción de truncamiento de datos

La especificación indica que el truncamiento de datos en un valor que debe escribirse en la base de datos debe lanzar una excepción. Si el truncamiento de datos no se realiza en el valor que se escribe en la base de datos, se genera un aviso. Esto significa que, en el momento en que se identifica una situación de truncamiento de datos, el usuario también debe tener conocimiento del tipo de sentencia que el truncamiento de datos está procesando. Dado este requisito, a continuación figura una lista del comportamiento de varios tipos de sentencias SQL:

- En una sentencia SELECT, los parámetros de consulta nunca dañan el contenido de la base de datos. Por tanto, las situaciones de truncamiento de datos se manejan siempre enviando avisos.
- En las sentencias VALUES INTO y SET, los valores de entrada sólo se utilizan para generar valores de salida. En consecuencia, se emiten avisos.
- En una sentencia CALL, el controlador JDBC no puede determinar lo que un procedimiento almacenado hace con un parámetro. Se lanzan siempre excepciones cuando se trunca un parámetro de procedimiento almacenado.
- Todos los demás tipos de sentencias lanzan excepciones en lugar de enviar avisos.

Propiedad de truncamiento de datos para Connection y DataSource: Durante muchos releases ha existido una propiedad de truncamiento de datos disponible. El valor por omisión de esa propiedad es true, que indica que se comprueban los problemas de truncamiento de datos y se envían avisos o se lanzan excepciones. La propiedad se suministra a efectos de conveniencia y rendimiento en los casos en que el hecho de que un valor no quepa en la columna de la base de datos no es preocupante. El usuario desea que el controlador coloque en la columna la mayor parte posible del valor.

La propiedad de truncamiento de datos sólo afecta a tipos de datos de carácter y basados en binario: Hasta hace dos releases, la propiedad de truncamiento de datos determinaba si podían lanzarse excepciones de truncamiento de datos. La propiedad de truncamiento de datos se introdujo para que las aplicaciones JDBC no tuvieran que preocuparse de si se truncaba un valor si el truncamiento no era importante. Existen unos pocos casos en los que deseará almacenar el valor 00 o 10 en la base de datos cuando las aplicaciones intenten insertar 100 en un DECIMAL(2,0). Por tanto, la propiedad de truncamiento de datos del controlador JDBC se ha cambiado para prestar atención sólo a las situaciones en las que el parámetro es para tipos basados en caracteres, como por ejemplo CHAR, VARCHAR, CHAR FOR BIT DATA y VARCHAR FOR BIT DATA.

La propiedad de truncamiento de datos sólo se aplica a parámetros: La propiedad de truncamiento de datos es un valor del controlador JDBC y no de la base de datos. En consecuencia, no tiene ningún efecto sobre los literales de sentencia. Por ejemplo, las sentencias siguientes que se procesan para insertar un valor en una columna CHAR(8) de la base de datos siguen fallando con el identificador de truncamiento de datos establecido en false (suponiendo que la conexión sea un objeto java.sql.Connection asignado a cualquier otro lugar).

```
Statement stmt = connection.createStatement();
Stmt.executeUpdate("create table cujosql.test (col1 char(8))");
Stmt.executeUpdate("insert into cujosql.test values('dettinger')");
// Falla debido a que el valor no cabe en la columna de la base de datos.
```

El controlador JDBC nativo lanza excepciones por truncamientos de datos insignificantes: El controlador JDBC nativo no observa los datos que el usuario proporciona para los parámetros. Al hacerlo, sólo ralentizaría el proceso. Sin embargo, pueden darse situaciones en las que no importa que un valor se trunque, pero no se ha establecido la propiedad de conexión de truncamiento de datos en false.

Por ejemplo, 'dettinger ', un valor char(10) que se pasa, lanza una excepción aunque quepan todas las partes importantes del valor. Este es el funcionamiento de JDBC para UDB NT; sin embargo, no es el comportamiento que obtendría si pasara el valor como un literal de una sentencia SQL. En este caso, el motor de base de datos eliminaría los espacios adicionales de forma silenciosa.

Los problemas con el controlador JDBC que no lanza una excepción son los siguientes:

- La actividad general de rendimiento es extensiva en todos los métodos set, sea o no necesaria. En la mayoría de los casos en los que no representa ninguna ventaja, se produce una actividad general de rendimiento considerable en una función tan común como setString().
- La solución alternativa es sencilla, por ejemplo, llamar a la función de ajuste (trim) en el valor de serie que se pasa.
- Existen situaciones en la columna de la base de datos que deben tenerse en cuenta. Un espacio del CCSID 37 no es en absoluto un espacio del CCSID 65535 o 13488.

Truncamiento silencioso: El método de sentencia setMaxFieldSize permite especificar un tamaño máximo de campo para cualquier columna. Si los datos se truncan debido a que el tamaño ha sobrepasado el valor de tamaño máximo de campo, no se informa de ningún aviso ni excepción. Este método, al igual que la propiedad de truncamiento de datos mencionada anteriormente, sólo afecta a tipos basados en caracteres, como por ejemplo CHAR, VARCHAR, CHAR FOR BIT DATA y VARCHAR FOR BIT DATA.

Transacciones

Una **transacción** es una unidad lógica de trabajo. Para realizar una unidad lógica de trabajo, puede ser necesario llevar a cabo varias acciones en una base de datos. El soporte transaccional permite a las aplicaciones asegurarse de que:

- Se siguen todos los pasos para realizar una unidad lógica de trabajo.
- Cuando falla uno de los pasos de los archivos de la unidad de trabajo, todo el trabajo realizado como parte de esa unidad lógica de trabajo puede deshacerse y la base de datos puede volver a su estado anterior al inicio de la transacción.

Las transacciones se utilizan para proporcionar integridad de los datos, una semántica correcta de la aplicación y una vista coherente de los datos durante el acceso concurrente. Todos los controladores compatibles con JDBC (Java Database Connectivity) deben dar soporte a las transacciones.

Nota: esta sección sólo describe las transacciones locales y el concepto de JDBC estándar con respecto a las transacciones. Java y el controlador JDBC nativo soportan la API de transacción Java (JTA), las transacciones distribuidas y el protocolo de compromiso de dos fases (2PC).

Todo el trabajo transaccional se maneja a nivel de objetos Connection. Cuando finaliza el trabajo de una transacción, ésta puede finalizarse llamando al método commit. Si la aplicación termina anormalmente la transacción, se llama al método rollback.

Todos los objetos Statement de una conexión forman parte de la transacción. Esto significa que, si una aplicación crea tres objetos Statement y utiliza cada uno de los objetos para efectuar cambios en la base de datos, cuando se produzca una llamada a commit o rollback, el trabajo de las tres sentencias se convierte en permanente o se descarta.

Las sentencias SQL commit y rollback SQL se utilizan para finalizar transacciones al trabajar sólo con SQL. Estas sentencias SQL no pueden prepararse dinámicamente, y no debe intentar utilizarlas en las aplicaciones JDBC para finalizar transacciones.

Para utilizar transacciones de forma eficaz en la aplicación, consulte las siguientes secciones:

Modalidad de compromiso automático
JDBC utiliza la modalidad de compromiso automático, en la que todas las actualizaciones de la base de datos se convierten inmediatamente en permanentes.

Niveles de aislamiento de las transacciones
Los niveles de aislamiento de las transacciones especifican qué datos son visibles para las sentencias dentro de una transacción e influyen directamente sobre el nivel de acceso concurrente.

Puntos de salvar
Los puntos de salvar son puntos de comprobación a los que la aplicación puede retrotraerse sin eliminar toda la transacción. Consulte la siguiente información acerca de los puntos de salvar:

- Establecer puntos de salvar y retrotraerse a ellos
- Liberar un punto de salvar

Modalidad de compromiso automático: Por omisión, JDBC utiliza una modalidad de operación denominada compromiso automático. Esto significa que todas las actualizaciones de la base de datos se convierten inmediatamente en permanentes. Cualquier situación en la que una unidad lógica de trabajo requiera más de una actualización de la base de datos no puede gestionarse de forma segura en modalidad de compromiso automático. Si se produce alguna anomalía en la aplicación o en el sistema después de realizar una actualización y antes de que se realicen otras actualizaciones, el primer cambio no puede deshacerse en la ejecución por modalidad de compromiso automático.

Dado que en la modalidad de compromiso automático los cambios se convierten inmediatamente en permanentes, no es necesario que la aplicación llame al método commit ni al método rollback. Esto facilita la escritura de las aplicaciones.

La modalidad de compromiso automático puede habilitarse e inhabilitarse dinámicamente durante la existencia de una conexión. El compromiso automático se habilita de la siguiente forma, suponiendo que el origen de datos ya exista:

```
Connection connection = dataSource.getConnection();

Connection.setAutoCommit(false);        // Inhabilita el compromiso automático.
```

Si se cambia el valor de compromiso automático en medio de una transacción, el trabajo pendiente se compromete automáticamente. Se genera una SQLException si se habilita el compromiso automático para una conexión que forma parte de una transacción distribuida.

Niveles de aislamiento de las transacciones: Los niveles de aislamiento de las transacciones especifican qué datos son visibles para las sentencias dentro de una transacción. Estos niveles influyen directamente sobre el nivel de acceso concurrente al definir qué interacción es posible entre las transacciones en el mismo origen de datos destino.

Anomalías de base de datos: Las anomalías de base de datos son resultados generados que parecen incorrectos cuando se observan desde el ámbito de una sola transacción, pero que son correctos cuando se observan desde el ámbito de todas las transacciones. A continuación se describen los diversos tipos de anomalías de base de datos:

- Se producen lecturas **sucias** cuando:
  1. La transacción A inserta una fila en una tabla.
  2. La transacción B lee la fila nueva.
  3. La transacción A efectúa una retrotracción.

  La transacción B puede haber realizado trabajo en el sistema basándose en la fila insertada por la transacción A, pero la fila nunca ha pasado a formar parte permanente de la base de datos.
- Se producen lecturas **no repetibles** cuando:
  1. La transacción A lee una fila.
  2. La transacción B cambia la fila.
  3. La transacción A lee la misma fila por segunda vez y obtiene los resultados nuevos.
- Se producen lecturas **fantasma** cuando:
  1. La transacción A lee todas las filas que satisfacen una cláusula WHERE de una consulta SQL.
  2. La transacción B inserta una fila adicional que satisface la cláusula WHERE.
  3. La transacción vuelve a evaluar la condición WHERE y recoge la fila adicional.

Nota: DB2/400 no siempre expone la aplicación a las anomalías de base de datos permitidas en los niveles prescritos debido a sus estrategias de bloqueo.

Niveles de aislamiento de las transacciones JDBC: Existen cinco niveles de aislamiento de las transacciones en la API JDBC de IBM Developer Kit para Java. A continuación figuran los niveles ordenados del menos restrictivo al más restrictivo:

JDBC_TRANSACTION_NONE
Esta es una constante especial que indica que el controlador JDBC no da soporte a las transacciones.

JDBC_TRANSACTION_READ_UNCOMMITTED
Este nivel permite que las transacciones vean los cambios no comprometidos en los datos. En este nivel son posibles todas las anomalías de base de datos.

JDBC_TRANSACTION_READ_COMMITTED
Este nivel indica que los cambios efectuados dentro de una transacción no son visibles fuera de ella hasta que se compromete la transacción. Esto impide que las lecturas sucias sean posibles.

JDBC_TRANSACTION_REPEATABLE_READ
Este nivel indica que las filas que se leen retienen bloqueos para que otra transacción no pueda cambiarlas si la transacción no ha finalizado. Esto no permite las lecturas sucias y las lecturas no repetibles. Las lecturas fantasma siguen siendo posibles.

JDBC_TRANSACTION_SERIALIZABLE
Se bloquean las tablas de la transacción para que otras transacciones que añaden valores a la tabla o los eliminan de la misma no puedan cambiar condiciones WHERE. Esto impide todos los tipos de anomalías de base de datos.

Puede utilizar el método setTransactionIsolation para cambiar el nivel de aislamiento de las transacciones para una conexión.

Consideraciones: Una malinterpretación común es que la especificación JDBC define los cinco niveles transaccionales mencionados anteriormente. Se cree generalmente que el valor TRANSACTION_NONE representa el concepto de ejecución sin control de compromiso. La especificación JDBC no define TRANSACTION_NONE de la misma forma. TRANSACTION_NONE se define en la especificación JDBC como un nivel en el que el controlador no soporta las transacciones y que no es un controlador compatible con JDBC. El nivel NONE nunca se indica cuando se llama al método getTransactionIsolation.

La cuestión se complica tangencialmente por el hecho de que el nivel de aislamiento de transacciones por omisión del controlador JDBC queda definido por la implementación. El nivel por omisión de aislamiento de transacciones para el controlador JDBC nativo es NONE. Esto permite que el controlador trabaje con archivos que no tienen diarios, y no es necesario que el usuario realice especificaciones, como por ejemplo los archivos de la biblioteca QGPL.

El controlador JDBC nativo permite pasar JDBC_TRANSACTION_NONE al método setTransactionIsolation o especificar none como propiedad de conexión. Sin embargo, el método getTransactionIsolation siempre indica JDBC_TRANSACTION_READ_UNCOMMITTED cuando el valor es none. Es responsabilidad de la aplicación efectuar el seguimiento del nivel que se está ejecutando, si la aplicación lo necesita.

En releases anteriores, el controlador JDBC manejaba la especificación de true por parte del usuario para el compromiso automático cambiando el nivel de aislamiento de las transacciones a none, debido a que el sistema no tenía el concepto de modalidad de compromiso automático true. Esta era una aproximación bastante exacta a la funcionalidad, pero no proporcionaba los resultados correctos en todos los escenarios. Esto ya no se realiza; la base de datos desasocia el concepto de compromiso automático del concepto de nivel de aislamiento de las transacciones. Por tanto, es completamente válida la ejecución al nivel JDBC_TRANSACTION_SERIALIZABLE con el compromiso automático habilitado. El único escenario que no es válido es la ejecución al nivel JDBC_TRANSACTION_NONE sin modalidad de compromiso

automático. La aplicación no puede tomar el control sobre los límites del compromiso si el sistema no se está ejecutando con un nivel de aislamiento de transacción.

Niveles de aislamiento de transacciones entre la especificación JDBC y la plataforma iSeries: La plataforma iSeries tiene nombres comunes para sus niveles de aislamiento de transacciones que no coinciden con los nombres que proporciona la especificación JDBC. La tabla siguiente empareja los nombres utilizados por la plataforma iSeries, pero no son equivalentes a los utilizados por la especificación JDBC.

Nivel JDBC*	**Nivel iSeries**
JDBC_TRANSACTION_NONE	*NONE o *NC
JDBC_TRANSACTION_READ_UNCOMMITTED	*CHG o *UR
JDBC_TRANSACTION_READ_COMMITTED	*CS
JDBC_TRANSACTION_REPEATABLE_READ	*ALL o *RS
JDBC_TRANSACTION_SERIALIZABLE	*RR

* En esta tabla, el valor JDBC_TRANSACTION_NONE se empareja con los niveles de iSeries *NONE y *NC a efectos de claridad. Este no es un emparejamiento directo de nivel entre la especificación e iSeries.

Puntos de salvar: Los puntos de salvar permiten establecer ″puntos intermedios″ en una transacción. Los puntos de salvar son puntos de comprobación a los que la aplicación puede retrotraerse sin eliminar toda la transacción. Los puntos de salvar son una novedad en JDBC 3.0, lo cual significa que la aplicación debe ejecutarse en Java Development Kit (JDK) 1.4 para utilizarlos. Además, los puntos de salvar son una novedad en Developer Kit para Java, es decir, que no están soportados si JDK 1.4 no se utiliza con releases anteriores de Developer Kit para Java.

Nota: el sistema suministra sentencias SQL para trabajar con puntos de salvar. No es aconsejable que las aplicaciones JDBC utilicen estas sentencias directamente en una aplicación. Puede funcionar, pero el controlador JDBC pierde su capacidad para efectuar el seguimiento de los puntos de salvar cuando se realiza esta operación. Como mínimo, debe evitarse mezclar los dos modelos (es decir, utilizar sentencias SQL propias de puntos de salvar y utilizar la API JDBC).

Establecer puntos de salvar y retrotraerse a ellos: Pueden establecerse puntos de salvar a lo largo del trabajo de una transacción. A continuación, la aplicación puede retrotraerse a cualquiera de estos puntos de salvar si se produce algún error y continuar el proceso a partir de ese punto. En el ejemplo siguiente, la aplicación inserta el valor FIRST en una tabla de base de datos. Después de eso, se establece un punto de salvar y se inserta otro valor, SECOND, en la base de datos. Se emite una retrotracción al punto de salvar y se deshace el trabajo de insertar SECOND, pero se conserva FIRST como parte de la transacción pendiente. Finalmente, se inserta el valor THIRD y se compromete la transacción. La tabla de base de datos contiene los valores FIRST y THIRD.

Ejemplo: establecer puntos de salvar y retrotraerse a ellos

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
Statement s = Connection.createStatement();
s.executeUpdate("insertar valores en table1 ('FIRST')");
Savepoint pt1 = connection.setSavepoint("FIRST SAVEPOINT");
s.executeUpdate("insertar valores en table1 ('SECOND')";);
connection.rollback(pt1);        // Deshace la inserción más reciente.
s.executeUpdate("insertar valores en table1 ('THIRD')");
connection.commit();
```

Aunque es poco probable que se produzcan problemas en los puntos de salvar establecidos si la modalidad es de compromiso automático, no pueden retrotraerse cuando sus vidas finalizan al final de la transacción.

Liberar un punto de salvar: La aplicación puede liberar puntos de salvar con el método releaseSavepoint del objeto Connection. Una vez liberado un punto de salvar, si intenta la retrotracción al mismo se provoca una excepción. Cuando una transacción se compromete o retrotrae, todos los puntos de salvar se liberan automáticamente. Cuando se retrotrae un punto de salvar, también se liberan los demás puntos de salvar que le siguen.

Transacciones distribuidas

Generalmente, en Java Database Connectivity (JDBC) las transacciones son locales. Esto significa que una sola conexión realiza todo el trabajo de la transacción y que la conexión sólo puede trabajar en una transacción a la vez. Cuando todo el trabajo para esa transacción ha finalizado o ha fallado, se llama al compromiso o a la retrotracción para convertir el trabajo en permanente y puede empezar una nueva transacción.

Existe un soporte avanzado para transacciones disponible en Java que proporciona funciones que van más allá de las transacciones locales. Este soporte se especifica plenamente en la especificación de la API de transacción Java (JTA) 1.0.1

.

La API de transacción Java (JTA) tiene soporte para transacciones complejas. También proporciona soporte para desasociar transacciones de objetos Connection. Al igual que JDBC se ha diseñado a partir de las especificaciones Object Database Connectivity (ODBC) y X/Open Call Level Interface (CLI), JTA se ha diseñado a partir de la especificación X/Open Extended Architecture (XA). JTA y JDBC funcionan conjuntamente para desasociar transacciones a partir de objetos Connection. El hecho de desasociar transacciones de objetos Connection permite que una sola conexión trabaje en varias transacciones simultáneamente. A la inversa, permite que varias conexiones trabajen en una sola transacción.

Nota: si tiene previsto trabajar con JTA, consulte la sección Iniciación a JDBC para obtener más información acerca de los archivos Java Archive (JAR) necesarios en la vía de acceso de clases de extensiones. Desea utilizar tanto el paquete opcional de JDBC 2.0 como los archivos JAR de JTA (JDK encuentra automáticamente estos archivos si ejecuta JDK 1.4). No se encuentran por omisión.

Transacciones con JTA: Cuando JTA y JDBC se utilizan conjuntamente, existen una serie de pasos entre ellos para realizar el trabajo transaccional. Se proporciona soporte para XA mediante la clase XADataSource. Esta clase contiene soporte para configurar la agrupación de conexiones exactamente de la misma forma que su superclase ConnectionPoolDataSource.

Con una instancia de XADataSource, puede recuperar un objeto XAConnection. El objeto XAConnection funciona como contenedor tanto para el objeto JDBC Connection como para un objeto XAResource. El objeto XAResource está diseñado para manejar el soporte transaccional XA. XAResource maneja las transacciones mediante objetos denominados ID de transacción (XID).

XID es una interfaz que debe implementar. Representa una correlación Java de la estructura XID del identificador de transacción X/Open. Este objeto contiene tres partes:

- Un ID formato de transacción global
- Una transacción global
- Un calificador de rama

Consulte la especificación JTA para obtener detalles completos acerca de esta interfaz.

El Ejemplo: utilizar JTA para manejar una transacción muestra cómo utilizar JTA para manejar una transacción en una aplicación.

Utilizar el soporte de UDBXADataSource para agrupación y transacciones distribuidas: El soporte de API de transacción Java proporciona soporte directo para agrupación de conexiones. UDBXADataSource es una ampliación de ConnectionPoolDataSource, que permite el acceso de las aplicaciones a objetos XAConnection agrupados. Puesto que UDBXADataSource es un ConnectionPoolDataSource, la configuración y utilización de UDBXADataSource es la misma que la descrita en la sección Utilizar el soporte de DataSource para agrupación de objetos.

Propiedades de XADataSource: Además de las propiedades que proporciona ConnectionPoolDataSource, la interfaz XADataSource proporciona las siguientes propiedades:

Método Set (tipo de datos)	**Valores**	**Descripción**
setLockTimeout (int)	0 o cualquier valor positivo	Cualquier valor positivo es un tiempo de espera de bloqueo válido (en segundos) a nivel de transacción. Un tiempo de espera de bloqueo 0 significa que no se impone ningún valor de tiempo de espera de bloqueo a nivel de transacción, aunque puede imponerse uno a otros niveles (el trabajo o la tabla). El valor por omisión es 0.
setTransactionTimeout (int)	0 o cualquier valor positivo	Cualquier valor positivo es un tiempo de espera de transacción válido (en segundos). Un tiempo de espera de transacción 0 significa que no se impone ningún tiempo de espera de transacción. Si la transacción permanece activa durante más tiempo del que indica el valor de tiempo de espera, se marca como de sólo retrotracción y los intentos subsiguientes de realizar trabajo en ella provocan una excepción. El valor por omisión es 0.

ResultSets y transacciones: Además de demarcar el inicio y la finalización de una transacción, como se ha mostrado en el ejemplo anterior, las transacciones pueden suspenderse durante un tiempo y reanudarse más tarde. Esto proporciona diversos escenarios para los recursos de ResultSet creados durante una transacción.

Fin de transacción simple: Al finalizar una transacción, todos los ResultSets abiertos que se crearon bajo esa transacción se cierran automáticamente. Es aconsejable cerrar explícitamente los ResultSets cuando haya terminado de utilizarlos para garantizar el máximo de proceso paralelo. Sin embargo, se produce una excepción si se accede a cualquier ResultSet que estaba abierto durante una transacción después de efectuar la llamada a XAResource.end.

Consulte el Ejemplo: finalizar una transacción, que muestra este comportamiento.

Suspender y reanudar: Mientras una transacción está suspendida, el acceso a un ResultSet creado mientras la transacción estaba activa no está permitido y provoca una excepción. Sin embargo, una vez que la transacción se ha reanudado, el ResultSet está disponible de nuevo y permanece en el mismo estado en que estaba antes de que se suspendiera la transacción.

Consulte el Ejemplo: suspender y reanudar una transacción, que muestra este comportamiento.

Efectuar ResultSets suspendidos: Mientras una transacción está suspendida, no puede accederse al ResultSet. Sin embargo, los objetos Statement pueden reprocesarse bajo otra transacción para realizar el trabajo. Debido a que los objetos JDBC Statement sólo pueden tener un ResultSet a la vez (excluyendo el soporte de JDBC 3.0 para varios ResultSets simultáneos de una llamada de procedimiento almacenado), el ResultSet de la transacción suspendida debe cerrarse para satisfacer la petición de la nueva transacción. Esto es exactamente lo que ocurre.

Consulte el Ejemplo: ResultSets suspendidos, que muestra este comportamiento.

Nota: Aunque JDBC 3.0 permite que un objeto Statement tenga varios ResultSets abiertos simultáneamente para una llamada de procedimiento almacenado, éstos se tratan como una sola unidad y todos ellos se cierran si el objeto Statement se reprocesa bajo una transacción nueva. Actualmente, no es posible tener ResultSets procedentes de dos transacciones activas simultáneamente para una sola sentencia.

Multiplexado: La API JTA está diseñada para desasociar transacciones de conexiones JDBC. Esta API permite que varias conexiones trabajen en una sola transacción o que una sola conexión trabaje en varias transacciones simultáneamente. Esto se denomina **multiplexado**, que permite realizar muchas tareas complejas que no pueden realizarse sólo con JDBC.

Este ejemplo muestra varias conexiones que trabajan en una sola transacción.

Este ejemplo muestra una sola conexión con varias transacciones que tienen lugar a la vez.

Para obtener más información acerca de la utilización de JTA, consulte la especificación JTA. La especificación JDBC 3.0 también contiene información acerca de cómo estas dos tecnologías funcionan conjuntamente para dar soporte a las transacciones distribuidas.

Compromiso de dos fases y anotación de transacciones: Las API JTA externalizan las responsabilidades del protocolo de compromiso de dos fases distribuido completamente en la aplicación. Como se ha mostrado en los ejemplos, al utilizar JTA y JDBC para acceder a una base de datos bajo una transacción JTA, la aplicación utiliza los métodos XAResource.prepare() y XAResource.commit() o sólo el método XAResource.commit() para comprometer los cambios.

Además, al acceder a varias bases de datos distintas utilizando una sola transacción, es responsabilidad de la aplicación garantizar que se realicen el protocolo de compromiso de dos fases y las anotaciones asociadas necesarias para la atomicidad de la transacción en esas bases de datos. Generalmente, el proceso de compromiso de dos fases en varias bases de datos (es decir, XAResources) y sus anotaciones se realizan bajo el control de un servidor de aplicaciones o de un supervisor de transacciones para que la propia aplicación no tenga que preocuparse de estas cuestiones.

Por ejemplo, la aplicación puede llamar a algún método commit() o efectuar la devolución desde su proceso sin errores. El servidor de aplicaciones o supervisor de transacciones subyacente empezará entonces a procesar cada una de las bases de datos (XAResource) que ha participado en la única transacción distribuida.

El servidor de aplicaciones utilizará anotaciones extensivas durante el proceso de compromiso de dos fases. Llamará al método XAResource.prepare() para cada base de datos participante (XAResource), seguido de una llamada al método XAResource.commit() para cada base de datos participante (XAResource).

Si se produce una anomalía durante este proceso, las anotaciones del supervisor de transacciones del servidor de aplicaciones permiten que el propio servidor de aplicaciones utilice subsiguientemente las API de JTA para recuperar la transacción distribuida. Esta recuperación, bajo el control del servidor de

aplicaciones o supervisor de transacciones, permite al servidor de aplicaciones situar la transacción en un estado conocido en cada base de datos participante (XAResource). Con ello garantiza un estado conocido de toda la transacción distribuida en todas las bases de datos participantes.

Tipos de sentencia

La interfaz Statement y sus subclases PreparedStatement y CallableStatement se utilizan para procesar mandatos de lenguaje de consulta estructurada (SQL) en la base de datos. Las sentencias SQL provocan la generación de objetos ResultSet.

Las subclases de la interfaz Statement se crean con diversos métodos de la interfaz Connection. Bajo un solo objeto Connection se pueden crear simultáneamente muchos objetos Statement. En releases anteriores, era posible dar números exactos de objetos Statement que podían crearse. Esto es imposible en este release, ya que los diversos tipos de objetos Statement toman números diferentes de ″handles″ dentro del motor de bases de datos. Por tanto, los tipos de objetos Statement que utilice influyen sobre el número de sentencias que pueden estar activas bajo una conexión en un momento dado.

Una aplicación llama al método Statement.close para indicar que ha terminado el proceso de una sentencia. Todos los objetos Statement se cierran cuando se cierra la conexión que los ha creado. Sin embargo, este comportamiento no es del todo fiable en lo que a cerrar objetos Statement se refiere. Por ejemplo, si la aplicación cambia de forma que se utiliza una agrupación de conexiones en lugar de cerrar explícitamente las conexiones, la aplicación “se queda” sin handles de sentencia, ya que la conexión nunca se cierra. El hecho de cerrar los objetos Statement en cuanto ya no son necesarios permite liberar inmediatamente los recursos externos de base de datos utilizados por la sentencia.

El controlador JDBC nativo intenta detectar fugas de sentencia y las maneja en nombre del usuario. Sin embargo, basarse en ese soporte provoca un rendimiento inferior.

Para utilizar sentencias y sus subclases, consulte las siguientes secciones.

Sentencias
Un objeto Statement (sentencia) se utiliza para procesar una sentencia SQL estática y obtener los resultados producidos por ella.

PreparedStatements
PreparedStatement es una subclase de la interfaz Statement que proporciona soporte para añadir parámetros a sentencias SQL.

CallableStatements
Las CallableStatements amplían la interfaz PreparedStatement y proporcionan soporte para parámetros de salida y de entrada/salida, además del soporte de parámetros de entrada proporcionado por PreparedStatement.

Debido a la jerarquía de herencia, según la cual CallableStatement amplía PreparedStatement, que a su vez amplía Statement, las características de cada una de las interfaces están disponibles en la clase que amplía la interfaz. Por ejemplo, las características de la clase Statement también están soportadas en las clases PreparedStatement y CallableStatement. La excepción principal la constituyen los métodos executeQuery, executeUpdate y execute de la clase Statement. Estos métodos toman una sentencia SQL para procesarla dinámicamente y provocan excepciones si el usuario intenta utilizarlos con objetos PreparedStatement o CallableStatement.

Sentencias: Un objeto Statement (sentencia) se utiliza para procesar una sentencia SQL estática y obtener los resultados producidos por ella. Sólo un ResultSet para cada objeto Statement puede estar abierto a la vez. Todos los métodos statement que procesan una sentencia SQL cierran implícitamente el ResultSet actual de una sentencia si existe uno abierto.

Crear sentencias: Los objetos Statement se crean a partir de objetos Connection con el método createStatement. Por ejemplo, suponiendo que ya exista un objeto Connection denominado conn, la siguiente línea de código crea un objeto Statement para pasar sentencias SQL a la base de datos:

```
Statement stmt = conn.createStatement();
```

Especificar características de ResultSet: Las características de los ResultSets están asociadas con la sentencia que finalmente los crea. El método Connection.createStatement permite especificar estas características de ResultSet. A continuación se ofrecen algunos ejemplos de llamadas válidas al método createStatement:

Ejemplo: método createStatement

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
// El siguiente código es nuevo en JDBC 2.0

Statement stmt2 = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATEABLE);

// El siguiente código es nuevo en JDBC 3.0

Statement stmt3 = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY, ResultSet.HOLD_CURSOR_OVER_COMMIT);
```

Para obtener más información acerca de estas características, consulte la sección ResultSets.

Procesar sentencias: El proceso de sentencias SQL con un objeto Statement se realiza mediante los métodos executeQuery(), executeUpdate() y execute().

Devolver resultados desde consultas SQL: Si debe procesarse una sentencia de consulta SQL que devuelva un objeto ResultSet, debe utilizarse el método executeQuery(). Puede consultar el programa de ejemplo que utiliza el método executeQuery de un objeto Statement para obtener un ResultSet.

Nota: si una sentencia SQL que se procesa con el método executeQuery no devuelve un ResultSet, se lanza una SQLException.

Devolver cuentas de actualización para sentencias SQL: Si se sabe que el código SQL es una sentencia DDL (Data Definition Language) o una sentencia DML (Data Manipulation Language) que devuelve una cuenta de actualización, debe utilizarse el método executeUpdate(). El programa StatementExample utiliza el método executeUpdate de un objeto Statement.

Procesar sentencias SQL en las que el valor de retorno esperado es desconocido: Si no se sabe cuál es el tipo de sentencia SQL, debe utilizarse el método execute. Una vez que se ha procesado este método, el controlador JDBC puede indicar a la aplicación qué tipos de resultados ha generado la sentencia SQL mediante las llamadas de API. El método execute devuelve true si el resultado es un ResultSet como mínimo, y false si el valor de retorno es una cuenta de actualización. Con esta información, las aplicaciones pueden utilizar los métodos de sentencia getUpdateCount o getResultSet para recuperar el valor de retorno del proceso de la sentencia SQL. El programa StatementExecute utiliza el método execute en un objeto Statement. Este programa espera que se pase un parámetro que sea una sentencia SQL. Sin mirar el texto de la sentencia SQL que el usuario proporciona, el programa procesa la sentencia y determina la información relativa a lo que se ha procesado.

Nota: la llamada al método getUpdateCount cuando el resultado es un ResultSet devuelve -1. La llamada al método getResultSet cuando el resultado es una cuenta de actualización devuelve nulo.

El método cancel: Los métodos del controlador JDBC nativo están sincronizados para evitar que dos hebras que se ejecutan en el mismo objeto provoquen daños en el mismo. Una excepción a esta norma la representa el método cancel. Una hebra puede utilizar el método cancel para detener una sentencia SQL

de larga ejecución en otra hebra que opera sobre el mismo objeto. El controlador JDBC nativo no puede obligar a la hebra a detenerse; sólo puede solicitarle que detenga la tarea que estaba realizando. Por esta razón, una sentencia cancelada tarda tiempo en detenerse. El método cancel puede utilizarse para detener consultas SQL incontroladas en el sistema.

PreparedStatements: Las PreparedStatements amplían la interfaz Statement y proporcionan soporte para añadir parámetros a sentencias SQL.

Las sentencias SQL que se pasan a la base de datos pasan por un proceso de dos pasos al devolver los resultados al usuario. Primero se preparan y, a continuación, se procesan. Con los objetos Statement, estas dos fases aparecen como una sola en las aplicaciones. Las PreparedStatements permiten independizar estos dos procesos. El paso de preparación se produce cuando se crea el objeto, y el paso de proceso se produce cuando se llama a los métodos executeQuery, executeUpdate o execute en el objeto PreparedStatement.

La posibilidad de dividir el proceso SQL en fases independientes no tiene sentido sin la adición de marcadores de parámetro. Los marcadores de parámetro se colocan en una aplicación para que ésta pueda indicar a la base de datos que no tiene un valor específico durante la preparación, pero que proporciona uno durante el proceso. En las sentencias SQL, los marcadores de parámetro se representan mediante signos de interrogación.

Los marcadores de parámetro posibilitan la creación de sentencias SQL generales que se utilizan para peticiones específicas. Por ejemplo, considere la siguiente sentencia de consulta SQL:

```
SELECT * FROM EMPLOYEE_TABLE WHERE LASTNAME = 'DETTINGER'
```

Se trata de una sentencia SQL específica que devuelve un solo valor; es decir, la información relativa a un empleado llamado Dettinger. Si se añade un marcador de parámetro, la sentencia puede ser más flexible:

```
SELECT * FROM EMPLOYEE_TABLE WHERE LASTNAME = ?
```

Estableciendo simplemente el marcador de parámetro en un valor, puede obtenerse información acerca de cualquier empleado de la tabla.

Las PreparedStatements proporcionan mejoras de rendimiento significativas con respecto a las Statements; el ejemplo de Statement anterior puede pasar por la fase de preparación una sola vez y, a continuación, procesarse repetidamente con valores diferentes para el parámetro.

Nota: la utilización de PreparedStatements es obligatoria para dar soporte a la agrupación de sentencias del controlador JDBC nativo.

Para obtener más información sobre cómo utilizar sentencias preparadas, incluida la creación de sentencias preparadas, especificar características del conjunto de resultados, trabajar con claves generadas automáticamente y establecer marcas de parámetros, consulte las páginas siguientes:

Crear y utilizar PreparedStatements

Procesar PreparedStatements

Ejemplo: Utilizar PreparedStatement para obtener un Result

Crear y utilizar PreparedStatements: Para crear objetos PreparedStatement nuevos, se utiliza el método prepareStatement. A diferencia de en el método createStatement, la sentencia SQL debe suministrarse al crear el objeto PreparedStatement. En ese momento, la sentencia SQL se precompila para su utilización. Por ejemplo, suponiendo que ya exista un objeto Connection, el ejemplo siguiente crea un objeto PreparedStatement y prepara la sentencia SQL para que se procese en la base de datos:

```
PreparedStatement ps = conn.prepareStatement("SELECT * FROM EMPLOYEE_TABLE
                                              WHERE LASTNAME = ?");
```

Especificar características de ResultSet y soporte de claves generado automáticamente: Al igual que el método createStatement, el método prepareStatement se ha cargado a posteriori para proporcionar soporte para la especificación de características de ResultSet. El método prepareStatement también presenta variantes para trabajar con claves generadas automáticamente. A continuación se ofrecen algunos ejemplos de llamadas válidas al método prepareStatement:

Ejemplo: método prepareStatement

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
     // Nuevo en JDBC 2.0

     PreparedStatement ps2 = conn.prepareStatement("SELECT * FROM
         EMPLOYEE_TABLE WHERE LASTNAME = ?",

ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATEABLE);

// Nuevo en JDBC 3.0

     PreparedStatement ps3 = conn.prepareStatement("SELECT * FROM
         EMPLOYEE_TABLE WHERE LASTNAME = ?",
    ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATEABLE,
    ResultSet.HOLD_CURSOR_OVER_COMMIT);

     PreparedStatement ps4 = conn.prepareStatement("SELECT * FROM
         EMPLOYEE_TABLE WHERE LASTNAME = ?", Statement.RETURN_GENERATED_KEYS);
```

Manejar parámetros: Para poder procesar un objeto PreparedStatement, debe establecerse un valor en cada uno de los marcadores de parámetro. El objeto PreparedStatement proporciona varios métodos para establecer parámetros. Todos los métodos tienen el formato set<Tipo>, siendo <Tipo> un tipo de datos Java. Son ejemplos de estos métodos setInt, setLong, setString, setTimestamp, setNull y setBlob. Casi todos estos métodos toman dos parámetros:

- El primero es el índice que el parámetro tiene dentro de la sentencia. Los marcadores de parámetro están numerados, empezando por el 1.
- El segundo parámetro es el valor que debe establecerse en el parámetro. Hay un par de métodos set<Tipo> que tienen parámetros adicionales, como el parámetro longitud del método setBinaryStream.

Consulte el Javadoc del paquete java.sql para obtener más información.Tomando la sentencia SQL preparada en los ejemplos anteriores del objeto ps, el código siguiente muestra cómo se especifica el valor de parámetro antes del proceso:

```
     ps.setString(1,'Dettinger');
```

Si se intenta procesar una PreparedStatement con marcadores de parámetro que no se han establecido, se lanza una SQLException.

Nota: una vez establecidos, los marcadores de parámetro conservan el mismo valor entre los procesos, a menos que se produzca una de las siguientes situaciones:

- Otra llamada a un método set cambia el valor.
- El valor se elimina cuando se llama al método clearParameters.

El método clearParameters identifica todos los parámetros como no establecidos. Una vez efectuada la llamada al método clearParameters, es necesario llamar de nuevo al método set en todos los parámetros antes del próximo proceso.

Soporte de ParameterMetaData: Una nueva interfaz ParameterMetaData permite recuperar información acerca de un parámetro. Este soporte es complementario de ResultSetMetaData, y es parecido. Se suministra información acerca de precisión, escala, tipos de datos, nombre de tipo de datos y si el parámetro permite el valor nulo.

Consulte el Ejemplo: ParameterMetaData que muestra cómo utilizar este nuevo soporte en un programa de aplicación.

Procesar PreparedStatements: El proceso de sentencias SQL con un objeto PreparedStatement se realiza mediante los métodos executeQuery, executeUpdate y execute, al igual que el proceso de objetos Statement. A diferencia de las versiones de Statement, no se pasan parámetros en estos métodos debido a que la sentencia SQL ya se ha suministrado al crear el objeto. Dado que PreparedStatement amplía Statement, las aplicaciones pueden intentar llamar a versiones de los métodos executeQuery, executeUpdate y execute que toman una sentencia SQL. Esta operación provoca el lanzamiento de una excepción SQLException.

Devolver resultados desde consultas SQL: Si debe procesarse una sentencia de consulta SQL que devuelva un objeto ResultSet, debe utilizarse el método executeQuery. El programa PreparedStatementExample utiliza el método executeQuery de un objeto PreparedStatement para obtener un ResultSet.

Nota: si una sentencia SQL se procesa con el método executeQuery y no devuelve un ResultSet, se lanza una SQLException.

Devolver cuentas de actualización para sentencias SQL: Si se sabe que el código SQL es una sentencia DDL (Data Definition Language) o una sentencia DML (Data Manipulation Language) que devuelve una cuenta de actualización, debe utilizarse el método executeUpdate. El programa PreparedStatementExample utiliza el método executeUpdate de un objeto PreparedStatement.

Procesar sentencias SQL en las que el valor de retorno esperado es desconocido: Si no se sabe cuál es el tipo de sentencia SQL, debe utilizarse el método execute. Una vez que se ha procesado este método, el controlador JDBC puede indicar a la aplicación qué tipos de resultados ha generado la sentencia SQL mediante las llamadas de API. El método execute devuelve true si el resultado es un ResultSet como mínimo, y false si el valor de retorno es una cuenta de actualización. Con esta información, las aplicaciones pueden utilizar los métodos de sentencia getUpdateCount o getResultSet para recuperar el valor de retorno del proceso de la sentencia SQL.

Nota: la llamada al método getUpdateCount cuando el resultado es un ResultSet devuelve -1. La llamada al método getResultSet cuando el resultado es una cuenta de actualización devuelve nulo.

CallableStatements: La interfaz CallableStatement amplía PreparedStatement y proporciona soporte para parámetros de salida y de entrada/salida. La interfaz CallableStatement tiene también soporte para parámetros de entrada, que proporciona la interfaz PreparedStatement.

La interfaz CallableStatement permite la utilización de sentencias SQL para llamar a procedimientos almacenados. Los procedimientos almacenados son programas que tienen una interfaz de base de datos. Estos programas tienen lo siguiente:

- Pueden tener parámetros de entrada y de salida, o parámetros que son tanto de entrada como de salida.
- Pueden tener un valor de retorno.
- Tienen la capacidad de devolver varios ResultSets.

Conceptualmente, en JDBC una llamada de procedimiento almacenado es una sola llamada a la base de datos, pero el programa asociado con el procedimiento almacenado puede procesar cientos de peticiones de base de datos. El programa de procedimiento almacenado también puede realizar otras diversas tareas programáticas que no realizan habitualmente las sentencias SQL.

Debido a que CallableStatements sigue el modelo de PreparedStatement de desasociar las fases de preparación y proceso, existe la posibilidad de reutilización optimizada (consulte la sección PreparedStatement para obtener detalles). Dado que las sentencias SQL de un procedimiento almacenado están enlazadas a un programa, se procesan como SQL estático, y con ello puede obtenerse un rendimiento superior. La encapsulación de gran cantidad de trabajo de base de datos en una sola llamada de base de datos reutilizable es un ejemplo de utilización óptima de procedimientos almacenados. Sólo esta llamada se transmite por la red al otro sistema, pero la petición puede realizar gran cantidad de trabajo en el sistema remoto.

Crear CallableStatements: El método prepareCall se utiliza para crear objetos CallableStatement nuevos. Al igual que en el método prepareStatement, la sentencia SQL debe suministrarse en el momento de crear el objeto CallableStatement. En ese momento, se precompila la sentencia SQL. Por ejemplo, suponiendo que ya exista un objeto Connection denominado conn, el siguiente código crea un objeto CallableStatement y realiza la fase de preparación de la sentencia SQL para que se procese en la base de datos:

```
PreparedStatement ps = conn.prepareStatement("? = CALL ADDEMPLOYEE(?, ?, ?");
```

El procedimiento almacenado ADDEMPLOYEE toma parámetros de entrada para un nuevo nombre de empleado, su número de seguridad social y el ID de usuario de su administrador. A partir de esta información, pueden actualizarse varias tablas de base de datos de la empresa con información relativa al empleado, como por ejemplo la fecha en que ha empezado a trabajar, la división, el departamento, etc. Además, un procedimiento almacenado es un programa que puede generar ID de usuario estándar y direcciones de correo electrónico para ese empleado. El procedimiento almacenado también puede enviar un correo electrónico al administrador que ha contratado al empleado con nombres de usuario y contraseñas iniciales; a continuación, el administrador puede proporcionar la información al empleado.

El procedimiento almacenado ADDEMPLOYEE está configurado para tener un valor de retorno. El código de retorno puede ser un código de éxito o error que el programa que efectúa la llamada puede utilizar cuando se produce una anomalía. El valor de retorno también puede definirse como el número de ID de empresa del nuevo empleado. Finalmente, el programa de procedimiento almacenado puede haber procesado consultas internamente y haber dejado los ResultSets de esas consultas abiertos y disponibles para el programa que efectúa la llamada. Consultar toda la información del nuevo empleado y ponerla a disposición del llamador mediante un ResultSet devuelto es una operación que tiene sentido.

En las secciones siguientes se describe cómo realizar cada uno de estos tipos de tareas.

Especificar características de ResultSet y soporte de claves generado automáticamente: Al igual que en createStatement y prepareStatement, existen varias versiones de prepareCall que proporcionan soporte para especificar características de ResultSet. A diferencia de prepareStatement, el método prepareCall no proporciona variantes para trabajar con claves generadas automáticamente a partir de CallableStatements (JDBC 3.0 no soporta este concepto). A continuación se ofrecen algunos ejemplos de llamadas válidas al método prepareCall:

Ejemplo: método prepareCall

```
// El siguiente código es nuevo en JDBC 2.0

CallableStatement cs2 = conn.prepareCall("? = CALL ADDEMPLOYEE(?, ?, ?)",
    ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATEABLE);

// Nuevo en JDBC 3.0

CallableStatement cs3 = conn.prepareCall("? = CALL ADDEMPLOYEE(?, ?, ?)",
    ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATEABLE,
    ResultSet.HOLD_CURSOR_OVER_COMMIT);
```

Manejar parámetros: Como se ha indicado, los objetos CallableStatement pueden tomar tres tipos de parámetros:

- **IN**
  Los parámetros IN se manejan de la misma forma que PreparedStatements. Los diversos métodos set de la clase PreparedStatement heredada se utilizan para establecer los parámetros.
- **OUT**
  Los parámetros OUT se manejan con el método registerOutParameter. En la forma más común de registerOutParameter, el primer parámetro es un índice de parámetro y el segundo es un tipo de SQL. Este indica al controlador JDBC qué tipo de datos debe esperar del parámetro cuando se procesa la sentencia. Existen otras dos variantes del método registerOutParameter que pueden encontrarse en el Javadoc del paquete java.sql.
- **INOUT**
  Los parámetros INOUT requieren que se realice el trabajo tanto para parámetros IN como para parámetros OUT. Para cada parámetro INOUT, debe llamarse a un método set y al método registerOutParameter para que pueda procesarse la sentencia. Si alguno de los parámetros no se establece o registra, se lanza una SQLException cuando se procesa la sentencia.

Consulte la sección Ejemplo: crear un procedimiento con parámetros de entrada y salida para obtener más información.

Al igual que en PreparedStatements, los valores de parámetro de CallableStatement permanecen estables entre los procesos, a menos que llame de nuevo a un método set. El método clearParameters no afecta a los parámetros registrados para la salida. Después de llamar a clearParameters, todos los parámetros IN deben establecerse de nuevo en un valor, pero ninguno de los parámetros OUT tiene que registrarse de nuevo.

Nota: el concepto de parámetro no debe confundirse con el de índice de un marcador de parámetro. Una llamada de procedimiento almacenado espera que se le pase un determinado número de parámetros. Una sentencia SQL determinada contiene caracteres ″?″ (marcadores de parámetro) para representar los valores que se suministrarán en el momento de la ejecución. El siguiente ejemplo muestra la diferencia que existe entre los dos conceptos:

```
CallableStatement cs = con.prepareCall("CALL PROC(?, "SECOND", ?)");

cs.setString(1, "First");     //Marcador de parámetro 1, Parámetro de
                                        //procedimiento almacenado 1

cs.setString(2, "Third");     //Marcador de parámetro 2, parámetro de
                                        //procedimiento almacenado 3
```

Acceder a los parámetros de procedimiento almacenado por nombre: Los parámetros de procedimiento almacenado tienen nombres asociados, como muestra este declaración de procedimiento almacenado:

Ejemplo: parámetros de procedimiento almacenado

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
CREATE
PROCEDURE MYLIBRARY.APROC
    (IN PARM1 INTEGER)
LANGUAGE SQL SPECIFIC MYLIBRARY.APROC
BODY: BEGIN
    <Realizar aquí una tarea...>
END BODY
```

Existe un solo parámetro de tipo integer con el nombre PARM1. En JDBC 3.0, existe soporte para especificar parámetros de procedimiento almacenado por el nombre y por índice. El código para configurar una CallableStatement para este procedimiento es el siguiente:

```
CallableStatement cs = con.prepareCall("CALL APROC(?)");

cs.setString("PARM1", 6);     //Establece el parámetro de entrada del índice 1
                              //(PARM1) en 6.
```

Para obtener más información, consulte la sección Procesar CallableStatements.

Procesar CallableStatements: El proceso de llamadas de procedimiento almacenado SQL con un objeto CallableStatement se realiza con los mismos métodos que en un objeto PreparedStatement.

Devolver resultados para procedimientos almacenados: Si una sentencia SQL se procesa dentro de un procedimiento almacenado, los resultados de la consulta pueden ponerse a disposición del programa que llama al procedimiento almacenado. También puede llamarse a varias consultas dentro del procedimiento almacenado, y el programa que efectúa la llamada puede procesar todos los ResultSets disponibles.

Consulte el Ejemplo: crear un procedimiento con varios ResultSets para obtener más información.

Nota: si un procedimiento almacenado se procesa con executeQuery y no devuelve un ResultSet, se lanza una SQLException.

Acceder a ResultSets de forma concurrente: La sección "Devolver resultados para procedimientos almacenados" trata sobre los ResultSets y los procedimientos almacenados y proporciona un ejemplo que funciona en todos los releases de Java Development Kit (JDK). En el ejemplo, los ResultSets se procesan por orden, empezando por el primer ResultSet que el procedimiento almacenado ha abierto hasta el último ResultSet abierto. El ResultSet anterior se cierra antes de utilizar el siguiente.

En JDK 1.4, existe soporte para trabajar con ResultSets de procedimientos almacenados de forma concurrente.

Nota: esta característica se añadió al soporte de sistema subyacente mediante la interfaz de línea de mandatos (CLI) en V5R2. En consecuencia, JDK 1.4 ejecutado en un sistema anterior a V5R2 no tiene disponible este soporte.

Devolver cuentas de actualización para procedimientos almacenados: La devolución de cuentas de actualización para procedimientos almacenados es una característica que se describe en la especificación JDBC, pero no está soportada actualmente en la plataforma iSeries. No existe ninguna forma de devolver varias cuentas de actualización desde una llamada de procedimiento almacenado. Si es necesaria una cuenta de actualización de una sentencia SQL procesada en un procedimiento almacenado, existen dos formas de devolver el valor:

- Devolver el valor como parámetro de salida.
- Pasar de nuevo el valor como valor de retorno del parámetro. Este es un caso especial de parámetro de salida. Consulte la sección "Procesar procedimientos almacenados que tienen valor de retorno" para obtener más información.

Procesar procedimientos almacenados en los que el valor esperado es desconocido: Si el resultado de una llamada de procedimiento almacenado no es conocido, debe utilizarse el método execute. Una vez que se ha procesado este método, el controlador JDBC puede indicar a la aplicación qué tipos de resultados ha generado el procedimiento almacenado mediante las llamadas de API. El método execute devuelve true si el resultado es uno o varios ResultSets. Las cuentas de actualización no provienen de llamadas de procedimiento almacenado.

Procesar procedimientos almacenados que tienen valor de retorno: La plataforma iSeries soporta los procedimientos almacenados que tienen un valor de retorno parecido al valor de retorno de una función. El valor de retorno de un procedimiento almacenado se etiqueta como otras marcas de parámetro, según lo asignado por la llamada de procedimiento almacenado. A continuación se ofrece un ejemplo de esta descripción:

```
? = CALL MYPROC(?, ?, ?)
```

El valor de retorno de una llamada de procedimiento almacenado es siempre un tipo integer y debe registrarse igual que cualquier otro parámetro de salida.

Consulte el Ejemplo: crear un procedimiento con valores de retorno para obtener más información.

ResultSets

La interfaz ResultSet proporciona acceso a los resultados generados al ejecutar consultas. Conceptualmente, los datos de un ResultSet pueden considerarse como una tabla con un número específico de columnas y un número específico de filas. Por omisión, las filas de la tabla se recuperan por orden. Dentro de una fila, se puede acceder a los valores de columna en cualquier orden.

Para utilizar el objeto ResultSet, consulte lo siguiente:

Características de ResultSet
Esta sección describe características de ResultSet, como por ejemplo:

- Tipos de ResultSet
- Concurrencia
- Posibilidad de cerrar el ResultSet comprometiendo el objeto Connection.
- Especificación de características de ResultSet

Movimiento de cursores
Los controladores JDBC (Java Database Connectivity) de iSeries soportan los ResultSets desplazables. Con un ResultSet desplazable, puede procesar filas de datos en cualquier orden mediante diversos métodos de posicionamiento de cursor.

Recuperar datos de ResultSet
Describe la forma en que un objeto ResultSet proporciona métodos para obtener datos de columna para una fila.

Cambiar ResultSets
Con los controladores JDBC de iSeries, puede cambiar los ResultSets realizando estas tareas:

- Actualizar filas
- Suprimir filas
- Insertar filas
- Cambiar actualizaciones posicionadas

Crear ResultSets
Puede crear un objeto ResultSet mediante los métodos executeQuery suministrados por las interfaces Statement, PreparedStatement o CallableStatement. Esta sección también describe cómo cerrar objetos ResultSet cuando ya no son necesarios en la aplicación.

Características de ResultSet: Por omisión, el tipo de todos los ResultSets creados es sólo hacia adelante, la concurrencia de sólo lectura y los cursores se retienen en los límites del compromiso. Una excepción a esta última norma la presenta WebSphere, que actualmente cambia el valor por omisión de la capacidad de retención de cursores de forma que los cursores se cierren implícitamente durante la operación de compromiso. Estas características pueden configurarse mediante los métodos accesibles en objetos Statement, PreparedStatement y CallableStatement.

Tipos de ResultSet: El tipo de un ResultSet especifica los siguiente acerca del ResultSet:

- Si el ResultSet es desplazable.
- Los tipos de ResultSets JDBC (Java Database Connectivity) definidos por constantes en la interfaz ResultSet.

Las definiciones de estos tipos de ResultSet son las siguientes:

TYPE_FORWARD_ONLY
Un cursor que sólo puede utilizarse para procesar desde el principio de un ResultSet hasta el final del mismo. Este es el tipo por omisión.

TYPE_SCROLL_INSENSITIVE
Un cursor que puede utilizarse para el desplazamiento en diversas formas a través de un ResultSet. Este tipo de cursor es insensible a los cambios efectuados en la base de datos mientras está abierto. Contiene filas que satisfacen la consulta cuando ésta se procesa o cuando se extraen datos.

TYPE_SCROLL_SENSITIVE
Un cursor que puede utilizarse para el desplazamiento en diversas formas a través de un ResultSet. Este tipo de cursor es sensible a los cambios efectuados en la base de datos mientras está abierto. Los cambios en la base de datos tienen un impacto directo sobre los datos del ResultSet.

Los ResultSets de JDBC 1.0 son siempre sólo hacia adelante. Los cursores desplazables se añadieron en JDBC 2.0.

Nota: las propiedades de agrupación por bloques habilitada y de conexión de tamaño de bloque afectan al grado de sensibilidad de un cursor TYPE_SCROLL_SENSITIVE. La agrupación por bloques mejora el rendimiento al almacenar en antememoria datos de laa propia capa del controlador JDBC.

Consulte el Ejemplo: ResultSets sensibles e insensibles, que muestra la diferencia entre los ResultSets sensibles e insensibles cuando se insertan filas en una tabla.

Consulte el Ejemplo: sensibilidad de ResultSet, que muestra cómo un cambio puede afectar directamente a una cláusula where de una sentencia SQL en función de la sensibilidad de un ResultSet.

Concurrencia: La concurrencia determina si el ResultSet puede actualizarse. Los tipos se definen de nuevo mediante constantes de la interfaz ResultSet. Los valores de concurrencia disponibles son los siguientes:

CONCUR_READ_ONLY
Un ResultSet que sólo puede utilizarse para leer datos de la base de datos. Este es el valor por omisión.

CONCUR_UPDATEABLE
Un ResultSet que permite efectuar cambios en el mismo. Estos cambios pueden colocarse en la base de datos subyacente. Consulte la sección Cambiar ResultSets para obtener más información.

Los ResultSets de JDBC 1.0 son siempre sólo hacia adelante. Los ResultSets actualizables se añadieron en JDBC 2.0.

Nota: según la especificación JDBC, el controlador JDBC puede cambiar el tipo de ResultSet del valor de concurrencia de ResultSet si los valores no pueden utilizarse conjuntamente. En tales casos, el controlador JDBC sitúa un aviso en el objeto Connection.

Existe una situación en la que la aplicación especifica un ResultSet TYPE_SCROLL_INSENSITIVE, CONCUR_UPDATEABLE. La insensibilidad se implementa en el motor de bases de datos efectuando una copia de los datos. A continuación, no se permite al usuario realizar actualizaciones mediante esa copia en la base de datos subyacente. Si especifica esta combinación, el controlador cambia la sensibilidad a TYPE_SCROLL_SENSITIVE y crea el aviso, que indica que la petición se ha cambiado.

Capacidad de retención: La característica de capacidad de retención determina si la llamada al compromiso en el objeto Connection cierra el ResultSet. La API de JDBC destinada a trabajar con la característica de capacidad de retención es nueva en la versión 3.0. Sin embargo, el controlador JDBC nativo ha suministrado una propiedad de conexión para varios releases que permite al usuario especificar ese valor

por omisión para todos los ResultSets creados bajo la conexión (consulte las secciones Propiedades de conexión y Propiedades de DataSource). El soporte de API altera temporalmente cualquier valor de la propiedad de conexión. Los valores de la característica de capacidad de retención se definen mediante constantes de ResultSet y son los siguientes:

HOLD_CURSOR_OVER_COMMIT
Todos los cursores abiertos permanecen así cuando se llama a la cláusula commit. Este es el valor por omisión del controlador JDBC nativo.

CLOSE_CURSORS_ON_COMMIT
Todos los cursores abiertos se cierran cuando se llama a la cláusula commit.

Nota: al llamar a la retrotracción en una conexión, siempre se cierran todos los cursores abiertos. Este es un hecho poco conocido, pero es una forma común de que las bases de datos manejen los cursores.

Según la especificación JDBC, el valor por omisión de la capacidad de retención está definida por la implementación. Algunas plataformas optan por utilizar CLOSE_CURSORS_ON_COMMIT como valor por omisión. Esto no representa generalmente un problema para la mayoría de las aplicaciones, pero el usuario debe estar al corriente de lo que realiza el controlador que utiliza si está trabajando con cursores en los límites del compromiso. El controlador JDBC de IBM Toolbox para Java también utiliza el valor por omisión HOLD_CURSORS_ON_COMMIT, pero el controlador JDBC de UDB para Windows NT tiene un valor por omisión CLOSE_CURSORS_ON_COMMIT.

Especificar características de ResultSet: Las características de un ResultSet no cambian una vez que se ha creado el objeto ResultSet. Por tanto, las características se han especificado antes de crear el objeto. Puede especificar estas características mediante variantes cargadas a posteriori de los métodos createStatement, prepareStatement y prepareCall.

Consulte los siguientes temas para especificar características de ResultSet:
- "Especificar características de ResultSet" en la página 81 para Statements
- Especificar características de ResultSet y soporte de claves generado automáticamente para PreparedStatement
- "Especificar características de ResultSet y soporte de claves generado automáticamente" en la página 85 para CallableStatements

Nota: Existen métodos de ResultSet para obtener el tipo de ResultSet y la concurrencia del ResultSet, pero no existe ningún método para obtener la capacidad de retención del ResultSet.

Movimiento de cursores: El método ResultSet.next se utiliza para desplazarse por filas en un ResultSet de una en una. Con JDBC (Java Database Connectivity) 2.0, los controladores JDBC de iSeries soportan los ResultSets desplazables. Los ResultSets desplazables permiten procesar las filas de datos en cualquier orden mediante los métodos previous, absolute, relative, first y last.

Por omisión, los ResultSets JDBC son siempre sólo hacia adelante, lo cual significa que el único método de posicionamiento de cursor válido al que puede llamarse es next(). Un ResultSet desplazable debe solicitarse explícitamente. Consulte la sección "Tipos de ResultSet" en la página 88 para obtener más información.

Con un ResultSet desplazable, puede utilizar los siguientes métodos de posicionamiento de cursor:

Método	Descripción
Next	Este método adelanta el cursor una fila del ResultSet. El método devuelve true si el cursor está situado en una fila válida, y false si no es así.

Método	Descripción
Previous	El método hace retroceder el cursor una fila del ResultSet. El método devuelve true si el cursor está situado en una fila válida, y false si no es así.
First	El método mueve el cursor a la primera fila del ResultSet. El método devuelve true si el cursor está situado en la primera fila, y false si el ResultSet está vacío.
Last	El método mueve el cursor a la última fila del ResultSet. El método devuelve true si el cursor está situado en la última fila, y false si el ResultSet está vacío.
BeforeFirst	El método mueve el cursor inmediatamente antes de la primera fila del ResultSet. En un ResultSet vacío, este método no tiene ningún efecto. No existe ningún valor de retorno de este método.
AfterLast	El método mueve el cursor inmediatamente después de la última fila del ResultSet. En un ResultSet vacío, este método no tiene ningún efecto. No existe ningún valor de retorno de este método.
Relative (int rows)	El método mueve el cursor en relación a su posición actual. • Si rows es 0, este método no tiene ningún efecto. • Si rows es positive, el cursor se mueve hacia adelante el número de filas indicado. Si entre la posición actual y el final del ResultSet existen menos filas que las indicadas por los parámetros de entrada, este método opera igual que afterLast. • Si rows es negative, el cursor se mueve hacia atrás el número de filas indicado. Si entre la posición actual y el final del ResultSet existen menos filas que las indicadas por el parámetro de entrada, este método opera igual que beforeFirst. El método devuelve true si el cursor está situado en una fila válida, y false si no es así.
Absolute (int row)	El método mueve el cursor a la fila especificada por el valor de fila (row). Si el valor de fila es positivo, el cursor se coloca en el número de filas indicado a partir del principio de ResultSet. El número de la primera fila es 1, el de la segunda es 2, etc. Si en el ResultSet existen menos filas que las especificadas por el valor de fila, este método opera igual que afterLast. Si el valor de fila es negativo, el cursor se coloca en el número de filas indicado a partir del final de ResultSet. El número de la última fila es -1, el de la penúltima es -2, etc. Si en el ResultSet existen menos filas que las especificadas por el valor de fila, este método opera igual que beforeLast. Si el valor de fila es 0, este método opera igual que beforeFirst. El método devuelve true si el cursor está situado en una fila válida, y false si no es así.

Recuperar datos de ResultSet: El objeto ResultSet proporciona varios métodos para obtener los datos de columna correspondientes a un fila. Todos ellos tienen el formato get<*Tipo*>, siendo <*Tipo*> un tipo de datos Java. Algunos ejemplos de estos métodos son getInt, getLong, getString, getTimestamp y getBlob. Casi todos estos métodos toman un solo parámetro, que es el índice que la columna tiene dentro del ResultSet o bien el nombre de la columna.

Las columnas de ResultSet están numeradas, empezando por el 1. Si se emplea el nombre de la columna y hay más de una columna que tenga ese mismo nombre en el ResultSet, se devuelve la primera. Algunos de los métodos get<Tipo> tienen parámetros adicionales, como el objeto opcional Calendar, que se puede pasar a los métodos getTime, getDate y getTimestamp. Consulte el Javadoc del paquete java.sql para obtener todos los detalles.

En los métodos get que devuelven objetos, el valor de retorno es null cuando la columna del ResultSet es nula. En tipos primitivos, no puede devolverse null. En estos casos, el valor es 0 o false. Si una aplicación debe distinguir entre null, y 0 o false, puede utilizarse el método wasNull inmediatamente después de la llamada. A continuación, este método puede determinar si el valor era un valor 0 o false real o si ese valor se ha devuelto debido a que el valor de ResultSet era de hecho null.

Consulte el Ejemplo: interfaz ResultSet para IBM Developer Kit para Java para obtener un ejemplo de utilización de la interfaz ResultSet.

Soporte de ResultSetMetaData: Cuando se llama al método getMetaData en un objeto ResultSet, el método devuelve un objeto ResultSetMetaData que describe las columnas de ese objeto ResultSet. En los casos en que la sentencia SQL que se va a procesar no se conoce hasta el momento de la ejecución, puede utilizarse ResultSetMetaData para determinar cuál de los métodos get hay que emplear para recuperar los datos. El ejemplo de código siguiente utiliza ResultSetMetaData para determinar cada uno de los tipos de columna del conjunto de resultados.

Ejemplo: utilizar ResultSetMetaData para determinar cada tipo de columna de un conjunto de resultados

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
ResultSet rs = stmt.executeQuery(sqlString);
ResultSetMetaData rsmd = rs.getMetaData();
int colType [] = new int[rsmd.getColumnCount()];
for (int idx = 0, int col = 1; idx < colType.length; idx++, col++)
colType[idx] = rsmd.getColumnType(col);
```

Consulte el Ejemplo: interfaz ResultSetMetaData para IBM Developer Kit para Java para obtener un ejemplo de utilización de la interfaz ResultSetMetaData.

Cambiar ResultSets: El valor por omisión para ResultSets es de sólo lectura. Sin embargo, con JDBC (Java Database Connectivity) 2.0, los controladores JDBC de iSeries proporcionan soporte completo para ResultSets actualizables. Puede consultar el valor de “Concurrencia” en la página 89 de ResultSet para conocer cómo actualizar ResultSets.

Actualizar filas: Pueden actualizarse las filas de una tabla de base de datos mediante la interfaz ResultSet. Los pasos que implica este proceso son los siguientes:

1. Cambie los valores de una fila específica mediante los diversos métodos update<*Tipo*>, donde <*Tipo*> es un tipo de datos Java. Estos métodos update<*Tipo*> corresponden al los métodos get<*Tipo*> disponibles para recuperar valores.
2. Aplique las filas a la base de datos subyacente.

La propia base de datos no se actualiza hasta el segundo paso. Si se actualizan columnas de un ResultSet sin llamar al método updateRow, no se realizan cambios en la base de datos.

Las actualizaciones planificadas en una fila pueden eliminarse con el método cancelUpdates. Una vez que se ha llamado al método updateRow, los cambios de la base de datos son finales y no pueden deshacerse.

Nota: el método rowUpdated siempre devuelve false, ya que la base de datos no tiene forma alguna de señalar qué filas se han actualizado. En consecuencia, el método updatesAreDetected devuelve false.

Suprimir filas: Pueden suprimirse las filas de una tabla de base de datos mediante la interfaz ResultSet. Se suministra el método deleteRow, que suprime la fila actual.

Insertar filas: Pueden insertarse filas en una tabla de base de datos mediante la interfaz ResultSet. Este proceso utiliza una operación ″insert row″ (insertar fila), a la que las aplicaciones mueven específicamente el cursor y crean los valores que desean insertar en la base de datos. Los pasos que implica este proceso son los siguientes:

1. Sitúe el cursor en la operación insert row.
2. Establezca cada uno de los valores para las columnas de la fila nueva.
3. Inserte la fila en la base de datos y, opcionalmente, mueva el cursor de nuevo a la fila actual de ResultSet.

Nota: no se insertan filas nuevas en la tabla en la que está situado el cursor. Generalmente, se añaden al final del espacio de datos de la tabla. Por omisión, una base de datos relacional no depende de la posición. Por ejemplo, no debe esperar mover el cursor a la tercera fila e insertar algo que aparezca antes de la cuarta fila si usuarios subsiguientes extraen los datos.

Soporte para actualizaciones posicionadas: Además del método destinado a actualizar la base de datos mediante un ResultSet, pueden utilizarse sentencias SQL para emitir actualizaciones posicionadas. Este soporte se basa en la utilización de cursores con nombre. JDBC suministra el método setCursorName de Statement y el método getCursorName de ResultSet para proporcionar acceso a estos valores.

Dos métodos de DatabaseMetaData, supportsPositionedUpdated y supportsPositionedDelete, devuelven true, ya que esta característica está soportada por el controlador JDBC nativo.

Consulte el Ejemplo: cambiar valores con una sentencia mediante el cursor de otra sentencia para obtener más información.

Consulte el Ejemplo: eliminar valores de una tabla mediante el cursor de otra sentencia para obtener más información.

Crear ResultSets: Para crear un objeto ResultSet, puede utilizar los métodos executeQuery de las interfaces Statement, PreparedStatement o CallableStatement. Sin embargo, existen otros métodos disponibles. Por ejemplo, los métodos DatabaseMetaData, como por ejemplo getColumns, getTables, getUDTs, getPrimaryKeys, etc., devuelven ResultSets. También es posible que una sola sentencia SQL devuelva varios ResultSets para el proceso. También puede utilizar el método getResultSet para recuperar un objeto ResultSet después de llamar al método execute suministrado por las interfaces Statement, PreparedStatement o CallableStatement.

Consulte el Ejemplo: crear un procedimiento con varios ResultSets para obtener más información.

Cerrar ResultSets: Aunque un objeto ResultSet se cierra automáticamente cuando se cierra el objeto Statement con el que está asociado, es aconsejable cerrar los objetos ResultSet cuando haya terminado de utilizarlos. Al hacerlo, liberará inmediatamente los recursos internos de base de datos que pueden aumentar el rendimiento de las aplicaciones.

También es importante cerrar los objetos ResultSet generados por llamadas a DatabaseMetaData. Debido a que el usuario no tiene acceso directo al objeto Statement utilizado para crear estos ResultSets, no se llama directamente a close en el objeto Statement. Estos objetos están enlazados conjuntamente de tal forma que el controlador JDBC cierra el objeto Statement interno cuando el usuario cierra el objeto ResultSet externo. Si estos objetos no se cierran manualmente, el sistema sigue en funcionamiento; sin embargo, utiliza más recursos de los necesarios.

Nota: la característica de capacidad de retención de ResultSets puede cerrar también los ResultSets automáticamente en nombre del usuario. Se permite llamar a close varias veces en un objeto ResultSet.

Agrupación de objetos JDBC

La agrupación de objetos es la cuestión más habitual que surge en la discusión sobre JDBC (Java Database Connectivity) y el rendimiento. Debido a que la creación de muchos objetos utilizados en JDBC

es costosa, como por ejemplo objetos Connection, Statement y ResultSet, pueden obtenerse ventajas significativas de rendimiento reutilizando estos objetos en lugar de crearlos cada vez que son necesarios.

Muchas aplicaciones ya manejan la agrupación de objetos por su cuenta. Por ejemplo, WebSphere tiene un amplio soporte para agrupar objetos JDBC y permite al usuario controlar la gestión de la agrupación. Debido a ello, el usuario puede obtener las funciones que desee sin necesidad de preocuparse de sus propios mecanismos de agrupación. Sin embargo, en las ocasiones en que no se suministra soporte, es necesario encontrar una solución para todas las aplicaciones, excepto las triviales.

Para utilizar la agrupación de objetos en los programas JDBC, consulte las siguientes secciones:

Utilizar el soporte de DataSource para la agrupación de objetos
Puede utilizar DataSources para que varias aplicaciones compartan una configuración común para acceder a una base de datos. Esta operación se realiza haciendo que cada una de las aplicaciones haga referencia al mismo nombre de DataSource.

Propiedades de ConnectionPoolDataSource
Puede configurar la interfaz ConnectionPoolDataSource utilizando el conjunto de propiedades que proporciona.

Agrupación de sentencias basada en DataSource
Puede utilizar la agrupación de sentencias dentro de una agrupación de conexiones. La propiedad maxStatements de la interfaz UDBConnectionPoolDataSource permite a DataSource especificar cuántas sentencias pueden agruparse bajo una conexión.

Crear su propia solución de agrupación
Puede desarrollar su propia agrupación de conexiones y sentencias sin necesidad de contar con soporte para DataSources ni confiar en otro producto.

Utilizar el soporte de DataSource para la agrupación de objetos: La utilización de DataSources permite que varias aplicaciones compartan una configuración común para acceder a la base de datos. Esta operación se realiza haciendo que cada una de las aplicaciones haga referencia al mismo nombre de DataSource.

Mediante la utilización de DataSources, pueden cambiarse muchas aplicaciones desde una ubicación central. Por ejemplo, si cambia el nombre de una biblioteca por omisión utilizada por todas las aplicaciones y ha utilizado un solo DataSource para obtener conexiones para todas ellas, puede actualizar el nombre de la colección en ese dataSource. A continuación, todas las aplicaciones empezarán a utilizar la nueva biblioteca por omisión.

Al utilizar DataSources para obtener conexiones para una aplicación, puede utilizar el soporte incorporado del controlador JDBC nativo para la agrupación de conexiones. Este soporte se suministra como implementación de la interfaz ConnectionPoolDataSource.

La agrupación se realiza pasando objetos lógicos Connection en lugar de objetos físicos Connection. Un **objeto lógico Connection** es un objeto de conexión devuelto por un objeto Connection de la agrupación. Cada objeto lógico de conexión actúa como handle temporal para la conexión física representada por el objeto de conexión de la agrupación. Con respecto a la aplicación, cuando se devuelve el objeto Connection no existe ninguna diferencia apreciable entre los dos. La ligera diferencia surge cuando se llama el método close en el objeto Connection. Esta llamada invalida la conexión lógica y devuelve la conexión física a la agrupación, donde otra aplicación puede utilizarla. Esta técnica permite que muchos objetos de conexión lógica reutilicen un solo objeto de conexión física.

Configurar la agrupación de conexiones: La agrupación de conexiones se realiza creando un objeto DataSource que hace referencia a un objeto ConnectionPoolDataSource. Los objetos ConnectionPoolDataSource tienen propiedades que pueden establecerse para manejar diversos aspectos del mantenimiento de la agrupación.

Consulte el ejemplo que muestra cómo configurar la agrupación de conexiones con UDBDataSource y UDBConnectionPoolDataSource para obtener más detalles. También puede consultar la sección Java Naming and Directory Interface (JNDI) para obtener detalles acerca del papel que juega JNDI en este ejemplo.

En el ejemplo, el enlace que conecta los dos objetos DataSource es dataSourceName. El enlace indica al objeto DataSource que retrase el establecimiento de las conexiones con el objeto ConnectionPoolDataSource que gestiona la agrupación automáticamente.

Aplicaciones con agrupación y sin agrupación: No existe ninguna diferencia entre una aplicación que utiliza la agrupación de conexiones y una que no lo hace. Por tanto, el soporte de agrupación puede añadirse una vez completado el código de la aplicación, sin efectuar cambios en el mismo.

Consulte el Ejemplo: probar el rendimiento de una agrupación de conexiones para obtener más detalles.

A continuación se muestra la salida de la ejecución local del programa anterior durante el desarrollo.

```
Iniciar temporización de la versión de DataSource sin agrupación...
Tiempo transcurrido: 6410

Iniciar temporización de la versión con agrupación...
Tiempo transcurrido: 282

Programa Java completado.
```

Por omisión, un objeto UDBConnectionPoolDataSource agrupa una sola conexión. Si una aplicación necesita una conexión varias veces y sólo necesita una conexión a la vez, la utilización de UDBConnectionPoolDataSource es una solución perfecta. Si necesita muchas conexiones simultáneas, debe configurar ConnectionPoolDataSource para que se ajuste a sus necesidades y recursos.

Propiedades de ConnectionPoolDataSource: La interfaz ConnectionPoolDataSource proporciona un conjunto de propiedades para su configuración. En la tabla siguiente figuran las descripciones de estas propiedades.

Propiedad	Descripción
initialPoolSize	Cuando se crea la primera instancia de la agrupación, esta propiedad determina cuántas conexiones se colocan en la agrupación. Si este valor se especifica fuera del rango de minPoolSize y maxPoolSize, se utiliza minPoolSize o maxPoolSize como número inicial de conexiones que deben crearse.

Propiedad	Descripción
maxPoolSize	A medida que se utiliza la agrupación, pueden solicitarse más conexiones que las que se encuentran en la agrupación. Esta propiedad especifica el número máximo de conexiones permitidas que pueden crearse en la agrupación. Las aplicaciones no se ″bloquean″ esperando a que se devuelva una conexión a la agrupación cuando ésta se encuentra en el tamaño máximo y todas las conexiones se están utilizando. En lugar de ello, el controlador JDBC crea una conexión nueva basada en las propiedades de DataSource y devuelve la conexión. Si se especifica un valor 0 en maxPoolSize, se permite que la agrupación crezca ilimitadamente siempre y cuando el sistema tenga recursos disponibles.
minPoolSize	Los picos de utilización de la agrupación pueden provocar que aumente el número de conexiones que se encuentran en ella. Si el nivel de actividad disminuye hasta el punto que algunas conexiones nunca se extraen de la agrupación, los recursos se están ocupando sin ninguna razón. En tales casos, el controlador JDBC tiene la capacidad de liberar algunas de las conexiones que ha acumulado. Esta propiedad permite indicar a JDBC que libere conexiones, asegurando que siempre exista un número determinado de conexiones disponibles para el uso. Si se especifica un valor 0 en minPoolSize, es posible que la agrupación libere todas sus conexiones y que la aplicación ″pague″ realmente por el tiempo de conexión en cada petición de conexión.
maxIdleTime	Las conexiones realizan el seguimiento del tiempo que han permanecido en la agrupación sin que se las utilice. Esta propiedad especifica el tiempo durante el que una aplicación permite que las conexiones permanezcan sin utilizar antes de liberarlas (es decir, existan más conexiones de las necesarias). Esta propiedad expresa un tiempo en segundos y no especifica cuándo se produce el cierre real. Especifica cuándo ha pasado el tiempo suficiente para que la conexión se libere.
propertyCycle	Esta propiedad representa el número de segundos que pueden transcurrir entre la entrada en vigor de estas normas.

Nota: si se establece en 0 el tiempo de maxIdleTime o propertyCycle, significa que el controlador JDBC no comprueba que las conexiones se eliminen de la agrupación por su cuenta. Las normas especificadas para el tamaño inicial, mínimo y máximo siguen en vigor.

Si maxIdleTime y propertyCycle no son 0, se utiliza una hebra de gestión para efectuar la observación de la agrupación. La hebra está a la escucha de cada segundo de propertyCycle y comprueba todas las conexiones de la agrupación para ver cuáles han permanecido en ella sin utilizarse durante más segundos que los indicados en maxIdleTime. Las conexiones que se ajustan a estos criterios se eliminan de la agrupación hasta que se alcanza el valor indicado en minPoolSize.

Agrupación de sentencias basada en DataSource: Otra propiedad disponible en la interfaz UDBConnectionPoolDataSource es maxStatements. Esta propiedad permite la agrupación de sentencias dentro de la agrupación de conexiones. La agrupación de sentencias sólo tiene efecto sobre PreparedStatements y CallableStatements. Los objetos Statement no se agrupan.

La implementación de la agrupación de sentencias es parecida a la de la agrupación de conexiones. Cuando la aplicación llama a Connection.prepareStatement("select * from tablex"), el módulo de agrupación comprueba si el objeto Statement ya se ha preparado bajo la conexión. Si es así, se pasa al usuario un objeto lógico PreparedStatement en lugar del objeto físico. Al llamar al método close, el objeto Connection se devuelve a la agrupación, el objeto lógico Connection se elimina y el objeto Statement puede reutilizarse.

La propiedad maxStatements permite a DataSource especificar cuántas sentencias pueden agruparse en una conexión. El valor 0 indica que no debe utilizarse la agrupación de sentencias. Cuando la agrupación de sentencias está llena, se aplica un algoritmo de menor utilización reciente para determinar qué sentencia debe eliminarse.

En el Ejemplo: probar el rendimiento de dos DataSources se prueba un DataSource que utiliza sólo la agrupación de conexiones y otro DataSource que utiliza la agrupación de sentencias y conexiones

El ejemplo siguiente muestra la salida de la ejecución local de este programa durante el desarrollo.

```
Despliegue del origen de datos de agrupación de sentencias
Iniciar temporización de la única versión de la agrupación de conexiones...
Tiempo transcurrido: 26312

Iniciar temporización de la versión de la agrupación de sentencias...
Tiempo transcurrido: 2292
Programa Java completado
```

Crear su propia agrupación de conexiones: Puede desarrollar su propia agrupación de conexiones y sentencias sin necesidad de contar con soporte para DataSources ni confiar en otro producto.

Las técnicas de agrupación se muestran en una pequeña aplicación Java, pero son igualmente aplicables a servlets o a aplicaciones grandes de n capas. Este ejemplo se utiliza para mostrar las cuestiones relativas al rendimiento.

La aplicación de muestra tiene dos funciones:

- Insertar un nombre y un índice nuevo en una tabla de base de datos.
- Leer el nombre para un índice determinado desde la tabla.

El código completo de la aplicación puede bajarse desde la página Web de JDBC de Developer Kit para Java

.

El rendimiento de la aplicación de ejemplo no es bueno. La ejecución de 100 llamadas al método getValue y 100 llamadas al método putValue en este código ocupa un promedio de 31,86 segundos en una estación de trabajo estándar.

El problema es que se produce demasiado trabajo en la base de datos para cada petición. Es decir, se obtiene una conexión, se obtiene una sentencia, se procesa la sentencia, se cierra la sentencia y se cierra la conexión. En lugar de descartar todos los elementos después de cada petición, debe existir una forma de reutilizar partes de este proceso. La **agrupación de conexiones** sustituye el código de creación de

conexiones por código destinado a obtener una conexión de la agrupación y, a continuación, sustituye el código de cierre de la conexión por código destinado a devolver la conexión a la agrupación para utilizarla.

El constructor de la agrupación de conexiones crea las conexiones y las sustituye en la agrupación. La clase de agrupación contiene métodos take y put para localizar una conexión que debe utilizarse y para devolver la conexión a la agrupación cuando se ha terminado de trabajar con ella. Estos métodos están sincronizados debido a que el objeto de agrupación es un recurso compartido, pero no es deseable que varias hebras intenten manipular simultáneamente los recursos agrupados.

Se ha producido un cambio en el código de llamada del método getValue. No se muestra el método putValue, pero el cambio exacto se ha realizado en éste, y está disponible en la página Web de JDBC de Developer Kit para Java de IBM.

. La creación de instancias del objeto de agrupación de conexiones tampoco se muestra. Puede llamar al constructor y pasar el número de objetos de conexión que desea que contenga la agrupación. Este paso debe realizarse al iniciar la aplicación.

La ejecución de la aplicación anterior (es decir, con 100 peticiones al método getValue y 100 al método putValue) con estos cambios ocupó un promedio de 13,43 segundos con el código de la agrupación de conexiones implementado. El tiempo de proceso para la carga de trabajo se ha recortado en más de la mitad con respecto al tiempo de proceso original sin agrupación de conexiones.

Crear su propia agrupación de sentencias: Al utilizar la agrupación de conexiones, se emplea tiempo en la creación y cierre de la sentencia cuando se proceso cada una de ellas. Este es otro ejemplo de desperdicio de un objeto que puede reutilizarse.

Para reutilizar un objeto, puede utilizar la clase de sentencia preparada. En la mayoría de aplicaciones, las mismas sentencias SQL se reutilizan con cambios secundarios. Por ejemplo, una iteración a través de una aplicación puede generar la consulta siguiente:

```
SELECT * from employee where salary > 100000
```

La próxima iteración puede generar la consulta siguiente:

```
SELECT * from employee where salary > 50000
```

Se trata de la misma consulta, pero utiliza un parámetro diferente. Ambas consultas pueden realizarse con la consulta siguiente:

```
SELECT * from employee where salary > ?
```

A continuación, puede establecer el marcador de parámetro (indicado por el signo de interrogación) en 100000 al procesar la primera consulta y en 50000 al procesar la segunda. Con ello mejorará el rendimiento por tres razones, más allá de lo que la agrupación de conexiones puede ofrecer:

- Se crean menos objetos. Se crea un objeto PreparedStatement y se reutiliza, en lugar de crear un objeto Statement para cada petición. Por tanto, se ejecutan menos constructores.
- El trabajo de la base de datos destinado a definir la sentencia SQL (denominado **preparación**) puede reutilizarse. La preparación de sentencias SQL es considerablemente costosa, ya que implica determinar lo que indica el texto de la sentencia SQL y la forma en que el sistema debe realizar la tarea solicitada.
- Al eliminar las creaciones de objetos adicionales, se produce una ventaja que con frecuencia no se tiene en cuenta. No es necesario destruir lo que no se ha creado. Este modelo facilita la función del recolector de basura de Java y también beneficia al rendimiento a lo largo de tiempo cuando existen muchos usuarios.

Las conexiones del programa de ejemplo pueden cambiarse por objetos PreparedStatement de agrupación. El hecho de cambiar el programa permite reutilizar más objetos y mejorar el rendimiento. Puede empezar por escribir la clase que contiene los objetos que deben agruparse. Esta clase debe encapsular los diversos recursos que deben utilizarse. En el ejemplo de agrupación de conexiones, Connection es el único recurso de la agrupación, y por tanto no es necesario ninguna clase de encapsulación. Cada objeto de la agrupación debe contener un objeto Connection y dos PreparedStatements. A continuación, puede crear una clase de agrupación que contenga objetos de acceso a la base de datos en lugar de conexiones.

Finalmente, la aplicación debe cambiarse para que obtenga un objeto de acceso a la base de datos y especifique qué recurso del objeto desea utilizar. Aparte de especificar el recurso específico, la aplicación permanece sin cambios.

Con este cambio, la misma ejecución de prueba ocupa ahora un promedio de 0,83 segundos. Este tiempo es aproximadamente 38 veces más rápido que el de la versión original del programa.

Consideraciones: El rendimiento mejora a través de la réplica. Si un elemento no se reutiliza, el hecho de colocarlo en la agrupación significa malgastar recursos.

La mayoría de aplicaciones contienen secciones de código de vital importancia. Generalmente, una aplicación utiliza entre el 80 y el 90 por ciento de su tiempo de proceso en sólo entre el 10 y el 20 por ciento del código. Si en una aplicación pueden utilizarse potencialmente 10.000 sentencias SQL, no todas ellas se colocan en una agrupación. El objetivo es identificar y agrupar las sentencias SQL que se utilizan en las secciones de código de la aplicación que son de vital importancia.

La creación de objetos en una implementación Java puede implicar un gran costo. La solución de agrupación puede utilizarse ventajosamente. Los objetos utilizados en el proceso se crean al principio, antes de que otros usuarios intenten utilizar el sistema. Estos objetos se reutilizan con la frecuencia que sea necesaria. El rendimiento es excelente, y es posible ajustar el rendimiento de la aplicación a lo largo del tiempo para facilitar su utilización por parte de un número mayor de usuarios. Como resultado, se agrupa un mayor número de objetos. Más aún, permite una ejecución multihebra más eficaz del acceso a base de datos de la aplicación para mejorar el rendimiento.

Java (con JDBC) se basa en SQL dinámico y tiende a ser lento. La agrupación puede minimizar este problema. Al preparar las sentencias durante el inicio, al acceso a la base de datos puede convertirse en estático. Una vez preparada la sentencia, existe poca diferencia en cuanto al rendimiento entre el SQL estático y el dinámico.

El rendimiento del acceso a bases de datos en Java puede ser eficaz y puede realizarse sin sacrificar el diseño orientado a objetos o la capacidad de mantenimiento del código. Escribir código para crear una agrupación de sentencias y conexiones no es difícil. Además, el código puede cambiarse y mejorarse para dar soporte a varias aplicaciones y tipos de aplicaciones (basadas en la Web, cliente/servidor), etc.

Actualizaciones por lotes

Una nueva característica de JDBC 2.0 es el soporte de actualización por lotes. Esta característica permite pasar las actualizaciones de la base de datos como una sola transacción entre el programa de usuario y la base de datos. Este procedimiento puede mejorar significativamente el rendimiento cuando deben realizarse muchas actualizaciones simultáneamente. Por ejemplo, si una empresa grande necesita que sus nuevos empleados empiecen a trabajar un Lunes, este requisito hace necesario procesar muchas actualizaciones (en este caso, inserciones) en la base de datos de empleados simultáneamente. Creando un lote de actualizaciones y sometiéndolas a la base de datos como una unidad, puede ahorrar tiempo de proceso.

Existen dos tipos de actualizaciones por lotes:

- Actualizaciones por lotes que utilizan objetos Statement.
- Actualizaciones por lotes que utilizan objetos PreparedStatement.

Para utilizar el soporte de actualización por lotes, consulte la sección:

Actualización por lotes de Statement
Antes de realizar una actualización por lotes de tipo statement, debe asegurarse de que el compromiso automático está desactivado. Si el valor de compromiso automático está desactivado, puede crear un objeto Statement estándar. A continuación, puede añadir las sentencias al lote con el método addBatch.Una vez añadidas todas las sentencias que desea al lote, puede procesarlas todas con el método executeBatch o vaciar el lote en cualquier momento con el método clearBatch.

Actualización por lotes de PreparedStatement
Un lote preparedStatement es parecido al lote Statement. Sin embargo, un lote preparedStatement funciona siempre sobre la misma sentencia preparada, y el usuario sólo cambia los parámetros de esa sentencia.

BatchUpdateException
Cuando falla una llamada al método executeBatch, se lanza una BatchUpdateException. BatchUpdateException permite llamar a los mismos métodos a los que ha llamado siempre para recibir el mensaje, SQLState y el código del proveedor. BatchUpdateException también proporciona el método getUpdateCounts, que devuelve una matriz de enteros. La matriz de enteros contiene cuentas de actualización de todas las sentencias del lote que se han procesado hasta el punto en el que se ha producido la anomalía.

Soporte de inserción por bloques
La inserción por bloques es una operación iSeries para insertar varias filas en una tabla de base de datos simultáneamente.

Actualización por lotes de Statement: Para realizar una actualización por lotes de Statement, debe desactivar el compromiso automático. En Java Database Connectivity (JDBC), el compromiso automático está activado por omisión. El compromiso automático significa que las actualizaciones de la base de datos se comprometen después de procesar cada sentencia SQL. Si desea tratar un grupo de sentencias que se manejan en la base de datos como un grupo funcional, no deseará que la base de datos comprometa cada sentencia individualmente. Si no desactiva el compromiso automático y falla una sentencia situada en medio del lote, no podrá retrotraer la totalidad del lote e intentarlo de nuevo, ya que la mitad de las sentencias se han convertido en finales. Además, el trabajo adicional de comprometer cada sentencia de un lote crea una gran cantidad de carga de trabajo. Consulte la sección Transacciones para obtener más detalles.

Después de desactivar el compromiso automático, puede crear un objeto Statement estándar. En lugar de procesar sentencias con métodos como por ejemplo executeUpdate, se añaden al lote con el método addBatch. Una vez añadidas todas las sentencias que desea al lote, puede procesarlas todas con el método executeBatch. Puede vaciar el lote en cualquier momento con el método clearBatch.

El ejemplo siguiente muestra cómo puede utilizar estos métodos:

Ejemplo: actualización por lotes de Statement

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
connection.setAutoCommit(false);
Statement statement = connection.createStatement();
statement.addBatch("INSERT INTO TABLEX VALUES(1, 'Cujo')");
statement.addBatch("INSERT INTO TABLEX VALUES(2, 'Fred')");
statement.addBatch("INSERT INTO TABLEX VALUES(3, 'Mark')");
int [] counts = statement.executeBatch();
connection.commit();
```

En este ejemplo, se devuelve una matriz de enteros desde el método executeBatch. Esta matriz tiene un valor entero para cada sentencia que se procesa en el lote. Si se insertan valores en la base de datos, el valor de cada sentencia es 1 (suponiendo que el proceso sea satisfactorio). Sin embargo, algunas de las sentencias pueden ser sentencias de actualización (update) que afectan a varias filas. Si coloca en el lote sentencias que no son INSERT, UPDATE o DELETE, se produce una excepción.

Actualización por lotes de PreparedStatement: Un lote preparedStatement es parecido al lote Statement; sin embargo, un lote preparedStatement funciona siempre sobre la misma sentencia ″preparada″, y el usuario sólo cambia los parámetros para dicha sentencia. A continuación se ofrece un ejemplo que utiliza un lote preparedStatement.

Ejemplo: actualización por lotes de PreparedStatement

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
connection.setAutoCommit(false);
PreparedStatement statement =
  connection.prepareStatement("INSERT INTO TABLEX VALUES(?, ?)");
statement.setInt(1, 1);
statement.setString(2, "Cujo");
statement.addBatch();
statement.setInt(1, 2);
statement.setString(2, "Fred");
statement.addBatch();
statement.setInt(1, 3);
statement.setString(2, "Mark");
statement.addBatch();
int [] counts = statement.executeBatch();
connection.commit();
```

BatchUpdateException: Una consideración importante acerca de las actualizaciones por lotes es la acción que debe realizarse cuando falla una llamada al método executeBatch. En este caso, se lanza un nuevo tipo de excepción, denominada BatchUpdateException. BatchUpdateException es una subclase de SQLException que permite llamar a los mismos métodos a los que ha llamado siempre para recibir el mensaje, SQLState y el código del proveedor. BatchUpdateException también proporciona el método getUpdateCounts, que devuelve una matriz de enteros. La matriz de enteros contiene cuentas de actualización de todas las sentencias del lote que se han procesado hasta el punto en el que se ha producido la anomalía. La longitud de la matriz indica qué sentencia del lote ha fallado. Por ejemplo, si la matriz devuelta en la excepción tiene una longitud de tres, la cuarta sentencia del lote ha fallado. Por tanto, a partir del único objeto BatchUpdateException devuelto puede determinar las cuentas de actualización para todas las sentencias que han sido satisfactorias, qué sentencia ha fallado y toda la información acerca de la anomalía.

Actualmente, el rendimiento estándar del proceso de actualizaciones por lotes es equivalente al rendimiento del proceso de cada sentencia por separado. Puede consultar la sección Soporte de inserción por bloques para obtener más información acerca del soporte optimizado para actualizaciones por lotes. Debe seguir utilizando el nuevo modelo al codificar y sacar provecho de futuras optimizaciones del rendimiento.

Nota: En la especificación JDBC 2.1, se suministra una opción diferente para el manejo de condiciones de excepción para actualizaciones por lotes. JDBC 2.1 presenta un modelo en el que el proceso por lotes continúa después de que falle una entrada del lote. Se sitúa una cuenta de actualización especial en la matriz de enteros de cuenta de actualización que se devuelve por cada entrada que falla. Esto permite que lotes de gran tamaño se continúen procesando aunque falle una de sus entradas. Consulte la especificación JDBC 2.1 o JDBC 3.0 para obtener detalles acerca de estas dos modalidades de operación. Por omisión, el controlador JDBC nativo utiliza la definición de JDBC 2.0. El controlador proporciona una propiedad Connection que se utiliza al utilizar DriverManager para establecer las conexiones. También

proporciona una propiedad DataSource que se utiliza al utilizar DataSources para establecer las conexiones. Estas propiedades permiten a las aplicaciones elegir cómo desean que las operaciones por lotes manejen las anomalías.

Soporte de inserción por bloques: Una **inserción por bloques** es un tipo especial de operación en un servidor iSeries que proporciona una forma altamente optimizada de insertar varias filas en una tabla de base de datos simultáneamente. Las inserciones por bloques pueden considerarse como un subconjunto de actualizaciones por lotes. Las actualizaciones por lotes pueden ser cualquier forma de petición de actualización, pero las inserciones por bloques son específicas. Sin embargo, los tipos de inserción por bloques de actualizaciones por lotes son comunes; el controlador JDBC nativo se ha modificado para sacar partido de esta característica.

Debido a las restricciones del sistema al utilizar el soporte de inserción por bloques, el valor por omisión para el controlador JDBC nativo es tener la inserción por bloques inhabilitada. Puede habilitarse mediante una propiedad Connection de una propiedad DataSource. La mayoría de las restricciones de utilización de una inserción por bloques pueden comprobarse y manejarse en nombre del usuario, pero no así algunas restricciones; esta es la razón por la que el soporte de inserción por bloques esté desactivado por omisión. La lista de restricciones es la siguiente:

- La sentencia SQL utilizada debe ser una sentencia INSERT con una cláusula VALUES, indicando que no es una sentencia INSERT con SUBSELECT. El controlador JDBC reconoce esta restricción y realiza las acciones adecuadas.
- Debe utilizarse una PreparedStatement, indicando que no existe soporte optimizado para objetos Statement. El controlador JDBC reconoce esta restricción y realiza las acciones adecuadas.
- La sentencia SQL debe especificar marcadores de parámetro para todas las columnas de la tabla. Esto significa que no puede utilizar valores de constante para una columna ni permitir que la base de datos inserte valores por omisión para ninguna de las columnas. El controlador JDBC no tiene ningún mecanismo para manejar las pruebas de marcadores de parámetro específicos en la sentencia SQL. Si establece la propiedad para realizar inserciones por bloques optimizadas y no evita los valores por omisión o constantes en las sentencias SQL, los valores que terminen en la tabla de base de datos no serán correctos.
- La conexión debe establecerse con el sistema local. Esto significa que no puede utilizarse una conexión que utilice DRDA para acceder a un sistema remoto, debido a que DRDA no soporta una operación de inserción por bloques. El controlador JDBC no tiene ningún mecanismo para manejar las pruebas de una conexión con un sistema local. Si establece la propiedad para realizar una inserción por bloques optimizada e intenta conectarse con un sistema remoto, el proceso de la actualización por lotes fallará.

Este ejemplo de código muestra cómo habilitar el soporte para el proceso de inserción por bloques. La única diferencia entre este código y una versión que no utilizara el soporte de inserción por bloques es `use block insert=true` que se añade al URL de conexión.

Ejemplo: Proceso de inserción por bloques

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
// Crear una conexión de base de datos
Connection c = DriverManager.getConnection("jdbc:db2:*local;use block insert=true");
BigDecimal bd = new BigDecimal("123456");

// Crear una PreparedStatement para insertarla en una tabla con 4 columnas
PreparedStatement ps =
  c.prepareStatement("insert into cujosql.xxx values(?, ?, ?, ?)");

// Iniciar temporización...
for (int i = 1; i <= 10000; i++) {
  ps.setInt(1, i);                 // Establecer todos los parámetros para una fila
  ps.setBigDecimal(2, bd);
  ps.setBigDecimal(3, bd);
```

```
  ps.setBigDecimal(4, bd);
  ps.addBatch();                   // Añadir los parámetros al proceso por lotes
}

// Procesar los lotes
int[] counts = ps.executeBatch();

// Finalizar temporización...
```

En pruebas similares, una inserción por bloques es varias veces más rápida que realizar las mismas operaciones sin utilizar una inserción por bloques. Por ejemplo, la prueba realizada en el código anterior es nueve veces más rápida con la utilización de inserciones por bloques. En los casos en que sólo se utilizan tipos nativos en lugar de objetos, la velocidad puede ser hasta dieciséis veces mayor. En las aplicaciones en las que existe una cantidad significativa de trabajo en ejecución, las expectativas deben modificarse adecuadamente.

Tipos de datos avanzados

Existen varios tipos de datos nuevos denominados tipos de datos SQL3 que se suministran en la base de datos de iSeries con el e-PACK de V4R4. Java Database Connectivity (JDBC) 2.0 y superior proporciona soporte para trabajar con estos tipos de datos que forman parte del estándar SQL99.

Los tipos de datos SQL3 proporcionan un enorme nivel de flexibilidad. Son perfectos para almacenar objetos Java serializados, documentos XML (Extensible Markup Language) y datos multimedia, como por ejemplo, canciones, imágenes de producto, fotografías de empleados y clips de vídeo.

Tipos distintivos (distinct): El **tipo distintivo** es un tipo definido por usuario que se basa en un tipo de base de datos estándar. Por ejemplo, puede definir un tipo de Número de Seguridad Social (Social Security Number), SSN, que internamente sea CHAR(9). La siguiente sentencia SQL crea un tipo DISTINCT de ese tipo.

```
CREATE DISTINCT TYPE CUJOSQL.SSN AS CHAR(9)
```

Un tipo distintivo se correlaciona siempre con un tipo de datos incorporado. Para obtener más información acerca de cómo, cuándo y dónde utilizar tipos distintivos en el contexto de SQL, consulte los manuales de SQL.

Para utilizar tipos distintivos en JDBC, se accede a ellos de la misma forma que a un tipo subyacente. El método getUDTs es un método nuevo que permite consultar los tipos distintivos que están disponibles en el sistema. Este programa de ejemplo muestra lo siguiente:

- La creación de un tipo distintivo.
- La creación de una tabla que lo utiliza.
- La utilización de una PreparedStatement para establecer un parámetro de tipo distintivo.
- La utilización de un ResultSet para devolver un tipo distintivo.
- La utilización de la llamada de la API (Interfaz del programa de aplicación) de metadatos a getUDTs para obtener información acerca de un tipo distintivo.

Para obtener más información, consulte el ejemplo siguiente que muestra diversas tareas comunes que puede realizar utilizando tipos distintivos:

Ejemplo: Tipos distintivos

Objetos grandes: Existen tres tipos de Objetos grandes (LOB):

- Objetos grandes binarios (BLOB)
- Objetos grandes de tipo carácter (CLOB)
- Objetos grandes de caracteres de doble byte (DBCLOB)

Los DBCLOB son parecidos a los CLOB, excepto en su representación de almacenamiento interno de los datos de tipo carácter. Debido a que Java y JDBC externalizan todos los datos de tipo carácter en forma de Unicode, en JDBC sólo existe soporte para los CLOB. Los DBCLOB funcionan de forma intercambiable con el soporte de CLOB desde una perspectiva JDBC.

Objetos grandes binarios: En muchos aspectos, una columna de Objeto grande binario (BLOB) es parecida a una columna CHAR FOR BIT DATA que puede convertirse en grande. En estas columnas puede almacenar cualquier objeto que pueda representarse como una corriente de bytes no convertidos. Las columnas BLOB se utilizan con frecuencia para almacenar objetos Java serializados, imágenes, canciones y otros datos binarios.

Puede utilizar los BLOB de la misma forma que otros tipos de base de datos estándar. Puede pasarlos a procedimientos almacenados, utilizarlos en sentencias preparadas y actualizarlos en conjuntos de resultados. La clase PreparedStatement contiene un método setBlob para pasar BLOB a la base de datos, y la clase ResultSet añade una clase getBlob para recuperarlos de la base de datos. Un BLOB se representa en un programa Java mediante un objeto BLOB que es una interfaz JDBC.

Consulte el apartado Escribir código que utilice BLOB para obtener más información acerca de cómo utilizar los BLOB.

Objetos grandes de tipo carácter: Los Objetos grandes de tipo carácter (CLOB) son el complemento de datos de tipo carácter de los BLOB. En lugar de almacenar datos en la base de datos sin conversión, los datos se almacenan en la base de datos en forma de texto y se procesan de la misma forma que una columna CHAR. Al igual que para los BLOB, JDBC 2.0 proporciona funciones para tratar directamente con los CLOB. La interfaz PreparedStatement contiene un método setClob y la interfaz ResultSet contiene un método getClob.

Consulte el apartado Escribir código que utilice CLOB para obtener más información acerca de cómo utilizar los CLOB.

Aunque las columnas BLOB y CLOB funcionan como las columnas CHAR FOR BIT DATA y CHAR, así es como funcionan conceptualmente desde la perspectiva del usuario externo. Internamente, son distintos; debido al tamaño potencialmente enorme de las columnas de Objeto grande (LOB), generalmente se trabaja indirectamente con los datos. Por ejemplo, cuando se extrae un bloque de filas de la base de datos, no se mueve un bloque de LOB al ResultSet. En lugar de ello, se mueven punteros denominados localizadores de LOB (es decir, enteros de cuatro bytes) a ResultSet. Sin embargo, no es necesario, tener conocimientos acerca de los localizadores al trabajar con los LOB en JDBC.

Enlaces de datos (Datalinks): Los **enlaces de datos** son valores encapsulados que contienen una referencia lógica de la base de datos a un archivo almacenado fuera de la misma. Los enlaces de datos se representan y utilizan desde una perspectiva JDBC de dos maneras diferentes, dependiendo de si se utiliza JDBC 2.0 o anterior, o si se utiliza JDBC 3.0 o posterior.

Consulte el apartado Escribir código que utilice Enlaces de datos para obtener más información acerca de cómo utilizar los Enlaces de datos.

Tipos de datos SQL3 no soportados: Existen otros tipos de datos SQL3 que se han definido y para los que la API de JDBC proporciona soporte. Son ARRAY, REF y STRUCT. Actualmente, los servidores iSeries no soportan estos tipos. Por tanto, el controlador JDBC no proporciona ninguna forma de soporte para ellos.

Escribir código que utilice BLOB: Existen diversas tareas que pueden realizarse con columnas BLOB (Gran objeto binario) de base de datos mediante la API (Interfaz de programación de aplicaciones) de Java Database Connectivity (JDBC). Los temas que siguen describen brevemente estas tareas e incluyen ejemplos de cómo realizarlas.

Leer los BLOB de la base de datos e insertar BLOB en la base de datos: Con la API de JDBC, existen formas de extraer los BLOB de la base de datos y formas de colocar BLOB en la base de datos. Sin embargo, no existe ninguna forma estandarizada para crear un objeto Blob. Esto no representa ningún problema si la base de datos ya está llena de BLOB, pero sí lo es si desea trabajar con los BLOB desde cero mediante JDBC. En lugar de definir un constructor para las interfaces Blob y Clob de la API de JDBC, se proporciona soporte para colocar los BLOB en la base de datos y extraerlos de la base de datos directamente como otros tipos. Por ejemplo, el método setBinaryStream puede trabajar con una columna de base de datos de tipo Blob. Este ejemplo muestra algunas de las formas comunes de colocar un BLOB en la base de datos o recuperarlo de la misma.

Trabajar con la API de objeto Blob: Los BLOB se definen en JDBC como una interfaz de la que los diversos controladores proporcionan implementaciones. Esta interfaz contiene una serie de métodos que pueden utilizarse para interactuar con el objeto Blob. Este ejemplo muestra algunas de las tareas comunes que pueden realizarse utilizando esta API. Consulte el Javadoc de JDBC para obtener una lista completa de los métodos disponibles en el objeto Blob.

Utilizar el soporte de JDBC 3.0 para actualizar BLOB: En JDBC 3.0 existe soporte para efectuar cambios en objetos LOB. Estos cambios pueden almacenarse en columnas BLOB de la base de datos. Este ejemplo muestra algunas de las tareas que pueden realizarse con el soporte BLOB de JDBC 3.0.

Escribir código que utilice CLOB: Existen diversas tareas que pueden realizarse con columnas CLOB y DBCLOB de base de datos mediante la API (Interfaz de programación de aplicaciones) de Java Database Connectivity (JDBC). Los temas que siguen describen brevemente estas tareas e incluyen ejemplos de cómo realizarlas.

Leer los CLOB de la base de datos e insertar CLOB en la base de datos: Con la API de JDBC, existen formas de extraer los CLOB de la base de datos y formas de colocar CLOB en la base de datos. Sin embargo, no existe ninguna forma estandarizada para crear un objeto Clob. Esto no representa ningún problema si la base de datos ya está llena de CLOB, pero sí lo es si desea trabajar con los CLOB desde cero mediante JDBC. En lugar de definir un constructor para las interfaces Blob y Clob de la API de JDBC, se proporciona soporte para colocar los CLOB en la base de datos y extraerlos de la base de datos directamente como otros tipos. Por ejemplo, el método setCharacterStream puede trabajar con una columna de base de datos de tipo Clob. Este ejemplo muestra algunas de las formas comunes de colocar un CLOB en la base de datos o recuperarlo de la misma.

Trabajar con la API de objeto Clob: Los CLOB se definen en JDBC como una interfaz de la que los diversos controladores proporcionan implementaciones. Esta interfaz contiene una serie de métodos que pueden utilizarse para interactuar con el objeto Clob. Este ejemplo muestra algunas de las tareas comunes que pueden realizarse utilizando esta API. Consulte el Javadoc de JDBC para obtener una lista completa de los métodos disponibles en el objeto Clob.

Utilizar el soporte de JDBC 3.0 para actualizar CLOB: En JDBC 3.0 existe soporte para efectuar cambios en objetos LOB. Estos cambios pueden almacenarse en columnas CLOB de la base de datos. Este ejemplo muestra algunas de las tareas que pueden realizarse con el soporte CLOB de JDBC 3.0.

Escribir código que utilice Enlaces de datos (Datalinks): La forma de trabajar con enlaces de datos depende del release con el que se trabaja. En JDBC 3.0, existe soporte para trabajar directamente con columnas Datalink mediante los métodos getURL y putURL. En las versiones anteriores de JDBC, era necesario trabajar con columnas Datalink como si fueran columnas String. Actualmente, la base de datos no soporta las conversiones automáticas entre Datalink y tipos de datos de carácter. En consecuencia, es necesario realizar una conversión temporal de tipos en las sentencias SQL.

Este ejemplo muestra algunas de las tareas básicas del trabajo con columnas Datalink.

RowSets

Los RowSets se añadieron en principio al paquete opcional de JDBC (Java Database Connectivity) 2.0. A diferencia de algunas de las interfaces más conocidas de la especificación JDBC, la especificación RowSet

está diseñada más como infraestructura que como implementación real. Las interfaces RowSet definen un conjunto de funciones centrales que comparten todos los RowSets. Los proveedores de la implementación RowSet tienen una libertad considerable para definir las funciones necesarias para satisfacer sus necesidades en un espacio de problemas específico.

Para implementar Rowsets mediante el controlador JDBC nativo, consulte las siguientes secciones:

Características de RowSet
Puede solicitar determinadas propiedades que los RowSets deben satisfacer. Las propiedades comunes incluyen el conjunto de interfaces a las que el rowset resultante debe dar soporte.

DB2JdbcRowSet
DB2JdbcRowSet es un RowSet conectado que funciona como envoltorio de un DB2ResultSet y proporciona soporte para el manejo de eventos.

DB2CachedRowSet
DB2CachedRowSet es un RowSet desconectado que permite almacenar datos de DB2ResultSet dentro del objeto. Una vez que los datos están dentro del objeto, el objeto DB2Connection subyacente puede cerrarse y el DB2CachedRowSet puede seguir utilizándose. Consulte la siguiente información acerca de DB2CachedRowSet:

- Utilizar DB2CachedRowSets
- Crear y llenar un DB2CachedRowSet
- Acceso a datos de DB2CachedRowSet y manipulación de cursores
- Cambiar datos de DB2CachedRowSet y reflejar de nuevo los cambios en el origen de datos
- Otras características de DB2CachedRowSet

Características de RowSet: Puede solicitar determinadas propiedades que los rowsets deben satisfacer. Las propiedades comunes incluyen el conjunto de interfaces a las que el rowset resultante debe dar soporte.

Los RowSets son ResultSets: La interfaz RowSet amplía la interfaz ResultSet, lo que significa que los RowSets tienen la capacidad de realizar todas las funciones que los ResultSets pueden efectuar. Por ejemplo, los RowSets pueden ser desplazables y actualizables.

Los RowSets pueden desconectarse de la base de datos: Existen dos categorías de RowSets:

- **Conectado**
  Aunque los RowSets conectados se llenan con datos, siempre tienen conexiones internas con la base de datos subyacente abierta y funcionan como envoltorios alrededor de una implementación de ResultSet.
- **Desconectado**
  No es necesario que los RowSets desconectados mantengan siempre conexiones con su origen de datos. Los RowSets desconectados pueden desconectarse de la base de datos, utilizarse de diversas formas y, a continuación, reconectarse a la base de datos para reflejar los cambios efectuados en ellos.

Los RowSets son componentes JavaBeans: Los RowSets tienen soporte para el manejo de eventos basado en el modelo de manejo de eventos de JavaBeans. También tienen propiedades que pueden establecerse. El RowSet puede utilizar estas propiedades para realizar las siguientes operaciones:

- Establecer una conexión con la base de datos.
- Procesar una sentencia SQL.
- Determinar características de los datos que el RowSet representa y manejar otras características internas del objeto RowSet.

Los RowSets son serializables: Los RowSets pueden serializarse y deserializarse para permitirles fluir a través de una conexión de red, escribirlos en un archivo plano (es decir, en un documento de texto sin proceso de texto ni otros caracteres de estructura), etc.

DB2CachedRowSet: El objeto DB2CachedRowSet es un RowSet desconectado, lo que significa que puede utilizarse sin estar conectado a la base de datos. Su implementación es muy parecida a la descripción de un CachedRowSet.

DB2CachedRowSet es un contenedor para filas de datos de un ResultSet. DB2CachedRowSet contiene la totalidad de sus propios datos, de forma que no necesita mantener una conexión con la base de datos aparte de la explícitamente necesaria para leer o escribir datos en la misma.

Utilizar DB2CachedRowSets
Puede utilizar métodos suministrados por DB2CachedRowSet para mejorar el rendimiento de la base de datos, permitiendo que varios usuarios utilicen los mismos datos. También puede entregar ResultSets comunes a los clientes creando una copia de los datos de tabla que no sufren cambios.

Crear y llenar un DB2CachedRowSet
Averigüe cómo crear y colocar datos en un DB2CachedRowSet mediante estas tareas:

- Utilizar el método populate
- Utilizar propiedades DB2CachedRowSet y DataSources
- Utilizar propiedades de DB2CachedRowSet y los URL de JDBC
- Utilizar el método setConnection(Connection) para utilizar una conexión de base de datos existente
- Utilizar el método execute(Connection) para utilizar una conexión de base de datos existente
- Utilizar el método execute(int) para agrupar peticiones de base de datos

Acceso a datos de DB2CachedRowSet y manipulación de cursores
Los RowSets dependen de métodos de ResultSet. En muchas operaciones, como por ejemplo acceso a datos DB2CachedRowSet y movimiento de cursores, no hay ninguna diferencia a nivel de aplicación entre utilizar un ResultSet y utilizar un RowSet.

Cambiar datos de DB2CachedRowSet y reflejar de nuevo los cambios en el origen de datos
DB2CachedRowSet utiliza los mismos métodos que la interfaz ResultSet estándar para efectuar cambios en los datos del objeto RowSet. DB2CachedRowSet proporciona el método acceptChanges, que se utiliza para reflejar de nuevo los cambios de RowSet en la base de datos de donde proceden los datos.

Otras características de DB2CachedRowSet
La clase DB2CachedRowSet tiene algunas funciones adicionales que hacen más flexible su utilización. Con los métodos suministrados por DB2CachedRowSet, puede realizar las siguientes tareas:

- Obtener colecciones a partir de DB2CachedRowSets
- Crear copias de RowSets
- Crear compartimientos para RowSets

Utilizar DB2CachedRowSet: Debido a que el objeto DB2CachedRowSet puede desconectarse y serializarse, resulta de utilidad en entornos donde no siempre es práctico ejecutar un controlador JDBC completo (por ejemplo, en asistentes digitales personales (PDA) y teléfonos móviles habilitados para Java).

Dado que el objeto DB2CachedRowSet se encuentra en memoria y sus datos siempre se conocen, puede funcionar como una forma altamente optimizada de ResultSet desplazable para las aplicaciones. Mientras que los DB2ResultSets desplazables pagan habitualmente un precio de rendimiento debido a que sus movimientos aleatorios interfieren con la capacidad del controlador JDBC para almacenar en antememoria filas de datos, los RowSets no presentan este problema.

En DB2CachedRowSet se suministran dos métodos que crean nuevos RowSets:

- El método createCopy crea un nuevo RowSet que es idéntico al copiado.
- El método createShared crea un nuevo RowSet que comparte los mismos datos subyacentes que el original.

Puede utilizar el método createCopy para entregar ResultSets comunes a los clientes. Si los datos de tabla no cambian, crear una copia de un RowSet y pasarla a cada cliente es más eficaz que ejecutar cada vez una consulta en la base de datos.

Puede utilizar el método createShared para mejorar el rendimiento de la base de datos permitiendo que varios usuarios utilicen los mismos datos. Por ejemplo, suponga que dispone de un sitio Web que muestra los veinte productos más vendidos en la página de presentación cuando un cliente se conecta. Desea que la información de la página principal se actualice periódicamente, pero ejecutar la consulta para obtener los productos adquiridos con mayor frecuencia cada vez que un cliente visita la página principal no resulta práctico. Mediante el método createShared, puede de hecho crear ″cursores″ para cada cliente sin necesidad de procesar de nuevo la consulta ni almacenar una enorme cantidad de información en la memoria. Cuando proceda, puede ejecutarse de nuevo la consulta para buscar los productos adquiridos con mayor frecuencia. Los datos nuevos pueden llenar el RowSet utilizado para crear los cursores compartidos y los servlets pueden utilizarlos.

Los DB2CachedRowSets proporcionan una característica de proceso retardado. Esta característica permite agrupar varias peticiones de consulta y procesarlas en la base de datos como una sola petición. Este es un ejemplo de “Utilizar el método execute(int) para agrupar peticiones de base de datos” en la página 111 para eliminar parte de la sobrecarga de cálculo a la que estaría sometida la base de datos.

Debido a que RowSet debe efectuar un cuidadoso seguimiento de los cambios que se producen en sí mismo para que se reflejen de nuevo en la base de datos, existe soporte para funciones que deshacen los cambios o permiten al usuario ver todos los cambios que se han efectuado. Por ejemplo, existe un método showDeleted que puede utilizarse para indicar al RowSet que permita al usuario extraer filas suprimidas. También existen los métodos cancelRowInsert y cancelRowDelete para deshacer inserciones y supresiones de filas, respectivamente, después de efectuarlas.

El objeto DB2CachedRowSet ofrece una mejor interoperatividad con otras API Java debido a su soporte de manejo de eventos y a sus métodos toCollection, que permiten convertir un RowSet o parte de él en una colección Java.

El soporte de manejo de eventos de DB2CachedRowSet puede utilizarse en aplicaciones de interfaz gráfica de usuario (GUI) para controlar pantallas, anotar información acerca de los cambios de RowSet a medida que se realizan o buscar información relativa a los cambios en orígenes que no sean RowSets. Consulte el Ejemplo: eventos DB2JdbcRowSet para obtener detalles.

Para obtener detalles específicos acerca del trabajo con DB2CachedRowSets, consulte los siguientes temas:

- Crear y llenar un DB2CachedRowSet
- Acceso a datos de DB2CachedRowSet y manipulación de cursores
- Cambiar datos de DB2CachedRowSet y reflejar de nuevo los cambios en el origen de datos
- Otras características de DB2CachedRowSet

Para obtener información sobre el modelo y el manejo de eventos, consulte DB2JdbcRowSet, ya que este soporte funciona de forma idéntica para ambos tipos de RowSets.

Crear y llenar un DB2CachedRowSet: Existen varias formas de colocar datos en un DB2CachedRowSet:

- “Utilizar el método populate” en la página 109
- “Utilizar propiedades DB2CachedRowSet y DataSources” en la página 109
- “Utilizar propiedades DB2CachedRowSet y los URL de JDBC” en la página 110

- "Utilizar el método setConnection(Connection) para utilizar una conexión de base de datos existente" en la página 110
- "Utilizar el método execute(Connection) para utilizar una conexión de base de datos existente" en la página 111
- "Utilizar el método execute(int) para agrupar peticiones de base de datos" en la página 111

Utilizar el método populate: Los DB2CachedRowSets tienen un método populate que puede utilizarse para colocar datos en el RowSet desde un objeto DB2ResultSet. A continuación se ofrece un ejemplo de este método.

Ejemplo: utilizar el método populate

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
// Establecer una conexión con la base de datos.
Connection conn = DriverManager.getConnection("jdbc:db2:*local");

// Crear una sentencia y utilizarla para realizar una consulta.
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("select col1 from cujosql.test_table");

// Crear y llenar un DB2CachedRowSet desde ella.
DB2CachedRowSet crs = new DB2CachedRowSet();
crs.populate(rs);

// Nota: desconectar los objetos ResultSet, Statement
// y Connection utilizados para crear el RowSet.
rs.close();
stmt.close();
conn.close();

// Bucle de los datos del RowSet.
while (crs.next()) {
  System.out.println("v1 es " + crs.getString(1));
}

crs.close();
```

Utilizar propiedades DB2CachedRowSet y DataSources: Los DB2CachedRowSets tienen propiedades que permiten a los DB2CachedRowSets aceptar una consulta SQL y un nombre DataSource. A continuación, utilizan la consulta SQL y el nombre DataSource para crear datos para sí mismos. A continuación se ofrece un ejemplo de este método. Se supone que la referencia al DataSource denominado BaseDataSource es un DataSource válido que se ha configurado anteriormente.

Ejemplo: utilizar propiedades DB2CachedRowSet y DataSources

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
// Crear un nuevo DB2CachedRowSet
DB2CachedRowSet crs = new DB2CachedRowSet();

// Establecer las propiedades necesarias para que
// el RowSet utilice un DataSource para llenarse.
crs.setDataSourceName("BaseDataSource");
crs.setCommand("seleccionar col1 de cujosql.test_table");

// Llamar al método execute de RowSet. Esto provoca que
// el RowSet utilice el DataSource y la consulta SQL
// especificada para llenarse con datos. Una vez
// que se ha llenado el RowSet, se desconecta de la base de datos.
crs.execute();
```

```
// Bucle de los datos del RowSet.
while (crs.next()) {
  System.out.println("v1 es " + crs.getString(1));
}

// Finalmente, cerrar el RowSet.
crs.close();
```

Utilizar propiedades DB2CachedRowSet y los URL de JDBC: Los DB2CachedRowSets tienen propiedades que permiten a los DB2CachedRowSets aceptar una consulta SQL y un URL de JDBC. A continuación, utilizan la consulta y el URL de JDBC para crear datos para sí mismos. A continuación se ofrece un ejemplo de este método.

Ejemplo: Utilizar propiedades DB2CachedRowSet y los URL de JDBC

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
// Crear un nuevo DB2CachedRowSet
DB2CachedRowSet crs = new DB2CachedRowSet();

// Establecer las propiedades necesarias para que
// el RowSet utilice un URL de JDBC para llenarse.
crs.setUrl("jdbc:db2:*local");
crs.setCommand("seleccionar col1 de cujosql.test_table");

// Llamar al método execute de RowSet. Esto provoca que
// el RowSet utilice el DataSource y la consulta SQL
// especificada para llenarse con datos. Una vez
// que se ha llenado el RowSet, se desconecta de la base de datos.
crs.execute();

// Bucle de los datos del RowSet.
while (crs.next()) {
  System.out.println("v1 es " + crs.getString(1));
}

// Finalmente, cerrar el RowSet.
crs.close();
```

Utilizar el método setConnection(Connection) para utilizar una conexión de base de datos existente: Para promocionar la reutilización de objetos JDBC Connection, DB2CachedRowSet proporciona un mecanismo para pasar un objeto Connection establecido al DB2CachedRowSet utilizado para llenar el RowSet. Si se pasa un objeto Connection suministrado por usuario, el DB2CachedRowSet no lo desconecta después de llenarse.

Ejemplo: utilizar el método setConnection(Connection) para utilizar una conexión de base de datos existente

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
// Establecer una conexión JDBC con la base de datos.
Connection conn = DriverManager.getConnection("jdbc:db2:*local");

// Crear un nuevo DB2CachedRowSet
DB2CachedRowSet crs = new DB2CachedRowSet();

// Establecer las propiedades necesarias para que
//el RowSet utilice una conexión ya establecida
// para llenarse.
crs.setConnection(conn);
crs.setCommand("seleccionar col1 de cujosql.test_table");
```

```
// Llamar al método execute de RowSet. Esto provoca que
// el RowSet utilice la conexión que se le suministró
// anteriormente. Una vez que el RowSet se ha llenado, no
// cierra la conexión suministrada por el usuario.
crs.execute();

// Bucle de los datos del RowSet.
while (crs.next()) {
  System.out.println("v1 es " + crs.getString(1));
}

// Finalmente, cerrar el RowSet.
crs.close();
```

Utilizar el método execute(Connection) para utilizar una conexión de base de datos existente: Para promocionar la reutilización de objetos JDBC Connection, DB2CachedRowSet proporciona un mecanismo para pasar un objeto Connection establecido al DB2CachedRowSet cuando se llama al método execute. Si se pasa un objeto Connection suministrado por usuario, el DB2CachedRowSet no lo desconecta después de llenarse.

Ejemplo: utilizar el método execute(Connection) para utilizar una conexión de base de datos existente

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
// Establecer una conexión JDBC con la base de datos.
Connection conn = DriverManager.getConnection("jdbc:db2:*local");

// Crear un nuevo DB2CachedRowSet
DB2CachedRowSet crs = new DB2CachedRowSet();

// Establecer la sentencia SQL que debe utilizarse para
// llenar el RowSet.
crs.setCommand("seleccionar col1 de cujosql.test_table");

// Llamar al método execute de RowSet, pasando la conexión
// que debe utilizarse. Una vez que el Rowset se ha llenado, no
// cierra la conexión suministrada por el usuario.
crs.execute(conn);

// Bucle de los datos del RowSet.
while (crs.next()) {
  System.out.println("v1 es " + crs.getString(1));
}

// Finalmente, cerrar el RowSet.
crs.close();
```

Utilizar el método execute(int) para agrupar peticiones de base de datos: Para reducir la carga de trabajo de la base de datos, DB2CachedRowSet proporciona un mecanismo para agrupar sentencias SQL de varias hebras en una sola petición de proceso de la base de datos.

Ejemplo: utilizar el método execute(int) para agrupar peticiones de base de datos

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
// Crear un nuevo DB2CachedRowSet
DB2CachedRowSet crs = new DB2CachedRowSet();

// Establecer las propiedades necesarias para que
// el RowSet utilice un DataSource para llenarse.
crs.setDataSourceName("BaseDataSource");
```

```
crs.setCommand("seleccionar col1 de cujosql.test_table");

// Llamar al método execute de RowSet. Esto provoca que
// el RowSet utilice el DataSource y la consulta SQL
// especificada para llenarse con datos. Una vez
// que se ha llenado el RowSet, se desconecta de la base de datos.
// Esta versión del método execute acepta el número de segundos
// que sean necesarios para esperar los resultados. Al
// permitir un retardo, el RowSet puede agrupar las peticiones
// de varios usuarios y procesar la petición en
// la base de datos subyacente una sola vez.
crs.execute(5);

// Bucle de los datos del RowSet.
while (crs.next()) {
  System.out.println("v1 es " + crs.getString(1));
}

// Finalmente, cerrar el RowSet.
crs.close();
```

Acceso a datos de DB2CachedRowSet y manipulación de cursores: Los RowSets dependen de métodos de ResultSet. En muchas operaciones, como por ejemplo "Acceso a datos de DB2CachedRowSet" y "Manipulación de cursores" en la página 114, no hay ninguna diferencia a nivel de aplicación entre utilizar un ResultSet y utilizar un RowSet.

Acceso a datos de DB2CachedRowSet: RowSets y ResultSets acceden a los datos de la misma manera. En el ejemplo siguiente, el programa crea una tabla y la llena con diversos tipos de datos mediante JDBC. Una vez que la tabla está preparada, se crea un DB2CachedRowSet y se llena con la información de la tabla. El ejemplo también utiliza diversos métodos get de la clase RowSet.

Ejemplo: acceso a datos de DB2CachedRowSet

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
import java.sql.*;
import javax.sql.*;
import com.ibm.db2.jdbc.app.*;
import java.io.*;
import java.math.*;

public class TestProgram
{
   public static void main(String args[])
  {
   // Se registra el controlador.
            try {
      Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
    }
    catch (ClassNotFoundException ex) {
      System.out.println("ClassNotFoundException: " +
                 ex.getMessage());
      // No es necesario continuar.
            System.exit(1);
    }

            try {
      Connection conn = DriverManager.getConnection("jdbc:db2:*local");

      Statement stmt = conn.createStatement();
```

```
  // Borrar ejecuciones anteriores
        try {
  stmt.execute("eliminar tabla cujosql.test_table");
  }
catch (SQLException ex) {
  System.out.println("Capturado eliminar tabla: " + ex.getMessage());
  }

  // Crear tabla de prueba
  stmt.execute("Crear tabla cujosql.test_table (col1 smallint, col2 int, " +
          "col3 bigint, col4 real, col5 float, col6 double, col7 numeric, " +
          "col8 decimal, col9 char(10), col10 varchar(10), col11 date, " +
          "col12 time, col13 timestamp)");
  System.out.println("Tabla creada.");

  // Insertar algunas filas de prueba
  stmt.execute("insertar en valores de cujosql.test_table (1, 1, 1, 1.5, 1.5, 1.5, 1.5, 1.5, 'one',
                'one', {d '2001-01-01'}, {t '01:01:01'}, {ts '1998-05-26 11:41:12.123456'})");

  stmt.execute("insertar en valores de cujosql.test_table (null, null, null, null, null, null, null,
                null, null, null, null, null, null)");
  System.out.println("Filas insertadas");

  ResultSet rs = stmt.executeQuery("select * from cujosql.test_table");
  System.out.println("Consulta ejecutada");

  // Crear un nuevo conjunto de filas (rowset) y llenarlo...
  DB2CachedRowSet crs = new DB2CachedRowSet();
  crs.populate(rs);
  System.out.println("RowSet lleno.");

conn.close();
  System.out.println("RowSet se desconecta...");

  System.out.println("Probar con getObject");
  int count = 0;
while (crs.next()) {
    System.out.println("Fila " + (++count));
    for (int i = 1; i <= 13; i++) {
      System.out.println("  Col " + i + " valor " + crs.getObject(i));
    }
  }

  System.out.println("Probar con getXXX... ");
  crs.first();
  System.out.println("Fila 1");
  System.out.println("  Valor Col 1 " + crs.getShort(1));
  System.out.println("  Valor Col 2 " + crs.getInt(2));
  System.out.println("  Valor Col 3 " + crs.getLong(3));
  System.out.println("  Valor Col 4 " + crs.getFloat(4));
  System.out.println("  Valor Col 5 " + crs.getDouble(5));
  System.out.println("  Valor Col 6 " + crs.getDouble(6));
  System.out.println("  Valor Col 7 " + crs.getBigDecimal(7));
  System.out.println("  Valor Col 8 " + crs.getBigDecimal(8));
  System.out.println("  Valor Col 9 " + crs.getString(9));
  System.out.println("  Valor Col 10 " + crs.getString(10));
  System.out.println("  Valor Col 11 " + crs.getDate(11));
  System.out.println("  Valor Col 12 " + crs.getTime(12));
  System.out.println("  Valor Col 13 " + crs.getTimestamp(13));
  crs.next();
  System.out.println("Fila 2");
  System.out.println("  Valor Col 1 " + crs.getShort(1));
  System.out.println("  Valor Col 2 " + crs.getInt(2));
  System.out.println("  Valor Col 3 " + crs.getLong(3));
  System.out.println("  Valor Col 4 " + crs.getFloat(4));
  System.out.println("  Valor Col 5 " + crs.getDouble(5));
```

```
      System.out.println("  Valor Col 6 " + crs.getDouble(6));
      System.out.println("  Valor Col 7 " + crs.getBigDecimal(7));
      System.out.println("  Valor Col 8 " + crs.getBigDecimal(8));
      System.out.println("  Valor Col 9 " + crs.getString(9));
      System.out.println("  Valor Col 10 " + crs.getString(10));
      System.out.println("  Valor Col 11 " + crs.getDate(11));
      System.out.println("  Valor Col 12 " + crs.getTime(12));
      System.out.println("  Valor Col 13 " + crs.getTimestamp(13));

      crs.close();
    }
    catch (Exception ex) {
      System.out.println("SQLException: " + ex.getMessage());
                 ex.printStackTrace();
    }
  }
}
```

Manipulación de cursores: Los RowSets son desplazables y actúan exactamente igual que un ResultSet desplazable. En el ejemplo siguiente, el programa crea una tabla y la llena con datos mediante JDBC. Una vez que la tabla está preparada, se crea un objeto DB2CachedRowSet y se llena con la información de la tabla. El ejemplo también utiliza diversas funciones de manipulación de cursores.

Ejemplo: manipulación de cursores

```
import java.sql.*;
import javax.sql.*;
import com.ibm.db2.jdbc.app.DB2CachedRowSet;

public class RowSetSample1
{
   public static void main(String args[])
  {
   // Se registra el controlador.
            try {
      Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
    }
    catch (ClassNotFoundException ex) {
      System.out.println("ClassNotFoundException: " +
                 ex.getMessage());
      // No es necesario continuar.
            System.exit(1);
  }

            try {
    Connection conn = DriverManager.getConnection("jdbc:db2:*local");

    Statement stmt = conn.createStatement();

      // Borrar ejecuciones anteriores
            try {
      stmt.execute("eliminar tabla cujosql.test_table");
    }
    catch (SQLException ex) {
      System.out.println("Capturado eliminar tabla: " + ex.getMessage());
    }

    // Crear una tabla de prueba
    stmt.execute("Crear tabla cujosql.test_table (col1 smallint)");
    System.out.println("Tabla creada.");

      // Insertar algunas filas de prueba
    for (int i = 0; i < 10; i++) {
      stmt.execute("insertar en valores de cujosql.test_table (" + i + ")");
    }
    System.out.println("Filas insertadas");
```

```
    ResultSet rs = stmt.executeQuery("seleccionar col1 de cujosql.test_table");
    System.out.println("Consulta ejecutada");

      // Crear un nuevo conjunto de filas (rowset) y llenarlo...
    DB2CachedRowSet crs = new DB2CachedRowSet();
    crs.populate(rs);
    System.out.println("RowSet lleno.");

    conn.close();
    System.out.println("RowSet se desconecta...");

    System.out.println("Utilizar next()");
    while (crs.next()) {
      System.out.println("v1 is " + crs.getShort(1));
    }

    System.out.println("Utilizar previous()");
    while (crs.previous()) {
    System.out.println("el valor es " + crs.getShort(1));
    }

    System.out.println("Utilizar relative()");
    crs.next();
    crs.relative(9);
    System.out.println("el valor es " + crs.getShort(1));

    crs.relative(-9);
    System.out.println("el valor es " + crs.getShort(1));

    System.out.println("Utilizar absolute()");
    crs.absolute(10);
    System.out.println("el valor es " + crs.getShort(1));
    crs.absolute(1);
    System.out.println("el valor es " + crs.getShort(1));
    crs.absolute(-10);
    System.out.println("el valor es " + crs.getShort(1));
    crs.absolute(-1);
    System.out.println("el valor es " + crs.getShort(1));

    System.out.println("Probar beforeFirst()");
    crs.beforeFirst();
    System.out.println("isBeforeFirst es " + crs.isBeforeFirst());
    crs.next();
    System.out.println("mover uno... isFirst es " + crs.isFirst());

    System.out.println("Probar afterLast()");
    crs.afterLast();
    System.out.println("isAfterLast es " + crs.isAfterLast());
    crs.previous();
    System.out.println("mover uno... isLast es " + crs.isLast());

    System.out.println("Probar getRow()");
    crs.absolute(7);
    System.out.println("la fila debería ser (7) y es " + crs.getRow() +
                       "el valor debería ser (6) y es " + crs.getShort(1));

    crs.close();
  }
  catch (SQLException ex) {
    System.out.println("SQLException: " + ex.getMessage());
  }
 }
}
```

Cambiar datos de DB2CachedRowSet y reflejar de nuevo los cambios en el origen de datos: DB2CachedRowSet utiliza los mismos métodos que la interfaz ResultSet estándar para efectuar cambios en los datos del objeto RowSet. No existe ninguna diferencia a nivel de aplicación entre "Suprimir, insertar y actualizar filas en un DB2CachedRowSet" y cambiar los datos de un ResultSet. DB2CachedRowSet proporciona el método acceptChanges, que se utiliza para "Reflejar de nuevo los cambios de un DB2CachedRowSet en la base de datos subyacente" en la página 118 de donde proceden los datos.

Suprimir, insertar y actualizar filas en un DB2CachedRowSet: Los DB2CachedRowSets pueden actualizarse. En el ejemplo siguiente, el programa crea una tabla y la llena con datos mediante JDBC. Una vez que la tabla está preparada, se crea un DB2CachedRowSet y se llena con la información de la tabla. En el ejemplo también se utilizan diversos métodos que pueden utilizarse para actualizar el RowSet y muestra cómo utilizar la propiedad showDeleted, que permite a la aplicación extraer filas incluso después de que se hayan suprimido. Además, en el ejemplo se utilizan los métodos cancelRowInsert y cancelRowDelete para permitir deshacer la inserción o la supresión de filas.

Ejemplo: suprimir, insertar y actualizar filas en un DB2CachedRowSet

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
import java.sql.*;
import javax.sql.*;
import com.ibm.db2.jdbc.app.DB2CachedRowSet;

public class RowSetSample2
{
   public static void main(String args[])
  {
   // Se registra el controlador.
            try {
     Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
    }
    catch (ClassNotFoundException ex) {
      System.out.println("ClassNotFoundException: " +
              ex.getMessage());

      // No es necesario continuar.
            System.exit(1);
    }

            try {
      Connection conn = DriverManager.getConnection("jdbc:db2:*local");

      Statement stmt = conn.createStatement();

      // Borrar ejecuciones anteriores
            try {
      stmt.execute("eliminar tabla cujosql.test_table");
      }

    catch (SQLException ex) {
      System.out.println("Capturado eliminar tabla: " + ex.getMessage());
      }

      // Crear tabla de prueba
      stmt.execute("Crear tabla cujosql.test_table (col1 smallint)");
      System.out.println("Tabla creada.");

      // Insertar algunas filas de prueba
      for (int i = 0; i < 10; i++) {
        stmt.execute("insertar en valores de cujosql.test_table (" + i + ")");
      }
      System.out.println("Filas insertadas");
```

```
  ResultSet rs = stmt.executeQuery("seleccionar col1 de cujosql.test_table");
  System.out.println("Consulta ejecutada");

  // Crear un nuevo conjunto de filas (rowset) y llenarlo...
  DB2CachedRowSet crs = new DB2CachedRowSet();
  crs.populate(rs);
  System.out.println("RowSet lleno.");

conn.close();
  System.out.println("RowSet se desconecta...");

  System.out.println("Suprimir las tres primeras filas");
  crs.next();
  crs.deleteRow();
  crs.next();
  crs.deleteRow();
  crs.next();
  crs.deleteRow();

  crs.beforeFirst();
  System.out.println("Insertar el valor -10 en el RowSet");
  crs.moveToInsertRow();
  crs.updateShort(1, (short)-10);
  crs.insertRow();
  crs.moveToCurrentRow();

  System.out.println("Actualizar las filas para que sean el contrario de lo que son ahora");
  crs.beforeFirst();
  while (crs.next())
    short value = crs.getShort(1);
    value = (short)-value;
    crs.updateShort(1, value);
    crs.updateRow();
  }

  crs.setShowDeleted(true);

  System.out.println("RowSet es ahora (value - inserted - updated - deleted)");
  crs.beforeFirst();
while (crs.next()) {
    System.out.println("el valor es " + crs.getShort(1) + " " +
                crs.rowInserted() + " " +
                crs.rowUpdated() + " " +
                crs.rowDeleted());
  }

  System.out.println("getShowDeleted es " + crs.getShowDeleted());

  System.out.println("Ahora, deshacer las inserciones y supresiones");
  crs.beforeFirst();
  crs.next();
  crs.cancelRowDelete();
  crs.next();
  crs.cancelRowDelete();
  crs.next();
  crs.cancelRowDelete();
  while (!crs.isLast()) {
  crs.next();
  }

  crs.cancelRowInsert();

  crs.setShowDeleted(false);

  System.out.println("RowSet es ahora (value - inserted - updated - deleted)");
  crs.beforeFirst();
```

```
    while (crs.next()) {
        System.out.println("el valor es " + crs.getShort(1) + " " +
                    crs.rowInserted() + " " +
                    crs.rowUpdated() + " " +
                    crs.rowDeleted());
      }

      System.out.println("finalmente, mostrar que el cancelRowUpdates llamante funciona");
      crs.first();
      crs.updateShort(1, (short) 1000);
      crs.cancelRowUpdates();
        crs.updateRow();
      System.out.println("el valor de la fila es " + crs.getShort(1));
      System.out.println("getShowDeleted es " + crs.getShowDeleted());

      crs.close();

    }

    catch (SQLException ex) {
      System.out.println("SQLException: " + ex.getMessage());
    }
  }
}
```

Reflejar de nuevo los cambios de un DB2CachedRowSet en la base de datos subyacente: Una vez efectuados los cambios en un DB2CachedRowSet, éstos sólo existen mientras exista el objeto RowSet. Es decir, realizar cambios en RowSet desconectado no tiene ningún efecto sobre la base de datos. Para reflejar los cambios de un RowSet en la base de datos subyacente, se utiliza el método acceptChanges. Este método indica al RowSet desconectado que debe volver a establecer una conexión con la base de datos e intentar efectuar en la base de datos subyacente los cambios que se han realizado en el RowSet. Si los cambios no pueden realizarse de forma segura en la base de datos debido a conflictos con otros cambios de base de datos después de crear el RowSet, se lanza una excepción y la transacción se retrotrae.

Ejemplo: reflejar de nuevo los cambios de un DB2CachedRowSet en la base de datos subyacente

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
import java.sql.*;
import javax.sql.*;
import com.ibm.db2.jdbc.app.DB2CachedRowSet;

public class RowSetSample3
{
   public static void main(String args[])
  {
   // Se registra el controlador.
            try {
      Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
    }
    catch (ClassNotFoundException ex) {
      System.out.println("ClassNotFoundException: " +
                  ex.getMessage());
      // No es necesario continuar.
            System.exit(1);
    }

            try {
      Connection conn = DriverManager.getConnection("jdbc:db2:*local");

      Statement stmt = conn.createStatement();

      // Borrar ejecuciones anteriores
            try {
```

```
  stmt.execute("eliminar tabla cujosql.test_table");
  }
 catch (SQLException ex) {
  System.out.println("Capturado eliminar tabla: " + ex.getMessage());
  }

  // Crear tabla de prueba
  stmt.execute("Crear tabla cujosql.test_table (col1 smallint)");
  System.out.println("Tabla creada.");

  // Insertar algunas filas de prueba
  for (int i = 0; i < 10; i++) {
    stmt.execute("insertar en valores de cujosql.test_table (" + i + ")");
  }
  System.out.println("Filas insertadas");

  ResultSet rs = stmt.executeQuery("seleccionar col1 de cujosql.test_table");
  System.out.println("Consulta ejecutada");

  // Crear un nuevo conjunto de filas (rowset) y llenarlo...
  DB2CachedRowSet crs = new DB2CachedRowSet();
  crs.populate(rs);
  System.out.println("RowSet lleno.");

 conn.close();
  System.out.println("RowSet se desconecta...");

  System.out.println("Suprimir las tres primeras filas");
  crs.next();
  crs.deleteRow();
  crs.next();
  crs.deleteRow();
  crs.next();
  crs.deleteRow();

  crs.beforeFirst();
  System.out.println("Insertar el valor -10 en el RowSet");
  crs.moveToInsertRow();
  crs.updateShort(1, (short)-10);
  crs.insertRow();
  crs.moveToCurrentRow();

  System.out.println("Actualizar las filas para que sean el contrario de lo que son ahora");
  crs.beforeFirst();
 while (crs.next()) {
    short value = crs.getShort(1);
    value = (short)-value;
    crs.updateShort(1, value);
    crs.updateRow();
  }

  System.out.println("Ahora, aceptar los cambios de la base de datos");

  crs.setUrl("jdbc:db2:*local");
  crs.setTableName("cujosql.test_table");

  crs.acceptChanges();
  crs.close();

  System.out.println("La tabla de base de datos tendrá este aspecto:");
  conn = DriverManager.getConnection("jdbc:db2:localhost");
  stmt = conn.createStatement();
  rs = stmt.executeQuery("seleccionar col1 de cujosql.test_table");
while (rs.next()) {
    System.out.println("El valor de la tabla es " + rs.getShort(1));
  }
```

```
    conn.close();

    }
    catch (SQLException ex) {
      System.out.println("SQLException: " + ex.getMessage());
    }
  }
}
```

Otras características de DB2CachedRowSet: Además de funcionar como un ResultSet como se ha mostrado en varios ejemplos, la clase DB2CachedRowSet tiene algunas funciones adicionales que hacen más flexible su utilización. Se suministran métodos para convertir todo el RowSet de Java Database Connectivity (JDBC) o sólo una parte de él en una colección Java. Más aún, debido a su naturaleza desconectada, DB2CachedRowSets no tiene una relación estricta de uno a uno con ResultSets.

Con los métodos suministrados por DB2CachedRowSet, puede realizar las siguientes tareas:

- "Obtener colecciones a partir de DB2CachedRowSets"
- "Crear copias de RowSets" en la página 122
- "Crear compartimientos para RowSets" en la página 123

Obtener colecciones a partir de DB2CachedRowSets: Existen tres métodos que devuelven alguna forma de colección desde un objeto DB2CachedRowSet. Son los siguientes:

- **toCollection** devuelve una ArrayList (es decir, una entrada por cada fila) de vectores (es decir, una entrada por cada columna).
- **toCollection(int columnIndex)** devuelve un vector que contiene el valor para cada fila a partir de la columna dada.
- **getColumn(int columnIndex)** devuelve una matriz que contiene el valor para cada columna para una columna determinada.

La diferencia principal entre toCollection(int columnIndex) y getColumn(int columnIndex) consiste en que el método getColumn puede devolver una matriz de tipos primitivos. Por tanto, si columnIndex representa una columna que tiene datos enteros, se devuelve una matriz de enteros y no una matriz que contiene objetos java.lang.Integer.

El ejemplo siguiente muestra cómo puede utilizar estos métodos.

Ejemplo: obtener colecciones a partir de DB2CachedRowSets

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
import java.sql.*;
import javax.sql.*;
import com.ibm.db2.jdbc.app.DB2CachedRowSet;
import java.util.*;

public class RowSetSample4
{
   public static void main(String args[])
  {
   // Se registra el controlador.
            try {
      Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
    }
    catch (ClassNotFoundException ex) {
      System.out.println("ClassNotFoundException: " +
                 ex.getMessage());
      // No es necesario continuar.
            System.exit(1);
    }
```

```
        try {
  Connection conn = DriverManager.getConnection("jdbc:db2:*local");
  Statement stmt = conn.createStatement();

  // Borrar ejecuciones anteriores
        try {
  stmt.execute("eliminar tabla cujosql.test_table");
  }
catch (SQLException ex) {
  System.out.println("Capturado eliminar tabla: " + ex.getMessage());
  }

  // Crear tabla de prueba
  stmt.execute("Crear tabla cujosql.test_table (col1 smallint, col2 smallint)");
  System.out.println("Tabla creada.");

  // Insertar algunas filas de prueba
  for (int i = 0; i < 10; i++) {
    stmt.execute("insertar en valores de cujosql.test_table (" + i + ", " + (i + 100) +  ")");
  }
  System.out.println("Filas insertadas");

  ResultSet rs = stmt.executeQuery("select * from cujosql.test_table");
  System.out.println("Consulta ejecutada");

  // Crear un nuevo conjunto de filas (rowset) y llenarlo...
  DB2CachedRowSet crs = new DB2CachedRowSet();
  crs.populate(rs);
  System.out.println("RowSet lleno.");

conn.close();
  System.out.println("RowSet se desconecta...");

  System.out.println("Probar el método toCollection()");
  Collection collection = crs.toCollection();
  ArrayList map = (ArrayList) collection;

  System.out.println("el tamaño es " + map.size());
  Iterator iter = map.iterator();
  int row = 1;
  while (iter.hasNext()) {
    System.out.print("fila [" + (row++) + "]: \t");

    Vector vector = (Vector)iter.next();
    Iterator innerIter = vector.iterator();
        int i = 1;
    while (innerIter.hasNext()) {
      System.out.print(" [" + (i++) + "]=" + innerIter.next() + "; \t");
    }
    System.out.println();
  }
  System.out.println("Probar el método toCollection(int)");
  collection = crs.toCollection(2);
  Vector vector = (Vector) collection;

  iter = vector.iterator();

  while (iter.hasNext()) {
    System.out.println("Iter: el valor es " + iter.next());
  }

  System.out.println("Probar el método getColumn(int)");
  Object values = crs.getColumn(2);
  short[] shorts = (short [])values;

  for (int i =0; i < shorts.length; i++) {
```

```
        System.out.println("Array: el valor es " + shorts[i]);
      }
    }
    catch (SQLException ex) {
      System.out.println("SQLException: " + ex.getMessage());
    }
  }
}
```

Crear copias de RowSets: El método createCopy crea una copia de DB2CachedRowSet. Se copian todos los datos asociados con el RowSet, junto con todas las estructuras de control, propiedades e identificadores de estado.

El ejemplo siguiente muestra cómo puede utilizar este método.

Ejemplo: crear copias de RowSets

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
import java.sql.*;
import javax.sql.*;
import com.ibm.db2.jdbc.app.*;
import java.io.*;

public class RowSetSample5
{
   public static void main(String args[])
  {
   // Se registra el controlador.
            try {
      Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
    }
    catch (ClassNotFoundException ex) {
      System.out.println("ClassNotFoundException: " +
                 ex.getMessage());
      // No es necesario continuar.
            System.exit(1);
    }

            try {
      Connection conn = DriverManager.getConnection("jdbc:db2:*local");

      Statement stmt = conn.createStatement();

      // Borrar ejecuciones anteriores
            try {
      stmt.execute("eliminar tabla cujosql.test_table");
      }
    catch (SQLException ex) {
      System.out.println("Capturado eliminar tabla: " + ex.getMessage());
      }

      // Crear tabla de prueba
      stmt.execute("Crear tabla cujosql.test_table (col1 smallint)");
      System.out.println("Tabla creada.");

      // Insertar algunas filas de prueba
      for (int i = 0; i < 10; i++) {
        stmt.execute("insertar en valores de cujosql.test_table (" + i + ")");
      }
      System.out.println("Filas insertadas");

      ResultSet rs = stmt.executeQuery("seleccionar col1 de cujosql.test_table");
      System.out.println("Consulta ejecutada");
```

```
      // Crear un nuevo conjunto de filas (rowset) y llenarlo...
      DB2CachedRowSet crs = new DB2CachedRowSet();
      crs.populate(rs);
      System.out.println("RowSet lleno.");

    conn.close();
      System.out.println("RowSet se desconecta...");

      System.out.println("Ahora, algunos RowSets nuevos a partir de uno.");
      DB2CachedRowSet crs2 = crs.createCopy();
      DB2CachedRowSet crs3 = crs.createCopy();

      System.out.println("Cambiar el segundo a valores negados");
      crs2.beforeFirst();
      while (crs2.next()) {
        short value = crs2.getShort(1);
        value = (short)-value;
        crs2.updateShort(1, value);
        crs2.updateRow();
      }

      crs.beforeFirst();
      crs2.beforeFirst();
      crs3.beforeFirst();
      System.out.println("Ahora, observar los tres de nuevo");

    while (crs.next()) {
        crs2.next();
        crs3.next();
        System.out.println("Valores: crs: " + crs.getShort(1) + ", crs2: " + crs2.getShort(1) +
                   ", crs3: " + crs3.getShort(1));
      }
    }
    catch (Exception ex) {
      System.out.println("SQLException: " + ex.getMessage());
                 ex.printStackTrace();
    }

  }

}
```

Crear compartimientos para RowSets: El método createShared crea un nuevo objeto RowSet con información de estado de alto nivel y permite que dos objetos RowSet compartan los mismos datos físicos subyacentes.

El ejemplo siguiente muestra cómo puede utilizar este método.

Ejemplo: crear compartimientos de RowSets

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
import java.sql.*;
import javax.sql.*;
import com.ibm.db2.jdbc.app.*;
import java.io.*;

public class RowSetSample5
{
   public static void main(String args[])
  {
   // Se registra el controlador.
           try {
      Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
    }
```

```
catch (ClassNotFoundException ex) {
  System.out.println("ClassNotFoundException: " +
             ex.getMessage());
  // No es necesario continuar.
        System.exit(1);
}

        try {
  Connection conn = DriverManager.getConnection("jdbc:db2:*local");

  Statement stmt = conn.createStatement();

  // Borrar ejecuciones anteriores
        try {
  stmt.execute("eliminar tabla cujosql.test_table");
  }
catch (SQLException ex) {
  System.out.println("Capturado eliminar tabla: " + ex.getMessage());
  }

  // Crear tabla de prueba
  stmt.execute("Crear tabla cujosql.test_table (col1 smallint)");
  System.out.println("Tabla creada.");

  // Insertar algunas filas de prueba
  for (int i = 0; i < 10; i++) {
    stmt.execute("insertar en valores de cujosql.test_table (" + i + ")");
  }
  System.out.println("Filas insertadas");

  ResultSet rs = stmt.executeQuery("seleccionar col1 de cujosql.test_table");
  System.out.println("Consulta ejecutada");

  // Crear un nuevo conjunto de filas (rowset) y llenarlo...
  DB2CachedRowSet crs = new DB2CachedRowSet();
  crs.populate(rs);
  System.out.println("RowSet lleno.");

conn.close();
  System.out.println("RowSet se desconecta...");

  System.out.println("Probar la función createShared (crear 2 compartimientos)");
  DB2CachedRowSet crs2 = crs.createShared();
  DB2CachedRowSet crs3 = crs.createShared();

  System.out.println("Utilizar el original para actualizar el valor 5 de la tabla");
  crs.absolute(5);
  crs.updateShort(1, (short)-5);
  crs.updateRow();

  crs.beforeFirst();
  crs2.afterLast();

  System.out.println("Ahora, mover los cursores en direcciones opuestas de los mismos datos.");

while (crs.next()) {
    crs2.previous();
    crs3.next();
    System.out.println("Valores: crs: " + crs.getShort(1) + ", crs2: " + crs2.getShort(1) +
                ", crs3: " + crs3.getShort(1));
  }
  crs.close();
  crs2.close();
  crs3.close();
}
catch (Exception ex) {
  System.out.println("SQLException: " + ex.getMessage());
```

```
                ex.printStackTrace();
    }
  }
}
```

DB2JdbcRowSet: DB2JdbcRowSet es un RowSet conectado, lo que significa que sólo puede utilizarse con el soporte de un objeto Connection subyacente, un objeto PreparedStatement o un objeto ResultSet. Su implementación es muy parecida a la descripción de un JdbcRowSet.

Utilizar DB2JdbcRowSet: Debido a que el objeto DB2JdbcRowSet soporta los eventos descritos en la especificación Java Database Connectivity (JDBC) 3.0 para todos los RowSets, puede actuar como un objeto intermediario entre una base de datos local y otros objetos a los que debe informarse de los cambios efectuados en los datos de la base de datos.

Como ejemplo, suponga que está trabajando en un entorno en el que tiene una base de datos principal y varios asistentes digitales personales (PDA) que utilizan un protocolo inalámbrico para conectarse a ella. Puede utilizarse un objeto DB2JdbcRowSet para mover a una fila y actualizarla utilizando una aplicación maestra que se ejecuta en el servidor. La actualización de fila provoca la generación de un evento por parte del componente RowSet. Si existe un servicio en ejecución que es responsable de enviar actualizaciones a los PDA, puede registrarse a sí mismo como ″escuchador″ del RowSet. Cada vez que recibe un evento RowSet, puede generar la actualización adecuada y enviarla a los dispositivos inalámbricos.

Consulte el Ejemplo: eventos DB2JdbcRowSet para obtener más información.

Crear JDBCRowSets: Existen varios métodos suministrados para crear un objeto DB2JDBCRowSet. Cada uno de ellos está diseñado de la siguiente forma.

Utilizar propiedades de DB2JdbcRowSet y DataSources

Los DB2JdbcRowSets tienen propiedades que aceptan una consulta SQL y un nombre DataSource. Con ello, los DB2JdbcRowSets quedan preparados para utilizarse. A continuación se ofrece un ejemplo de este método. Se supone que la referencia al DataSource denominado BaseDataSource es un DataSource válido que se ha configurado anteriormente.

Ejemplo: utilizar propiedades de DB2JdbcRowSet DataSources

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
    // Crear un nuevo DB2JdbcRowSet
    DB2JdbcRowSet jrs = new DB2JdbcRowSet();

    // Establecer las propiedades necesarias para
    // procesar el RowSet.
    jrs.setDataSourceName("BaseDataSource");
    jrs.setCommand("seleccionar col1 de cujosql.test_table");

    // Llamar al método execute de RowSet. Este método provoca
    // que el RowSet utilice el DataSource y la consulta SQL
    // especificada para prepararse para el proceso de datos.
    jrs.execute();

    // Bucle de los datos del RowSet.
    while (jrs.next()) {
        System.out.println("v1 is " + jrs.getString(1));
    }

   // Finalmente, cerrar el RowSet.
    jrs.close();
```

Utilizar propiedades de DB2JdbcRowSet y URL de JDBC

Los DB2JdbcRowSets tienen propiedades que aceptan una consulta SQL y un URL de JDBC. Con ello, los DB2JdbcRowSets quedan preparados para utilizarse. A continuación se ofrece un ejemplo de este método:

Ejemplo: Utilizar propiedades DB2JdbcRowSet y los URL de JDBC

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
// Crear un nuevo DB2JdbcRowSet
DB2JdbcRowSet jrs = new DB2JdbcRowSet();

// Establecer las propiedades necesarias para
// procesar el RowSet.
jrs.setUrl("jdbc:db2:*local");
jrs.setCommand("seleccionar col1 de cujosql.test_table");

// Llamar al método execute de RowSet. Esto provoca que
// el RowSet utilice el URL y la consulta SQL especificada
// anteriormente para prepararse para el proceso de datos.
jrs.execute();

// Bucle de los datos del RowSet.
while (jrs.next()) {
    System.out.println("v1 is " + jrs.getString(1));
}

// Finalmente, cerrar el RowSet.
jrs.close();
```

Utilizar el método setConnection(Connection) para utilizar una conexión de base de datos existente

Para promocionar la reutilización de objetos JDBC Connection, DB2JdbcRowSet permite pasar un objeto connection establecido al DB2JdbcRowSet. DB2JdbcRowSet utiliza esta conexión para prepararse para la utilización cuando se llama al método execute.

Ejemplo: utilizar el método setConnection

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
// Establecer una conexión JDBC con la base de datos.
Connection conn = DriverManager.getConnection("jdbc:db2:*local");

// Crear un nuevo DB2JdbcRowSet.
DB2JdbcRowSet jrs = new DB2JdbcRowSet();

// Establecer las propiedades necesarias para
// el RowSet utilice una conexión establecida.
jrs.setConnection(conn);
jrs.setCommand("seleccionar col1 de cujosql.test_table");

// Llamar al método execute de RowSet. Esto provoca que
// el RowSet utilice la conexión que se le suministró
// anteriormente para prepararse para el proceso de datos.
jrs.execute();

// Bucle de los datos del RowSet.
while (jrs.next()) {
    System.out.println("v1 is " + jrs.getString(1));
```

```
    }

   // Finalmente, cerrar el RowSet.
    jrs.close();
```

Acceso a datos y movimiento de cursores: La manipulación de la posición del cursor y el acceso a los datos de la base de datos a través de un DB2JdbcRowSet corre a cargo del objeto ResultSet subyacente. Las tareas que pueden realizarse con un objeto ResultSet también se aplican al objeto DB2JdbcRowSet.

Cambiar datos y reflejarlos en la base de datos subyacente: El objeto ResultSet subyacente maneja completamente el soporte para actualizar la base de datos a través de un DB2JdbcRowSet. Las tareas que pueden realizarse con un objeto ResultSet también se aplican al objeto DB2JdbcRowSet.

Eventos DB2JdbcRowSet: Todas las implementaciones de RowSet soportan el manejo de eventos para situaciones que son de interés para otros componentes. Este soporte permite a los componentes de aplicación "conversar" entre sí cuando se producen eventos en ellos. Por ejemplo, la actualización de una fila de base de datos mediante un RowSet puede provocar que se muestre al usuario una tabla de interfaz gráfica de usuario (GUI) para que se actualice automáticamente.

En el ejemplo siguiente, el método main efectúa la actualización en RowSet y es la aplicación central. El escuchador forma parte del servidor inalámbrico utilizado por los clientes desconectados en el campo. Es posible enlazar estos dos aspectos de gestión sin necesidad de mezclar el código de los dos procesos. Aunque el soporte de eventos de RowSets se ha diseñado principalmente para actualizar GUI con datos de base de datos, funciona perfectamente para este tipo de problema de aplicación.

Ejemplo: eventos DB2JdbcRowSet

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
import java.sql.*;
import javax.sql.*;
import com.ibm.db2.jdbc.app.DB2JdbcRowSet;

public class RowSetEvents {
   public static void main(String args[])
    {
   // Se registra el controlador.
            try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
        } catch (ClassNotFoundException ex) {
            System.out.println("ClassNotFoundException: " +
                               ex.getMessage());
    // No es necesario continuar.
            System.exit(1);
        }


            try {
            // Obtener la conexión JDBC y la sentencia necesarias para configurar
            // este ejemplo.
            Connection conn = DriverManager.getConnection("jdbc:db2:*local");
            Statement stmt = conn.createStatement();

            // Borrar ejecuciones anteriores.
           try {
           stmt.execute("eliminar tabla cujosql.test_table");
        } catch (SQLException ex) {
                System.out.println("Capturado eliminar tabla: " + ex.getMessage());
            }

            // Crear tabla de prueba
            stmt.execute("Crear tabla cujosql.test_table (col1 smallint)");
```

```
            System.out.println("Tabla creada.");

            // Llenar la tabla con datos.
            for (int i = 0; i < 10; i++) {
                stmt.execute("insertar en valores de cujosql.test_table (" + i + ")");
            }
            System.out.println("Filas insertadas");

            // Eliminar los objetos de configuración.
            stmt.close();
            conn.close();

            // Crear un nuevo rowset y establecer las propiedades necesarias para
            // procesarlo.
            DB2JdbcRowSet jrs = new DB2JdbcRowSet();
            jrs.setUrl("jdbc:db2:*local");
            jrs.setCommand("seleccionar col1 de cujosql.test_table");
            jrs.setConcurrency(ResultSet.CONCUR_UPDATEABLE);

            // Dar al objeto RowSet un escuchador. Este objeto maneja
            // el proceso especial cuando se realizan determinadas acciones
            // en el RowSet.
            jrs.addRowSetListener(new MyListener());

            // Procesar el RowSet para proporcionar acceso a los datos de la base
            // de datos.
            jrs.execute();

            // Provocar algunos eventos de cambio de cursor. Estos eventos provocan
            // que el método cursorMoved del objeto escuchador obtenga en control.
            jrs.next();
            jrs.next();
            jrs.next();

            // Provocar un evento de cambio de fila. Este evento provoca que el
            // método rowChanged del objeto escuchador obtenga en control.
            jrs.updateShort(1, (short)6);
            jrs.updateRow();

            // Finalmente, provocar un evento de cambio de RowSet. Este provoca que
            // el método rowSetChanged del objeto escuchador obtenga el control.
            jrs.execute();

            // Al terminar, cerrar el RowSet.
            jrs.close();
        } catch (SQLException ex) {
            ex.printStackTrace();
        }
    }
}


/**
 * Este es un ejemplo de escuchador. Imprime mensajes que muestran
 * cómo se mueve el flujo de control a través de la aplicación y ofrece algunas
 * sugerencias acerca de lo que puede hacerse si la aplicación se ha implementado plenamente.
 */
class MyListener
implements RowSetListener {
    public void cursorMoved(RowSetEvent rse) {
        System.out.println("Evento a realizar:  Posición de cursor cambiada.");
        System.out.println("  Para el sistema remoto, no hacer nada ");
        System.out.println("  cuando se produzca este evento. La vista remota de los datos");
        System.out.println("  puede controlarse independientemente de la vista local.");
            try {
            DB2JdbcRowSet rs = (DB2JdbcRowSet) rse.getSource();
            System.out.println("la fila es " + rs.getRow() + ". \n\n");
```

```
            } catch (SQLException e) {
            System.out.println("Hacer:  Manejar adecuadamente los posibles problemas.");
        }
    }

    public void rowChanged(RowSetEvent rse) {
        System.out.println("Evento a realizar:  Fila cambiada.");
        System.out.println("  Indicar al sistema remoto que una fila ha cambiado. A continuación,");
        System.out.println("  pasar todos los valores sólo de esa fila al ");
        System.out.println("  sistema remoto.");
            try {
            DB2JdbcRowSet rs = (DB2JdbcRowSet) rse.getSource();
            System.out.println("los nuevos valores son " + rs.getShort(1) + ". \n\n");
            } catch (SQLException e) {
            System.out.println("Hacer:  Manejar adecuadamente los posibles problemas.");
        }
    }

    public void rowSetChanged(RowSetEvent rse) {
        System.out.println("Evento a realizar:  RowSet cambiado.");
        System.out.println("  Si existe un RowSet remoto ya establecido, ");
        System.out.println("  indicar al sistema remoto que los valores que ");
        System.out.println("  tiene deben descartarse. A continuación, pasarle todos ");
        System.out.println("  los valores actuales.\n\n");
    }
}
```

Consejos sobre el rendimiento del controlador JDBC de IBM Developer Kit para Java

El controlador JDBC de IBM Developer Kit para Java está diseñado como interfaz Java de alto rendimiento para trabajar con la base de datos. Sin embargo, para obtener el mejor rendimiento posible, es necesario crear las aplicaciones de manera que aprovechen el potencial que puede ofrecer el controlador JDBC. Los siguientes consejos pretenden llevar a la práctica las mejores técnicas de programación JDBC. La mayoría de ellos no son específicos del controlador JDBC nativo. Por tanto, las aplicaciones que se escriban de acuerdo con estas directrices también tendrán un buen rendimiento si se emplean con controladores JDBC distintos del nativo.

- Evitar consultas SQL SELECT * (page 129)
- Utilizar getXXX(int) en vez de getXXX(String) (page 130)
- Evitar llamadas a getObject para tipos Java primitivos (page 130)
- Utilizar PreparedStatement más que Statement (page 130)
- Evitar llamadas costosas a DatabaseMetaData (page 130)
- Utilizar el nivel de compromiso correcto para la aplicación (page 130)
- Considerar la posibilidad de almacenar datos en Unicode (page 131)
- Utilizar procedimientos almacenados (page 131)
- Utilizar BigInt en vez de Numérico/Decimal (page 131)
- Cerrar explícitamente los recursos JDBC cuando ya no se necesitan (page 131)
- Utilizar agrupación de conexiones (page 131)
- Considerar la posibilidad de utilizar la agrupación de sentencias PreparedStatement (page 132)
- Utilizar SQL eficaz (page 132)

Evitar consultas SQL SELECT *

SELECT * FROM... es una forma muy habitual de establecer una consulta en SQL. Sin embargo, muchas veces no es necesario consultar todos los campos. Para cada columna que hay que devolver, el controlador JDBC tiene que hacer el trabajo adicional de enlazar y devolver la fila. Aún en el caso de que la aplicación no llegue a utilizar nunca una columna concreta, el controlador JDBC debe tenerla presente y reservar espacio por si se utiliza. Esta cuestión no supone una actividad general significativa si son

pocas las columnas que no se utilizan de las tablas. Sin embargo, si son numerosas las columnas no utilizadas, la actividad general puede llegar a ser significativa. En tal caso, sería mejor listar individualmente las columnas que a la aplicación le interesa consultar, como en este ejemplo:

```
SELECT COL1, COL2, COL3 FROM...
```

Utilizar getXXX(int) en vez de getXXX(String)

Utilice los métodos getXXX de ResultSet que toman valores numéricos, en vez de las versiones que toman nombres de columna. Si bien la libertad que supone utilizar nombres de columna en vez de constantes numéricas parece ser una ventaja, la base de datos propiamente dicha solo sabe manejar los índices de las columnas. Por ello, cada vez que se llama a un método getXXX utilizando el nombre de una columna, el controlador JDBC lo debe resolver para que el método se pueda pasar a la base de datos. Normalmente, a los métodos getXXX se les suele llamar dentro de bucles que se pueden ejecutar millones de veces, y ello provocaría rápidamente la acumulación de la pequeña actividad general que supone la resolución de cada uno de los nombres de columna.

Evitar llamadas a getObject para tipos Java primitivos

Cuando de la base de datos se obtienen valores de tipos primitivos (int, long, float, etc.), es más rápido utilizar el método get específico del tipo primitivo (getInt, getLong, getFloat) que utilizar el método getObject. La llamada a getObject realiza el trabajo de obtener el tipo primitivo y luego crea un objeto para devolverlo. Normalmente, esto se hace en los bucles, lo que supone crear millones de objetos de corta vida. Si se emplea getObject para los mandatos de primitivos, existe el inconveniente adicional de que se activa con frecuencia el colector de basura, lo que aún reduce más el rendimiento.

Utilizar PreparedStatement más que Statement

Si escribe una sentencia SQL que se va a utilizar más de una vez, el rendimiento será mayor si la sentencia es un objeto PreparedStatement que si es un objeto Statement. Cada vez que se ejecuta una sentencia, se realiza un proceso de dos pasos: la sentencia se prepara y luego se procesa. Si se emplea una sentencia preparada, el paso de preparar la sentencia solo tiene lugar en el momento de construir la sentencia, y no se repite cada vez que se ejecuta la sentencia. Los programadores, aunque saben que el rendimiento de PreparedStatement es mayor que el de Statement, suelen desaprovechar esta ventaja en muchas ocasiones. Debido a la mejora de rendimiento que proporcionan las sentencias PreparedStatement, conviene utilizarlas en el diseño de las aplicaciones siempre que sea posible (consulte la sección Agrupación de sentencias PreparedStatement (page 132)).

Evitar llamadas a DatabaseMetaData

Conviene tener en cuenta que algunas de las llamadas a DatabaseMetaData pueden ser costosas. En particular, los métodos getBestRowIdentifier, getCrossReference, getExportedKeys y getImportedKeys pueden resultar costosos. Algunas llamadas a DataBaseMetaData implican complejas condiciones de unión sobre tablas a nivel del sistema. Únicamente se deben emplear cuando se necesita la información que proporcionan, no porque resulte más práctico.

Utilizar el nivel de compromiso correcto para la aplicación

JDBC proporciona varios niveles de compromiso, que determinan cómo se afectan mutuamente varias transacciones con respecto al sistema (consulte la sección Transacciones para obtener más detalles). El valor por omisión es utilizar el nivel de compromiso más bajo. Esto implica que las transacciones pueden ver parte del trabajo de cada una de ellas a través de los límites del compromiso. También implica la posibilidad de que se produzcan ciertas anomalías de base de datos. Algunos programadores aumentan el nivel de compromiso para no tener que preocuparse de si se produce este tipo de anomalías. Tenga en cuenta que los niveles de compromiso más altos implican que la base de datos se cuelgue en bloqueos más bastos. Esto limita la cantidad de concurrencia que el sistema puede tener, disminuyendo en gran medida el rendimiento de algunas aplicaciones. A menudo, las condiciones de anomalía no se pueden

producir debido en primer lugar al diseño de la aplicación. Tómese su tiempo para comprender lo que está tratando de lograr y limite el nivel de aislamiento de las transacciones al mínimo que pueda emplear sin arriesgarse.

Considerar la posibilidad de almacenar datos en Unicode

En Java, todos los datos de tipo carácter con los que se trabaja (objetos String) deben tener el formato Unicode. Por lo tanto, para las tablas cuyos datos no tengan el formato Unicode, se necesitará que el controlador JDBC convierta los datos a ese formato y desde él al ponerlos en la base de datos y al recuperarlos de la base de datos. Si la tabla ya tiene los datos en Unicode, el controlador JDBC no tendrá que convertirlos, por lo que será más rápido colocar en ella los datos de la base de datos. Fíjese, sin embargo, que los datos en Unicode pueden no funcionar con las aplicaciones no Java, ya que estas no saben cómo manejar el formato Unicode. Tenga presente también que el rendimiento no se ve afectado para los datos que no son de tipo carácter, ya que esos datos nunca se tienen que convertir. Aún hay que tener en cuenta otra particularidad, y es que los datos almacenados en Unicode ocupan el doble de espacio que los datos de un solo byte. Sin embargo, si son numerosas las columnas de tipo carácter que se leen muchas veces, puede llegar a ser notable el aumento de rendimiento que supone almacenar los datos en Unicode.

Utilizar procedimientos almacenados

El uso de procedimientos almacenados está soportado en Java. El rendimiento de los procedimientos almacenados puede ser mayor al permitir que el controlador JDBC ejecute SQL estático en vez de SQL dinámico. No cree procedimientos almacenados para cada sentencia SQL individual que ejecute en el programa. No obstante, cuando sea posible, cree un procedimiento almacenado que ejecute un grupo de sentencias SQL.

Utilizar BigInt en lugar de Numérico o Decimal

En vez de utilizar campos numéricos o decimales cuya escala sea 0, utilice el tipo de datos BigInt. BigInt se convierte directamente al tipo Java primitivo Long, mientras que los tipos de datos numéricos o decimales se convierten en objetos String o BigDecimal. Como se ha indicado en la sección Evitar llamadas a getObject para tipos primitivos Java (page 130), es preferible utilizar tipos de datos primitivos a utilizar tipos que requieren la creación de objetos.

Cerrar explícitamente los recursos JDBC cuando ya no se necesitan

La aplicación debe cerrar explícitamente los objetos ResultSet, Statement y Connection cuando ya no se necesitan. Así se hace una limpieza de recursos del modo más eficaz posible, y el rendimiento puede aumentar. Además, los recursos de base de datos que no se cierran de manera explícita pueden provocar fugas de recursos, y los bloqueos de base de datos se pueden prolongar más de lo debido. Ello puede producir anomalías de aplicación o reducir la concurrencia en las aplicaciones.

Utilizar agrupación de conexiones

La agrupación de conexiones es una estrategia que permite reutilizar los objetos Connection de JDBC por parte de múltiples usuarios, en vez de dejar que cada usuario solicite crear su propio objeto Connection. La creación de objetos Connection es costosa. En vez de hacer que cada usuario cree una conexión nueva, conviene que las aplicaciones sensibles al rendimiento compartan una agrupación de conexiones. Muchos productos (como por ejemplo WebSphere) proporcionan soporte para la agrupación de conexiones que puede utilizarse con poco esfuerzo adicional por parte del usuario. Si no desea utilizar un producto que tenga soporte para la agrupación de conexiones o si prefiere construir una propia con objeto de controlar mejor su funcionamiento y su ejecución, piense que es relativamente fácil hacerlo.

Considerar la posibilidad de utilizar la agrupación de sentencias PreparedStatement

La agrupación de sentencias funciona de manera muy parecida a la agrupación de conexiones. En vez de poner en la agrupación tan solo las conexiones, en ella se pone un objeto que contenga la conexión y las sentencias PreparedStatement. Luego se recupera ese objeto y se accede a la sentencia concreta que se desea utilizar. Esta estrategia puede aumentar drásticamente el rendimiento.

Utilizar SQL eficaz

Dado que JDBC se construye encima de SQL, cualquier técnica que mejore la eficacia de SQL mejorará también la eficacia de JDBC. Por tanto, JDBC se beneficia de las consultas optimizadas, de los índices acertadamente elegidos y de otros aspectos que mejoren el diseño de SQL.

Acceder a bases de datos mediante el soporte SQLJ DB2 de IBM Developer Kit para Java

El soporte de lenguaje de consulta estructurada para Java (SQLJ) DB2 se basa en el estándar SQLJ ANSI. El soporte SQLJ DB2 está contenido en IBM Developer Kit para Java. Este soporte le permite crear, construir y ejecutar SQL intercalado para aplicaciones Java.

El soporte SQLJ proporcionado por IBM Developer Kit para Java incluye las clases de ejecución SQLJ y está disponible en el archivo /QIBM/ProdData/Java400/ext/runtime.zip. Para obtener más información acerca de las clases de ejecución SQLJ, consulte la documentación de la API de ejecución suministrada en las Preguntas frecuentes de www.oracle.com/technology/tech/java/sqlj_jdbc/htdocs/faq.html

.

Puesta a punto de SQLJ

Para poder utilizar SQLJ en aplicaciones Java en el servidor, debe preparar el servidor para utilizar SQLJ. Para obtener más información, consulte la página siguiente:

Preparación del servidor para utilizar SQLJ

Herramientas SQLJ

Las herramientas siguientes también se incluyen en el soporte SQLJ suministrado por IBM Developer Kit para Java:

- El conversor de SQLJ, `sqlj`, sustituye las sentencias SQL intercaladas en el programa SQLJ por sentencias de código fuente Java y genera un perfil serializado que contiene información acerca de las operaciones SQLJ que se encuentran en el programa SQLJ.
- El personalizador de perfiles SQLJ DB2, `db2profc`, precompila las sentencias SQL almacenadas en el perfil generado y genera un paquete en la base de datos DB2.
- El impresor de perfiles SQLJ DB2, `db2profp`, imprime el contenido de un perfil personalizado DB2 en texto plano.
- El instalador de auditores de perfiles SQLJ, `profdb`, instala y desinstala auditores de clase de depuración en un conjunto de perfiles binarios existentes.
- La herramienta de conversión de perfiles SQLJ, `profconv`, convierte una instancia de perfil serializado a formato de clase Java.

Nota: estas herramientas deben ejecutarse en el intérprete de Qshell.

Restricciones de SQLJ DB2

Cuando cree aplicaciones DB2 con SQLJ, debe tener en cuenta las siguientes restricciones:

- El soporte SQLJ DB2 se ajusta a las restricciones estándar de DB2 Universal Database con respecto a la emisión de sentencias SQL.
- El personalizador de perfil SQLJ DB2 solo se debe ejecutar en los perfiles asociados a conexiones a la base de datos local.
- Para la implementación de referencia SQLJ se necesita JDK 1.1 o superior. En Soporte para múltiples Java Development Kits (JDK) hallará más información sobre cómo ejecutar múltiples versiones de Java Development Kit.

Para obtener información acerca de la utilización de SQL en las aplicaciones Java, consulte las secciones Intercalar sentencias SQL en la aplicación Java y Compilar y ejecutar programas SQLJ.

Perfiles de lenguaje de consulta estructurada para Java

El conversor de SQLJ, sqlj, genera los perfiles cuando se convierte el archivo fuente SQLJ. Los perfiles son archivos binarios serializados. Esta es la razón por la que estos archivos tienen la extensión .ser. Estos archivos contienen las sentencias SQL del archivo fuente SQLJ asociado.

Para generar perfiles a partir del código fuente SQLJ, ejecute el conversor de SQLJ, sqlj, en el archivo .sqlj.

Para obtener más información, consulte la sección Compilar y ejecutar programas SQLJ.

Conversor de lenguaje de consulta estructurada para Java (SQLJ) (sqlj)

El conversor de SQL para Java, sqlj, genera un perfil serializado que contiene información sobre las operaciones SQL que se encuentran en el programa SQLJ. El conversor SQLJ utiliza el archivo /QIBM/ProdData/Java400/ext/translator.zip.

Para obtener más información acerca de las opciones de la línea de mandatos sqlj, consulte las Preguntas frecuentes de SQLJ proporcionadas en la implementación de www.oracle.com/technology/tech/java/sqlj_jdbc/htdocs/faq.html

.

Precompilar sentencias SQL en un perfil mediante el personalizador de perfiles SQLJ DB2, db2profc

Puede utilizar el personalizador de perfiles SQLJ DB2, db2profc, para que la aplicación Java trabaje de forma más eficaz con la base de datos.

El personalizador de perfiles SQLJ DB2 hace lo siguiente:

- Precompila las sentencias SQL almacenadas en un perfil y genera un paquete en la base de datos DB2.
- Personaliza el perfil SQLJ sustituyendo las sentencias SQL por referencias a la sentencia asociada en el paquete que se ha creado.

Para precompilar las sentencias SQL de un perfil, escriba lo siguiente en el indicador de mandatos de Qshell:

```
db2profc MyClass_SJProfile0.ser
```

Donde *MyClass_SJProfile0.ser* es el nombre del perfil que desea precompilar.

Utilización y sintaxis del personalizador de perfil SQLJ DB2

db2profc[*opciones*] <*nombre_perfil_SQLJ*>

Donde *nombre_perfil_SQLJ* es el nombre del perfil que debe imprimirse, y *opciones* es la lista de opciones que desea.

Las opciones disponibles para db2profp son las siguientes:

- `-URL=<URL_JDBC>`
- `-user=<nombreusuario>`
- `-password=<contraseña>`
- `-package=<nombre_biblioteca/nombre_paquete>`
- `-commitctrl=<control_compromiso>`
- `-datefmt=<formato_fecha>`
- `-datesep=<separador_fecha>`
- `-timefmt=<formato_hora>`
- `-timesep=<separador_hora>`
- `-decimalpt=<separador_decimal>`
- `-stmtCCSID=<CCSID>`
- `-sorttbl=<nombre_biblioteca/nombre_tabla_secuencia_ordenación>`
- `-langID=<identificar_idioma>`

A continuación se describen estas opciones:

-URL=<*URL_JDBC*>
Donde *URL_JDBC* es el URL de la conexión JDBC. La sintaxis del URL es:

```
"jdbc:db2:nombresistema"
```

Para obtener más información, consulte la sección Acceder a la base de datos de iSeries con el controlador JDBC de IBM Developer Kit para Java.

-user=<*nombreusuario*>
Donde *nombreusuario* es el nombre de usuario. El valor por omisión es el ID del usuario actual que ha iniciado la sesión en la conexión local.

-password=<*contraseña*>
Donde *contraseña* es su contraseña. El valor por omisión es la contraseña del usuario actual que ha iniciado la sesión en la conexión local.

-package=<*nombre_biblioteca/nombre_paquete*>
Donde *nombre_biblioteca* es la biblioteca donde se encuentra el paquete y *nombre_paquete* es el nombre del paquete que debe generarse. El nombre de biblioteca por omisión es QUSRSYS. El nombre de paquete por omisión se genera a partir del nombre del perfil. La longitud máxima del nombre de paquete es de 10 caracteres. Debido a que el nombre del perfil SQLJ siempre es superior a los 10 caracteres, el nombre de paquete por omisión que se crea es diferente del nombre de perfil. El nombre de paquete por omisión se crea concatenando las primeras letras del nombre del perfil con el número de clave del perfil. Si el número de clave del perfil es superior a 10 caracteres, se utilizan los 10 últimos caracteres del número de clave del perfil como nombre de paquete por omisión. Por ejemplo, el siguiente diagrama muestra algunos nombres de perfil y sus nombres de paquete por omisión:

Nombre de perfil	Nombre de paquete por omisión
App_SJProfile0	App_SJPro0
App_SJProfile01234	App_S01234
App_SJProfile012345678	A012345678
App_SJProfile01234567891	1234567891

-commitctrl=<*control_compromiso*>

Donde *control_compromiso* es el nivel de control de compromiso que se desea. El control de compromiso puede tener cualquiera de los siguientes valores de tipo carácter:

Valor	Definición
C	*CHG. Son posibles las lecturas sucias, las lecturas no repetibles y las lecturas fantasma.
S	*CS. No son posibles las lecturas sucias, pero sí las lecturas no repetibles y las lecturas fantasma.
A	*ALL. No son posibles las lecturas sucias ni las lecturas no repetibles, pero sí las lecturas fantasma.
N	*NONE. No son posibles las lecturas sucias, las lecturas no repetibles ni las lecturas fantasma. Este es el valor por omisión.

-datefmt=<*formato_fecha*>

Donde *formato_fecha* es el tipo de formato de fecha que se desea. El formato de fecha puede tener cualquiera de los siguientes valores:

Valor	Definición
USA	Estándar IBM de Estados Unidos (mm.dd.aaaa,hh:mm a.m., hh:mm p.m.)
ISO	International Standards Organization (aaaa-mm-dd, hh.mm.ss). Este es el valor por omisión.
EUR	Estándar europeo de IBM (dd.mm.aaaa, hh.mm.ss)
JIS	Japanese Industrial Standard Christian Era (aaaa-mm-dd, hh:mm:ss)
MDY	Mes/Día/Año (mm/d/aa)
DMY	Día/Mes/Año (dd/mm/aa)
YMD	Año/Mes/Día (aa/mm/dd)
JUL	Juliana (aa/ddd)

El formato de fecha se utiliza al acceder a las columnas de resultados de tipo fecha. Todos los campos de fecha de salida se devuelven en el formato especificado. Para las series de fecha de entrada, se utiliza el valor indicado para determinar si la fecha se ha especificado con un formato válido. El valor por omisión es ISO.

-datesep=<*separador_fecha*>

Donde *separador_fecha* es el tipo de separador que desea utilizar. El separador de fecha se utiliza al acceder a las columnas de resultados de tipo fecha. El separador de fecha puede ser cualquiera de los siguientes:

Valor	Definición
/	Se utiliza una barra inclinada.
.	Se utiliza un punto.
,	Se utiliza una coma.
-	Se utiliza un guión. Este es el valor por omisión.
blank	Se utiliza un espacio.

-timefmt=<*formato_hora*>

Donde *formato_hora* es el formato que desea utilizar para visualizar campos de hora. El formato de hora se utiliza al acceder a las columnas de resultados de tipo hora. Para las series de hora de

entrada, se utiliza el valor indicado para determinar si la hora se ha especificado con un formato válido. El formato de hora puede tener cualquiera de los siguientes valores:

Valor	Definición
USA	Estándar IBM de Estados Unidos (mm.dd.aaaa,hh:mm a.m., hh:mm p.m.)
ISO	International Standards Organization (aaaa-mm-dd, hh.mm.ss). Este es el valor por omisión.
EUR	Estándar europeo de IBM (dd.mm.aaaa, hh.mm.ss)
JIS	Japanese Industrial Standard Christian Era (aaaa-mm-dd, hh:mm:ss)
HMS	Hora/Minutos/Segundos (hh:mm:ss)

-timesep=<*separador_hora***>**

Donde *separador_hora* es el carácter que desea utilizar para acceder a las columnas de resultado de hora. El separador de hora puede ser cualquiera de los siguientes:

Valor	Definición
:	Se utilizan dos puntos.
.	Se utiliza un punto. Este es el valor por omisión.
,	Se utiliza una coma.
blank	Se utiliza un espacio.

-decimalpt=<*separador_decimal***>**

Donde *separador_decimal* es el separador decimal que desea utilizar. El separador decimal se utiliza para las constantes numéricas en las sentencias SQL. El separador decimal puede ser cualquiera de los siguientes:

Valor	Definición
.	Se utiliza un punto. Este es el valor por omisión.
,	Se utiliza una coma.

-stmtCCSID=<*CCSID***>**

Donde *CCSID* es el identificador de juego de caracteres para las sentencias SQL preparadas en el paquete.El valor por omisión es el valor del trabajo durante el tiempo de personalización.

-sorttbl=<*nombre_biblioteca/nombre_tabla_secuencia_ordenación***>**

Donde *nombre_biblioteca/nombre_tabla_secuencia_ordenación* es la ubicación y el nombre de tabla de la tabla de secuencia de ordenación que desea utilizar. La tabla de secuencia de ordenación se utiliza para las comparaciones de series en las sentencias SQL. Los dos nombres, el de la biblioteca y el de la tabla de secuencia de ordenación, no pueden tener más de 10 caracteres. El valor por omisión se toma del trabajo durante el tiempo de personalización.

-langID=<*identificar_idioma***>**

Donde *identificador_idioma* es el identificador de idioma que desea utilizar. El valor por omisión para el identificador de idioma se toma del trabajo actual durante el tiempo de personalización. El identificador de idioma se utiliza junto con la tabla de secuencia de ordenación.

Para obtener información más detallada acerca de cualquiera de estos campos, consulte la publicación DB2 for iSeries SQL Programming Concepts, SC41-5611

.

Imprimir el contenido de los perfiles SQLJ de DB2 (db2profp y profp)

El impresor de perfiles SQLJ DB2, db2profp, imprime el contenido de un perfil personalizado DB2 en texto plano. El impresor de perfiles, profp, imprime el contenido de los perfiles generados por el conversor SQLJ en texto plano.

Para imprimir el contenido de los perfiles generados por el conversor SQLJ en texto plano, utilice el programa de utilidad profp de la manera siguiente:

```
profp MyClass_SJProfile0.ser
```

Donde *MyClass_SJProfile0.ser* es el nombre del perfil que desea imprimir.

Para imprimir el contenido de la versión personalizada de DB2 del perfil en texto plano, utilice el programa de utilidad db2profp de la manera siguiente:

```
db2profp MyClass_SJProfile0.ser
```

Donde *MyClass_SJProfile0.ser* es el nombre del perfil que desea imprimir.

Nota: si ejecuta db2profp en un perfil no personalizado, el programa le indicará esta circunstancia. Si ejecuta profp en un perfil personalizado, el programa visualiza el contenido del perfil sin la personalización.

Utilización y sintaxis del Impresor de perfiles SQLJ de DB2:

```
db2profp [opciones] <nombre_perfil_SQLJ>
```

Donde *nombre_perfil_SQLJ* es el nombre del perfil que debe imprimirse, y *opciones* es la lista de opciones que desea.

Las opciones disponibles para db2profp son las siguientes:

-URL=<*URL_JDBC***>**
: Donde *JDBC_URL* es el URL al que desea conectarse. Para obtener más información, consulte la sección Acceder a la base de datos de iSeries con el controlador JDBC de IBM Developer Kit para Java.

-user=<*nombreusuario***>**
: Donde *nombreusuario* es el nombre de usuario del perfil de usuario.

-password=<*contraseña***>**
: Donde *contraseña* es la contraseña del perfil de usuario.

Instalador de auditores de perfiles SQLJ (profdb)

El instalador de auditores de perfiles SQLJ (profdb) instala y desinstala auditores de clase de depuración. Los auditores de clase de depuración se instalan en un conjunto existente de perfiles binarios. Una vez instalados los auditores de clase de depuración, se anotan todas las llamadas a RTStatement y RTResultSet efectuadas durante la ejecución de la aplicación. Pueden anotarse en un archivo o en la salida estándar. Luego se pueden inspeccionar las anotaciones con objeto de verificar el comportamiento y rastrear los errores de la aplicación. Tenga en cuenta que solo se auditan las llamadas efectuadas en tiempo de ejecución a las interfaces RTStatement y RTResultSetcall subyacentes.

Para instalar auditores de clase de depuración, entre lo siguiente en el indicador de mandatos de Qshell:

```
profdb MyClass_SJProfile0.ser
```

Donde *MyClass_SJProfile0.ser* es el nombre del perfil generado por el conversor SQLJ.

Para desinstalar auditores de clase de depuración, entre lo siguiente en el indicador de mandatos de Qshell:

```
profdb -Cuninstall MyClass_SJProfile.ser
```

Donde *MyClass_SJProfile0.ser* es el nombre del perfil generado por el conversor SQLJ.

Para obtener más información acerca de las opciones de línea de mandatos de profdb, consulte SQLJ Frequently Asked Questions.

Convertir una instancia de perfil serializado a formato de clase Java mediante la herramienta de conversión de perfiles SQLJ (profconv)

La herramienta de conversión de perfiles SQLJ (profconv) convierte una instancia de perfil serializado a formato de clase Java. La herramienta profconv es necesaria debido a que algunos navegadores no dan soporte a cargar un objeto serializado desde un archivo de recurso asociado a un applet. Ejecute el programa de utilidad profconv para realizar la conversión.

Para ejecutar el programa de utilidad profconv, escriba lo siguiente en la línea de mandatos de Qshell:

```
profconv MyApp_SJProfile0.ser
```

donde *MyApp_SJProfile0.ser* es el nombre de la instancia de perfil que desea convertir.

La herramienta `profconv` invoca `sqlj -ser2class`. Las opciones de línea de mandatos están en sqlj.

Intercalar sentencias SQL en la aplicación Java

Las sentencias de SQL estático en SQLJ están en las cláusula SQLJ. Las cláusulas SQLJ empiezan por `#sql` y terminan en punto y coma (;).

Antes de crear cláusulas SQLJ en la aplicación Java, importe los siguientes paquetes:

- import java.sql.*;
- import sqlj.runtime.*;
- import sqlj.runtime.ref.*;

Las cláusulas SQLJ más sencillas son aquellas que pueden procesarse y que constan del símbolo `#sql` seguido de una sentencia SQL entre llaves. Por ejemplo, la siguiente cláusula SQLJ puede aparecer en cualquier lugar donde esté permitido que aparezca una sentencia Java:

```
#sql { DELETE FROM TAB };
```

El ejemplo anterior suprime todas las filas de una tabla denominada TAB.

Nota: para obtener información acerca de la compilación y ejecución de aplicaciones SQLJ, consulte la sección Compilar y ejecutar programas SQLJ.

En una cláusula de proceso SQLJ, los símbolos que aparecen dentro de las llaves son símbolos SQL o variables de lenguaje principal. Todas las variables del lenguaje principal se distinguen mediante el carácter de dos puntos (:). Los símbolos SQL nunca aparecen fuera de las llaves de una cláusula de proceso SQLJ. Por ejemplo, el siguiente método Java inserta sus argumentos en una tabla SQL:

```
public void insertIntoTAB1 (int x, String y, float z) throws SQLException
{
    #sql { INSERT INTO TAB1 VALUES (:x, :y, :z) };
}
```

El cuerpo del método consta de una cláusula de proceso SQLJ que contiene las variables de lenguaje principal x, y, y z. Para obtener más información acerca de las variables de lenguaje principal, consulte la sección Variables de lenguaje principal en SQLJ.

En general, los símbolos SQL son sensibles a mayúsculas y minúsculas (excepto los identificadores delimitados mediante comillas dobles) y pueden escribirse con mayúsculas, minúsculas o con una combinación de ellas. Sin embargo, los símbolos *Java* son sensibles a mayúsculas/minúsculas. A efectos de claridad en los ejemplos, los símbolos SQL no sensibles a mayúsculas/minúsculas están en mayúsculas, y los símbolos Java están en minúsculas o combinadas. A lo largo de este tema, se utilizan las minúsculas `null` para representar el valor "null" de Java y las mayúsculas `NULL` para representar el valor "null" de SQL.

En los programas SQLJ pueden aparecer los siguientes tipos de construcciones SQL:

- Consultas
  Por ejemplo, expresiones y sentencias SELECT.
- Sentencias de cambio de datos SQL (DML)
  Por ejemplo, INSERT, UPDATE, DELETE.
- Sentencias de datos
  Por ejemplo, FETCH, SELECT..INTO.
- Sentencias de control de transacción
  Por ejemplo, COMMIT, ROLLBACK, etc.
- Sentencias de lenguaje de definición de datos (DDL, también conocido como SML o lenguaje de manipulación de esquemas)
  Por ejemplo, CREATE, DROP, ALTER.
- Llamadas a procedimientos almacenados
  Por ejemplo, CALL MYPROC(:x, :y, :z)
- Invocaciones de funciones almacenadas
  Por ejemplo, VALUES(MYFUN(:x))

Para obtener un ejemplo de SQLJ intercalado, consulte el Ejemplo: intercalar sentencias SQL en la aplicación Java

Variables de lenguaje principal del lenguaje de consulta estructurada para Java: Los argumentos de las sentencias SQL intercaladas se pasan por medio de las variables del lenguaje principal. Las variables del lenguaje principal son las que se utilizan en el lenguaje principal y pueden aparecer en las sentencias SQL. Las variables de lenguaje principal tienen tres partes como máximo:

- Un signo de dos puntos (:) como prefijo.
- Una variable de lenguaje principal Java que es un identificador Java para un parámetro, variable o campo.
- Un identificador de modalidad de parámetro opcional.

  Este identificador de modalidad puede ser uno de los siguientes:
  IN, OUT o INOUT.

La evaluación de un identificador Java no tiene efectos colaterales en un programa Java, por lo que puede aparecer múltiples veces en el código Java generado para sustituir una cláusula SQLJ.

La consulta siguiente contiene la variable de lenguaje principal `:x`. Esta variable de lenguaje principal es la variable Java, el campo o el parámetro `x` que es visible en el ámbito que contiene la consulta.

```
SELECT COL1, COL2 FROM TABLE1 WHERE :x > COL3
```

Compilar y ejecutar programas SQLJ

Si el programa Java tiene sentencias SQLJ intercaladas, debe seguir un procedimiento especial para compilarlo y ejecutarlo.

1. Preparación del servidor para utilizar SQLJ.
2. Utilice el conversor de SQLJ, `sqlj`, en el código fuente Java con SQL intercalado para generar el código fuente Java y los perfiles asociados. Se genera un perfil para cada conexión.

   Por ejemplo, escriba el siguiente mandato:

   ```
   sqlj MyClass.sqlj
   ```

   donde *MyClass.sqlj* es el nombre del archivo SQLJ.

   En este ejemplo, el conversor de SQLJ genera el archivo de código fuente MyClass.java y los perfiles asociados. Los perfiles asociados se denominan MyClass_SJProfile0.ser, MyClass_SJProfile1.ser, MyClass_SJProfile2.ser, etc.

   Nota: el conversor de SQLJ compila automáticamente el código fuente Java convertido en un archivo de clase a menos que se desactive explícitamente la opción de compilación con la cláusula `-compile=false`.
3. Utilice la herramienta de personalización de perfiles SQLJ, `db2profc`, para instalar personalizadores SQLJ DB2 en los perfiles generados y crear los paquetes DB2 en el sistema local.

   Por ejemplo, escriba el mandato:

   ```
   db2profc MyClass_SJProfile0.ser
   ```

   donde *MyClass_SJProfile0.ser* es el nombre del perfil en el que se ejecuta DB2 SQLJ Customizer.

   Nota: este paso es opcional, pero se lo recomendamos porque aumenta el rendimiento en tiempo de ejecución.
4. Ejecute el archivo de clase Java igual que cualquier otro archivo de clase Java.

   Por ejemplo, escriba el mandato:

   ```
   java MyClass
   ```

   donde *MyClass* es el nombre del archivo de clase Java.

Rutinas SQL Java

El servidor iSeries proporciona la posibilidad de acceder a programas Java desde programas y sentencias SQL. Esta operación puede realizarse mediante procedimientos almacenados Java y funciones definidas por usuario (UDF) Java. El servidor iSeries soporta convenciones DB2 y SQLJ para llamar a procedimientos almacenados Java y UDF Java. Tanto los procedimientos almacenados Java como las UDF Java pueden utilizar clases Java almacenadas en archivos JAR. El servidor iSeries utiliza procedimientos almacenados definidos por el estándar *SQLJ Parte 1* para registrar los archivos JAR con la base de datos.

Para acceder a las aplicaciones Java desde programas y sentencias SQL, consulte las siguientes secciones:

Utilizar rutinas SQL Java
Para utilizar rutinas SQL Java, lleve a cabo los siguientes pasos:

- Habilitar SQLJ.
- Escribir los métodos Java para las rutinas.
- Compilar las clases Java.
- Hacer accesibles las clases compiladas para la máquina virtual Java utilizada por la base de datos.
- Registrar la rutina con la base de datos.
- Utilizar el procedimiento SQL Java.

Procedimientos almacenados Java
Al utilizar Java para escribir procedimientos almacenados, puede utilizar los siguientes estilos para pasar parámetros:

- Estilo de parámetro JAVA
- Estilo de parámetro DB2GENERAL

Funciones escalares Java definidas por usuario
Una función escalar Java devuelve un valor desde un programa Java a la base de datos. Al igual que los procedimientos almacenados Java, las funciones escalares Java utilizan uno de los dos estilos de parámetro, JAVA y DB2GENERAL.

Funciones de tabla definidas por usuario Java
DB2 proporciona la posibilidad de que una función devuelva una tabla. Esto resulta de utilidad para exponer información externa a la base de datos en formato de tabla.

Procedimientos SQLJ que manipulan archivos JAR
Tanto los procedimientos almacenados Java como las UDF Java pueden utilizar clases Java almacenadas en archivos JAR Java. Consulte la siguiente información relativa a los procedimientos SQLJ que manipulan archivos JAR:

- SQLJ.INSTALL_JAR
- SQLJ.REMOVE_JAR
- SQLJ.REPLACE_JAR
- SQLJ.UPDATEJARINFO
- SQLJ.RECOVERJAR

Convenciones de pase de parámetros para procedimientos almacenados y UDF Java
Esta sección describe cómo se representan los tipos de datos SQL en los procedimientos almacenados y UDF Java.

Utilizar rutinas SQL Java

Puede acceder a programas Java desde sentencias y programas SQL. Esta operación puede realizarse mediante procedimientos almacenados Java y funciones definidas por usuario (UDF) Java.

Para utilizar rutinas SQL Java, lleve a cabo las siguientes tareas:

1. Habilitar SQLJ

   Dado que cualquier rutina SQL Java puede utilizar SQLJ, haga que el soporte de ejecución SQLJ esté siempre disponible al ejecutar Java 2 Software Development Kit (J2SDK). Para habilitar el soporte de ejecución para SQLJ en J2SDK, añada un enlace al archivo runtime.zip de SQLJ desde el directorio de ampliaciones. Para obtener más información, consulte la página siguiente:

   Preparación del servidor para utilizar SQLJ
2. Escribir los métodos Java para las rutinas

   Una rutina SQL Java procesa un método Java desde SQL. Este método debe haberse escrito utilizando las convenciones de paso de parámetros de DB2 o SQLJ. Consulte las secciones Procedimientos almacenados Java, Funciones definidas por usuario Java y Funciones de tabla definidas por usuario Java para obtener más información acerca de cómo codificar un método utilizado por una rutina SQL Java.
3. Compilar las clases Java

   Las rutinas SQL Java escritas mediante el estilo de parámetro Java pueden compilarse sin ninguna configuración adicional. Sin embargo, las rutinas SQL Java que utilizan el estilo de parámetro DB2GENERAL deben ampliar las clases com.ibm.db2.app.UDF o com.ibm.db2.app.StoredProc. Estas clases se encuentran en el archivo JAR /QIBM/ProdData/Java400/ext/db2routines_classes.jar. Si se utiliza javac para compilar estas rutinas, este archivo JAR debe existir en la CLASSPATH. Por ejemplo, el siguiente mandato compila un archivo fuente Java que contiene una rutina que utiliza el estilo de parámetro DB2GENERAL:

   ```
   javac -DCLASSPATH=/QIBM/ProdData/Java400/ext/db2routines_classes.jar
   source.java
   ```
4. Hacer accesibles las clases compiladas para la JVM utilizada por la base de datos

Las clases definidas por usuario utilizadas por la máquina virtual Java (JVM) de la base de datos pueden residir en el directorio /QIBM/UserData/OS400/SQLLib/Function o en un archivo JAR registrado para la base de datos.

/QIBM/UserData/OS400/SQLLib/Function es el equivalente iSeries de /sqllib/function, el directorio en el que DB2 UDB almacena los procedimientos almacenados Java y las UDF Java en otras plataformas. Si la clase forma parte de un paquete Java, debe residir en el subdirectorio adecuado. Por ejemplo, si se crea la clase runit como parte del paquete foo.bar, el archivo runnit.class debe estar en el directorio del sistema de archivos integrado, /QIBM/ProdData/OS400/SQLLib/Function/foo/bar.

El archivo de clase también puede colocarse en un archivo JAR registrado para la base de datos. El archivo JAR se registra mediante el procedimiento almacenado SQLJ.INSTALL_JAR. Este procedimiento almacenado se utiliza para asignar un ID de JAR a un archivo JAR. Este ID de JAR se utiliza para identificar el archivo JAR en el que reside el archivo de clase. Consulte la sección Procedimientos SQLJ que manipulan archivos JAR para obtener más información acerca de SQLJ.INSTALL_JAR y sobre otros procedimientos almacenados para manipular archivos JAR.

5. Registrar la rutina con la base de datos

   Las rutinas SQL Java se registran con la base de datos mediante las sentencias SQL CREATE PROCEDURE y CREATE FUNCTION. Estas sentencias contienen los siguientes elementos:

   Palabras clave CREATE
   Las sentencias SQL destinadas a crear una rutina SQL Java empiezan por CREATE PROCEDURE o CREATE STATEMENT.

   Nombre de la rutina
   A continuación, la sentencia SQL identifica el nombre de la rutina conocido por la base de datos. Es el nombre utilizado para acceder a la rutina Java desde SQL.

   Parámetros y valor de retorno
   A continuación, la sentencia SQL identifica los parámetros y valores de retorno, si procede, para la rutina Java.

   LANGUAGE JAVA
   La sentencia SQL utiliza las palabras clave LANGUAGE JAVA para indicar que la rutina se ha escrito en Java.

   Palabras clave PARAMETER STYLE
   A continuación, la sentencia SQL identifica el estilo de parámetro mediante las palabras clave PARAMETER STYLE JAVA o PARAMETER STYLE DB2GENERAL.

   Nombre externo
   A continuación, la sentencia SQL identifica el método Java que debe procesarse como rutinas SQL Java. El nombre externo puede tener dos formatos:

   - Si el método se encuentra en un archivo de clase ubicado en el directorio /QIBM/UserData/OS400/SQLLib/Function, se identifica mediante el formato *nombreclase.nombremétodo*, donde *nombreclase* es el nombre totalmente calificado de la clase y *nombremétodo* es el nombre del método.
   - Si el método se encuentra en un archivo JAR registrado para la base de datos, se identifica mediante el formato *jarid*:*nombreclase.nombremétodo*, donde *jarid* es el ID de JAR del archivo JAR registrado, *nombreclase* es el nombre de la clase y *nombremétodo* es el nombre del método.

   Puede utilizarse iSeries Navigator para crear un procedimiento almacenado o una función definida por usuario que utilice el estilo de parámetro Java.

6. Utilizar el procedimiento Java

   Un procedimiento almacenado Java se llama mediante la sentencia SQL CALL. Una función UDF Java es aquella a la que se llama como parte de otra sentencia SQL.

Procedimientos almacenados Java

Al utilizar Java para escribir procedimientos almacenados, puede utilizar dos estilos posibles para pasar parámetros. El estilo recomendado es el estilo de parámetro JAVA, que coincide con el estilo de parámetro especificado en el estándar de rutinas SQLj: SQL. El segundo estilo, DB2GENERAL, es un estilo de parámetro definido por DB2 UDB. El estilo de parámetro también determina las convenciones que deben utilizarse al codificar un procedimiento almacenado Java.

Además, también debe tener conocimiento de algunas restricciones que se aplican a procedimientos almacenados Java.

Estilo de parámetro JAVA: Al codificar un procedimiento almacenado Java que utiliza el estilo de parámetro JAVA, debe utilizar las siguientes convenciones:

- El método Java debe ser un método público estático (no de instancia) void.
- Los parámetros del método Java deben ser tipos compatibles con SQL.
- Un método Java puede probar un valor SQL NULL cuando el parámetro es un tipo con capacidad de nulos (como String).
- Los parámetros de salida se devuelven utilizando matrices de un solo elemento.
- El método Java puede acceder a la base de datos actual utilizando el método getConnection.

Los procedimientos almacenados Java que utilizan el estilo de parámetro JAVA son métodos estáticos públicos. Dentro de las clases, los procedimientos almacenados se identifican mediante el nombre y la firma de método. Al llamar a un procedimiento almacenado, su firma se genera automáticamente, en función de los tipos de variable definidos por la sentencia CREATE PROCEDURE.

Si se pasa un parámetro en un tipo Java que permite el valor nulo, un método Java puede comparar el parámetro con null para determinar si un parámetro de entrada es SQL NULL.

Los siguientes tipos Java no dan soporte al valor nulo:

- short
- int
- long
- float
- double

Si se pasa un valor nulo a un tipo Java que no da soporte al valor nulo, se devolverá una excepción SQL con un código de error -20205.

Los parámetros de salida se pasan como matrices que contienen un elemento. El procedimiento almacenado Java puede establecer el primer elemento de la matriz para establecer el parámetro de salida.

Se accede a una conexión con el contexto de aplicación de incorporación utilizando la siguiente llamada JDBC (Java Database Connectivity):

```
connection=DriverManager.getConnection("jdbc:default:connection");
```

A continuación, esta conexión ejecuta sentencias SQL con las API JDBC.

A continuación se ofrece un pequeño procedimiento almacenado con un parámetro de entrada y dos parámetros de salida. Ejecuta la consulta SQL dada y devuelve el número de filas del resultado y SQLSTATE.

Ejemplo: procedimiento almacenado con una entrada y dos salidas

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
package mystuff;

import java.sql.*;
public class sample2 {
  public static void donut(String query, int[] rowCount,
    String[] sqlstate) throws Exception {
        try {
    Connection c=DriverManager.getConnection("jdbc:default:connection");
    Statement s=c.createStatement();
    ResultSet r=s.executeQuery(query);
    int counter=0;
    while(r.next()){
      counter++;
    }
    r.close(); s.close();
    rowCount[0] = counter;
    }catch(SQLException x){
    sqlstate[0]= x.getSQLState();
   }
  }
 }
```

En el estándar SQLj, para devolver un conjunto de resultados en rutinas que utilizan el estilo de parámetro JAVA, el conjunto de resultados debe establecerse explícitamente. Cuando se crea un procedimiento que devuelve conjuntos de resultados, se añaden parámetros adicionales de conjunto de resultados al final de la lista de parámetros. Por ejemplo, la sentencia

```
CREATE PROCEDURE RETURNTWO()
DYNAMIC RESULT SETS 2
LANGUAGE JAVA
PARAMETER STYLE JAVA
EXTERNAL NAME 'javaClass!returnTwoResultSets'
```

llamará a un método Java con la firma `public static void returnTwoResultSets(ResultSet[] rs1, ResultSet[] rs2)`.

Los parámetros de salida del conjunto de resultados deben establecerse explícitamente, como se muestra en el ejemplo siguiente. Al igual que en el estilo DB2GENERAL, los conjuntos de resultados y las sentencias correspondientes no deben cerrarse.

Ejemplo: procedimiento almacenado que devuelve dos conjuntos de resultados

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
   import java.sql.*;
public class javaClass {
  /**
    * Procedimiento almacenado Java, con parámetros de estilo JAVA,
    * que procesa dos sentencias predefinidas
    * y devuelve dos conjuntos de resultados
    *
    * @param ResultSet[] rs1    primer ResultSet
    * @param ResultSet[] rs2    segundo ResultSet
    */
public static void returnTwoResultSets (ResultSet[] rs1, ResultSet[] rs2) throws Exception
{
  // obtener conexión del llamador con la base de datos; heredado de StoredProc
  Connection con = DriverManager.getConnection("jdbc:default:connection");

  //definir y procesar la primera sentencia select
  Statement stmt1 = con.createStatement();
```

```
  String sql1 = "select value from table01 where index=1";
  rs1[0] = stmt1.executeQuery(sql1);

  //definir y procesar la segunda sentencia select
  Statement stmt2 = con.createStatement();
  Stringsql2 = "select value from table01 where index=2";
  rs2[0] = stmt2.executeQuery(sql2);
  }
}
```

En el servidor, los parámetros adicionales de conjunto de resultados no se examinan para determinar el orden de los resultados. Los conjuntos de resultados del servidor se devuelven en el orden en el que se han abierto. Para garantizar la compatibilidad con el estándar SQLj, el resultado debe asignarse en el orden en que se abre, como se ha mostrado anteriormente.

Estilo de parámetro DB2GENERAL: Al codificar un procedimiento almacenado Java que utiliza el estilo de parámetro DB2GENERAL, debe utilizar las siguientes convenciones:

- La clase que define un procedimiento almacenado Java debe *ampliar*, o ser una subclase de, la clase Java com.ibm.db2.app.StoredProc.
- El método Java debe ser un método público de instancia void.
- Los parámetros del método Java deben ser tipos compatibles con SQL.
- Un método Java puede probar un valor SQL NULL utilizando el método isNull.
- El método Java debe establecer explícitamente los parámetros de retorno utilizando el método set.
- El método Java puede acceder a la base de datos actual utilizando el método getConnection.

Una clase que incluya un procedimiento almacenado Java debe ampliar la clase com.ibm.db2.app.StoredProc. Los procedimientos almacenados Java son métodos públicos de instancia. Dentro de las clases, los procedimientos almacenados se identifican mediante el nombre y la firma de método. Al llamar a un procedimiento almacenado, su firma se genera automáticamente, en función de los tipos de variable definidos por la sentencia CREATE PROCEDURE.

La clase com.ibm.db2.app.StoredProc proporciona el método isNull, que permite a un método Java determinar si un parámetro de entrada es SQL NULL. La clase com.ibm.db2.app.StoredProc también proporciona métodos set...( ) que establecen parámetros de salida. Debe utilizar estos métodos para establecer parámetros de salida. Si no establece un parámetro de salida, el parámetro de salida devuelve el valor SQL NULL.

La clase com.ibm.db2.app.StoredProc proporciona la rutina siguiente para extraer una conexión JDBC con el contexto de aplicación de incorporación. Se accede a una conexión con el contexto de aplicación de incorporación utilizando la siguiente llamada JDBC:

```
public Java.sql.Connection getConnection( )
```

A continuación, esta conexión ejecuta sentencias SQL con las API JDBC.

A continuación se ofrece un pequeño procedimiento almacenado con un parámetro de entrada y dos parámetros de salida. Procesa la consulta SQL dada y devuelve el número de filas del resultado y SQLSTATE.

Ejemplo: procedimiento almacenado con una entrada y dos salidas

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
package mystuff;

import com.ibm.db2.app.*;
import java.sql.*;
```

```
public class sample2 extends StoredProc {
  public void donut(String query, int rowCount,
    String sqlstate) throws Exception {
           try {
    Statement s=getConnection().createStatement();
    ResultSet r=s.executeQuery(query);
    int counter=0;
    while(r.next()){
      counter++;
    }
    r.close(); s.close();
    set(2, counter);
    }catch(SQLException x){
    set(3, x.getSQLState());
    }
  }
}
```

Para devolver un conjunto de resultados en procedimientos que utilizan el estilo de parámetro DB2GENERAL, el conjunto de resultados y la sentencia que responde deben dejarse abiertos al final del procedimiento. El conjunto de resultados que se devuelve debe cerrarlo la aplicación cliente. Si se devuelven varios conjuntos de resultados, se devuelven en el orden en el que se han abierto. Por ejemplo, el siguiente procedimiento almacenado devuelve dos conjuntos de resultados.

Ejemplo: procedimiento almacenado que devuelve dos conjuntos de resultados

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
public void returnTwoResultSets() throws Exception
{
  // obtener conexión del llamador con la base de datos; heredado de StoredProc
  Connection con = getConnection ();
  Statement stmt1 = con.createStatement ();
  String sql1 = "select value from table01 where index=1";
  ResultSet rs1 = stmt1.executeQuery(sql1);
  Statement stmt2 = con.createStatement();
  String sql2 = "select value from table01 where index=2";
  ResultSet rs2 = stmt2.executeQuery(sql2);
}
```

Restricciones de procedimientos almacenados Java: Los procedimientos almacenados Java están sujetos a las siguientes restricciones:

- Un procedimiento almacenado Java no debe crear hebras adicionales. Sólo puede crearse una hebra adicional en un trabajo si el trabajo tiene capacidad multihebra. Puesto que no existe ninguna garantía de que un trabajo que llama a un procedimiento almacenado SQL tenga capacidad multihebra, un procedimiento almacenado Java no debe crear hebras adicionales.
- No puede utilizarse una autorización adoptada para acceder a archivos de clase Java.
- Un procedimiento almacenado Java siempre utiliza la versión más reciente de Java Development Kit instalada en el sistema.
- Dado que las clases Blob y Clob residen en los paquetes java.sql y com.ibm.db2.app, el programador debe utilizar el nombre completo de estas clases si ambas clases se utilizan en el mismo programa. El programa debe garantizar que las clases Blob y Clob de com.ibm.db2.app se utilizan como parámetros pasados al procedimiento almacenado.
- Cuando se crea un procedimiento almacenado Java, el sistema genera un programa de servicio en la biblioteca. Este programa de servicio se utiliza para almacenar la definición de procedimiento. El programa de servicio tiene un nombre generado por el sistema. Este nombre puede obtenerse examinando las anotaciones del trabajo que ha creado el procedimiento almacenado. Si el objeto de programa de servicio se guarda y luego se restaura, se restaura también la definición de procedimiento.

Si un procedimiento almacenado Java debe trasladarse de un sistema a otro, el usuario es responsable de trasladar el programa que contiene la definición de procedimiento, así como el archivo del sistema de archivos integrado, que contiene la clase Java.

- Un procedimiento almacenado Java no puede establecer las propiedades (por ejemplo, denominación de sistema) de la conexión JDBC utilizada para conectarse a la base de datos. Siempre se utilizan las propiedades de conexión JDBC por omisión, excepto cuando la preextracción está inhabilitada.

Funciones escalares Java definidas por usuario

Una **función escalar Java** devuelve un valor de un programa Java a la base de datos. Por ejemplo, puede crearse una función escalar que devuelva la suma de dos números. Al igual que los procedimientos almacenados Java, las funciones escalares Java utilizan uno de los dos estilos de parámetro, "Estilo de parámetro Java" y "Estilo de parámetro DB2GENERAL" en la página 148. Al codificar una función definida por usuario (UDF) Java, debe estar al corriente de las restricciones que se aplican a la creación de funciones escalares Java.

Estilo de parámetro Java: El estilo de parámetro Java es el estilo especificado por el estándar *SQLJ Part 1: SQL Routines*. Al codificar una UDF Java, utilice las siguientes convenciones.

- El método Java debe ser un método público estático.
- El método Java debe devolver un tipo compatible SQL. El valor de retorno es el resultado del método.
- Los parámetros del método Java deben ser tipos compatibles SQL.
- Un método Java puede probar un valor SQL NULL para los tipos Java que permiten el valor nulo.

Por ejemplo, dada una UDF denominada sample!test3 que devuelve INTEGER y toma argumentos de tipo CHAR(5), BLOB(10K) y DATE, DB2 espera que la implementación Java de la UDF tenga la firma siguiente:

```
import com.ibm.db2.app.*;
public class sample {
   public static int test3(String arg1, Blob arg2, Date arg3) { ... }
}
```

Los parámetros de un método Java deben ser tipos compatibles SQL. Por ejemplo, si se declara una UDF que toma argumentos de los tipos SQL t1, t2 y t3, y devuelve el tipo t4, se la llama como un método Java con la firma Java esperada:

```
public static T4 nombre (T1 a, T2 b, T3 c)  { .....}
```

donde:

- *nombre* es el nombre del método
- T1 a T4 son los tipos Java que corresponden a los tipos SQL t1 a t4.
- *a*, *b* y *c* son nombres de variable arbitrarios para los argumentos de entrada.

La correlación entre los tipos SQL y los tipos Java se encuentra en la sección Convenios de pase de parámetros para procedimientos almacenados y UDF.

Los valores SQL NULL se representan mediante variables Java que no se inicializan. Estas variables tiene un valor Java nulo si son tipos de objeto. Si se pasa un SQL NULL a un tipo de datos escalar Java, como por ejemplo int, se produce una condición de excepción.

Para devolver un resultado desde una UDF Java al utilizar el estilo de parámetro JAVA, simplemente devuelva el resultado del método.

```
{ ....
  valor de retorno;
}
```

Al igual que en los módulos C utilizados en las UDF y los procedimientos almacenados, no puede utilizar las corrientes de E/S estándar Java (System.in, System.out y System.err) en las UDF Java.

Estilo de parámetro DB2GENERAL: Las UDF Java utilizan el estilo de parámetro DB2GENERAL. En este estilo de parámetro, el valor de retorno se pasa como el último parámetro de la función y debe establecerse mediante un método *set* de la clase com.ibm.db2.app.UDF.

Al codificar una UDF Java, utilice las siguientes convenciones:

- La clase, que incluye la UDF Java, debe *ampliar*, o ser una subclase de, la clase Java com.ibm.db2.app.UDF.
- En el estilo de parámetro DB2GENERAL, el método Java debe ser un método de instancia void público.
- Los parámetros del método Java deben ser tipos compatibles con SQL.
- El método Java puede probar un valor SQL NULL utilizando el método isNull.
- En el estilo de parámetro DB2GENERAL, el método Java debe establecer explícitamente los parámetros de retorno utilizando el método set().

Una clase que incluya una UDF Java debe ampliar la clase Java com.ibm.db2.app.UDF. Una UDF Java que utilice el estilo de parámetro DB2GENERAL debe ser un método de instancia void de la clase Java. Por ejemplo, dada una UDF denominada sample!test3 que devuelve INTEGER y toma argumentos de tipo CHAR(5), BLOB(10K) y DATE, DB2 espera que la implementación Java de la UDF tenga la firma siguiente:

```
import com.ibm.db2.app.*;
public class sample extends UDF {
   public void test3(String arg1, Blob arg2, String arg3, int result) { ... }
}
```

Los parámetros de un método Java deben ser tipos SQL. Por ejemplo, si se declara una UDF que toma argumentos de los tipos SQL t1, t2 y t3, y devuelve el tipo t4, se la llama como un método Java con la firma Java esperada:

```
public void nombre (T1 a, T2 b, T3 c, T4 d)  { .....}
```

donde:

- *nombre* es el nombre del método
- T1 a T4 son los tipos Java que corresponden a los tipos SQL t1 a t4.
- *a*, *b* y *c* son nombres de variable arbitrarios para los argumentos de entrada.
- *d* es un nombre de variable arbitrario que representa el resultado de la UDF que se calcula.

La correlación entre los tipos SQL y los tipos Java se encuentra en la sección Convenios de pase de parámetros para procedimientos almacenados y UDF.

Los valores SQL NULL se representan mediante variables Java que no se inicializan. Estas variables tienen el valor cero si son tipos primitivos y el valor nulo Java si son tipos de objeto, según las normas de Java. Para indicar un valor SQL NULL que no sea un cero ordinario, puede llamarse al método isNull para un argumento de entrada:

```
   { ....
if (isNull(1)) { /* argument #1 was a SQL NULL */ }
else           { /* not NULL */ }
   }
```

En el ejemplo anterior, los número de argumento empiezan por el uno. La función isNull(), al igual que las demás funciones que siguen, se hereda de la clase com.ibm.db2.app.UDF. Para devolver un resultado desde una UDF Java al utilizar el estilo de parámetro DB2GENERAL, utilice el método set() en la UDF, como se indica a continuación:

```
   { ....
set(2, valor);
   }
```

Donde 2 es el índice de un argumento de salida y *valor* es un literal o variable de un tipo compatible. El número de argumento es el índice de la lista de argumentos de la salida seleccionada. En el primer ejemplo de esta sección, la variable de resultado int tiene un índice 4. Un argumento de salida que no se establece antes de que la UDF efectúe el retorno tiene un valor NULL.

Al igual que en los módulos C utilizados en las UDF y los procedimientos almacenados, no puede utilizar las corrientes de E/S estándar Java (System.in, System.out y System.err) en las UDF Java.

Generalmente, DB2 llama a una UDF muchas veces, una para cada fila de una entrada o conjunto de resultados de una consulta. Si se especifica SCRATCHPAD en la sentencia CREATE FUNCTION de la UDF, DB2 reconoce que es necesaria alguna "continuidad" entre las sucesivas llamadas de la UDF y, por tanto, en las funciones de estilo de parámetro DB2GENERAL, no se crea una instancia de la clase Java de implementación para cada llamada, sino que generalmente se crea una vez por referencia a UDF y por sentencia. Sin embargo, si se especifica NO SCRATCHPAD para una UDF, se crea una instancia pura por cada llamada a la UDF por medio de una llamada al constructor de la clase.

Un scratchpad puede ser de utilidad para guardar información a lo largo de las llamadas a una UDF. Las UDF Java pueden utilizar variables de instancia o establecer el scratchpad para conseguir la continuidad entre las llamadas. Las UDF Java acceden al scratchpad con los métodos getScratchPad y setScratchPad disponibles en com.ibm.db2.app.UDF. Al final de una consulta, si especifica la opción FINAL CALL en la sentencia CREATE FUNCTION, se llama al método public void close() del objeto (para las funciones del estilo de parámetro DB2GENERAL). Si no define este método, una función de apéndice toma el control y el evento se pasa por alto. La clase com.ibm.db2.app.UDF contiene variables y métodos útiles que pueden utilizarse con una UDF del estilo de parámetro DB2GENERAL. En la tabla siguiente se describen estas variables y métodos.

Variables y métodos	Descripción
public static final int SQLUDF_FIRST_CALL = -1; **public static final int SQLUDF_NORMAL_CALL = 0;** **public static final int SQLUDF_TF_FIRST = -2;** **public static final int SQLUDF_TF_OPEN = -1;** **public static final int SQLUDF_TF_FETCH = 0;** **public static final int SQLUDF_TF_CLOSE = 1;** **public static final int SQLUDF_TF_FINAL = 2;**	En las UDF escalares, se trata de constantes para determinar si la llamada es una primera llamada o una llamada normal. En las UDF de tabla, se trata de constantes para determinar si la llamada es una primera llamada, una llamada abierta, una llamada de extracción, una llamada de cierre o una llamada final.
public Connection getConnection();	El método contiene el handle de conexión JDBC para esta llamada de procedimiento almacenado y devuelve un objeto JDBC que representa la conexión de la aplicación que efectúa la llamada con la base de datos. Es análogo al resultado de una llamada a SQLConnect() nula de un procedimiento almacenado C.
public void close();	La base de datos llama a este método al final de una evaluación de UDF, si la UDF se ha creado con la opción FINAL CALL. Es análogo a la llamada final de una UDF C. Si una clase de UDF Java no implementa este método, este evento se pasa por alto.
public boolean isNull(int i)	Este método com comprueba si un argumento de entrada con el índice dado es SQL NULL.

Variables y métodos	Descripción
public void set(int i, short s); **public void set(int i, int j);** **public void set(int i, long j);** **public void set(int i, double d);** **public void set(int i, float f);** **public void set(int i, BigDecimal bigDecimal);** **public void set(int i, String string);** **public void set(int i, Blob blob);** **public void set(int i, Clob clob);** **public boolean needToSet(int i);**	Estos métodos establecen un argumento de salida en el valor dado. Se lanza una excepción si se produce alguna anomalía, incluyendo las siguientes: • La llamada de UDF no progresa • El índice no hace referencia a un argumento de salida válido • El tipo de datos no coincide • La longitud de los datos no coincide • Se produce un error de conversión de página de códigos
public void setSQLstate(String string);	Puede llamarse a este método desde una UDF para establecer el SQLSTATE que debe devolverse desde esta llamada. Si la serie no es aceptable como SQLSTATE, se lanza una excepción. El usuario puede establecer SQLSTATE en el programa externo para que devuelva un error o un aviso desde la función. En este caso, SQLSTATE debe contener uno de los siguientes elementos: • '00000' para indicar el éxito • '01Hxx', donde xx son dos dígitos o letras mayúsculas cualesquiera, para indicar un aviso • '38yxx', donde y es una letra mayúscula entre 'I' y 'Z' y xx son dos dígitos o letras mayúsculas cualesquiera, para indicar un error
public void setSQLmessage(String string);	Este método es parecido al método setSQLstate. Establece el resultado del mensaje SQL. Si la serie no es aceptable (por ejemplo, es superior a los 70 caracteres), se lanza una excepción.
public String getFunctionName();	Este método devuelve el nombre de la UDF de proceso.
public String getSpecificName();	Este método devuelve el nombre específico de la UDF de proceso.
public byte[] getDBinfo();	Este método devuelve una estructura DBINFO sin procesar para la UDF de proceso, como matriz de bytes. La UDF debe haberse registrado (mediante CREATE FUNCTION) con la opción DBINFO.
public String getDBname(); **public String getDBauthid();** **public String getDBver_rel();** **public String getDBplatform();** **public String getDBapplid();** **public String getDBapplid();** **public String getDBtbschema();** **public String getDBtbname();** **public String getDBcolname();**	Estos métodos devuelven el valor del campo adecuado de la estructura DBINFO de la UDF de proceso. La UDF debe haberse registrado (mediante CREATE FUNCTION) con la opción DBINFO. Los métodos getDBtbschema(), getDBtbname() y getDBcolname() sólo devuelven información significativa si se ha especificado una función definida por usuario a la derecha de una cláusula SET en una sentencia UPDATE.
public int getCCSID();	Este método devuelve el CCSID del trabajo.
public byte[] getScratchpad();	Este método devuelve una copia del scratchpad de la UDF de proceso actual. Primero debe declarar la UDF con la opción SCRATCHPAD.
public void setScratchpad(byte ab[]);	Este método sobrescribe el scratchpad de la UDF de proceso actual con el contenido de la matriz de bytes dada. Primero debe declarar la UDF con la opción SCRATCHPAD. La matriz de bytes debe tener el mismo tamaño que el devuelto por getScratchpad().

Variables y métodos	Descripción
public int getCallType();	Este método devuelve el tipo de llamada que se está efectuando actualmente. Estos valores corresponden a los valores C definidos en sqludf.h. Los valores de retorno posibles son los siguientes: • SQLUDF_FIRST_CALL • SQLUDF_NORMAL_CALL • SQLUDF_TF_FIRST • SQLUDF_TF_OPEN • SQLUDF_TF_FETCH • SQLUDF_TF_CLOSE • SQLUDF_TF_FINAL

Restricciones de funciones definidas por usuario Java: Las funciones definidas por usuario (UDF) Java están sujetas a las siguientes restricciones:

- Una UDF Java no debe crear hebras adicionales. Sólo puede crearse una hebra adicional en un trabajo si el trabajo tiene capacidad multihebra. Puesto que no existe ninguna garantía de que un trabajo que llama a un procedimiento almacenado SQL tenga capacidad multihebra, un procedimiento almacenado Java no debe crear hebras adicionales.
- El nombre completo del procedimiento almacenado Java definido en la base de datos está limitado a 279 caracteres. Este límite es consecuencia de la columna EXTERNAL_NAME, que tiene una anchura máxima de 279 caracteres.
- No puede utilizarse autorización adoptada para acceder a archivos de clase Java.
- Una UDF Java siempre utiliza la versión más reciente de JDK instalada en el sistema.
- Dado que las clases Blob y Clob residen en los paquetes java.sql y com.ibm.db2.app, el programador debe utilizar el nombre completo de estas clases si ambas clases se utilizan en el mismo programa. El programa debe garantizar que las clases Blob y Clob de com.ibm.db2.app se utilizan como parámetros pasados al procedimiento almacenado.
- Al igual que las funciones con código fuente, cuando se crea una UDF se utiliza un programa de servicio de la biblioteca para almacenar la definición de la función. El sistema genera el nombre del programa de servicio, que puede encontrarse en las anotaciones del trabajo que ha creado la función. Si este objeto se guarda y luego se restaura en otro sistema, se restaura también la definición de la función. Si una UDF Java debe trasladarse de un sistema a otro, el usuario es responsable de trasladar el programa de servicio que contiene la definición de la función, así como el archivo del sistema de archivos integrado que contiene la clase Java.
- Una UDF Java no puede establecer las propiedades (por ejemplo, denominación de sistema) de la conexión JDBC utilizada para conectarse a la base de datos. Siempre se utilizan las propiedades de conexión JDBC por omisión, excepto cuando la preextracción está inhabilitada.

Funciones de tabla definidas por usuario Java: DB2 proporciona la posibilidad de que una función devuelva una tabla. Esto resulta de utilidad para exponer información externa a la base de datos en formato de tabla. Por ejemplo, puede crearse una tabla que exponga las propiedades establecidas en la máquina virtual Java (JVM) utilizada para procedimientos almacenados y UDF (tanto de tabla como escalares) Java.

El estándar *SQLJ Part 1: SQL Routines* no da soporte a las funciones de tabla. En consecuencia, las funciones de tabla sólo están disponibles mediante el estilo de parámetro DB2GENERAL.

A una función de tabla se efectúan cinco tipos diferentes de llamadas. La tabla siguiente describe estas llamadas. Se presupone que se ha especificado scratchpad en la sentencia SQL de creación de función.

Punto en tiempo de exploración	NO FINAL CALL LANGUAGE JAVA SCRATCHPAD	FINAL CALL LANGUAGE JAVA SCRATCHPAD
Antes del primer OPEN de la función de tabla	Sin llamadas	Se llama al constructor de la clase (indica nuevo scratchpad). Se llama al método de la UDF con llamada FIRST.
En cada OPEN de la función de tabla.	Se llama al constructor de la clase (indica nuevo scratchpad). Se llama al método de la UDF con llamada OPEN.	Se llama al método de la UDF con llamada OPEN.
En cada FETCH de una fila nueva de datos de la función de tabla.	Se llama al método de la UDF con llamada FETCH.	Se llama al método de la UDF con llamada FETCH.
En cada CLOSE de la función de tabla.	Se llama al método de la UDF con llamada CLOSE. También se llama al método close(), si existe.	Se llama al método de la UDF con llamada CLOSE.
Después de la última CLOSE de la función de tabla.	Sin llamadas	Se llama al método de la UDF con llamada FINAL. También se llama al método close(), si existe.

Ejemplo: función de tabla Java: A continuación se ofrece un ejemplo de una función de tabla Java que determina las propiedades establecidas en la JVM utilizada para ejecutar la función de tabla definida por usuario Java.

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
     import com.ibm.db2.app.*;
 import java.util.*;

public class JVMProperties extends UDF {
      Enumeration propertyNames;
      Properties properties ;

      public void dump (String property, String value) throws Exception
      {
        int callType = getCallType();
        switch(callType) {
          case SQLUDF_TF_FIRST:
        break;
          case SQLUDF_TF_OPEN:
            properties = System.getProperties();
            propertyNames = properties.propertyNames();
        break;
          case SQLUDF_TF_FETCH:
            if (propertyNames.hasMoreElements()) {
                property = (String) propertyNames.nextElement();
                value = properties.getProperty(property);
                set(1, property);
                set(2, value);
    } else {
               setSQLstate("02000");
            }
        break;
          case SQLUDF_TF_CLOSE:
        break;
          case SQLUDF_TF_FINAL:
        break;
        default:
```

```
            throw new Exception("UNEXPECT call type of "+callType);
        }
      }
    }
```

Una vez compilada la función de tabla y su archivo de clase copiado en /QIBM/UserData/OS400/SQLLib/Function, la función puede registrarse en la base de datos mediante la siguiente sentencia SQL.

```
    create function properties()
returns table (property varchar(500), value varchar(500))
external name 'JVMProperties.dump' language java
parameter style db2general fenced no sql
disallow parallel scratchpad
```

Después de registrar la función, ésta puede utilizarse como parte de una sentencia SQL. Por ejemplo, la siguiente sentencia SELECT devuelve la tabla generada por la función de tabla.

```
    SELECT * FROM TABLE(PROPERTIES())
```

Procedimientos SQLJ que manipulan archivos JAR

Tanto los procedimientos almacenados Java como las UDF Java pueden utilizar clases Java almacenadas en archivos JAR Java. Para utilizar un archivo JAR, debe haber un *id-jar* asociado con el archivo JAR. El sistema proporciona procedimientos almacenados en el esquema SQLJ que permiten manipular *id-jar* y archivos JAR. Estos procedimientos permiten instalar, sustituir y eliminar archivos JAR. También proporcionan la posibilidad de utilizar y actualizar los catálogos SQL asociados con archivos JAR.

Para obtener más información, consulte los siguientes temas:

- SQLJ.INSTALL_JAR
- SQLJ.REMOVE_JAR
- SQLJ.REPLACE_JAR
- SQLJ.UPDATEJARINFO
- SQLJ.RECOVERJAR
- 
  SQLJ.REFRESH_CLASSES
  

SQLJ.INSTALL_JAR: El procedimiento almacenado SQLJ.INSTALL_JAR instala un archivo JAR en el sistema de bases de datos. Este archivo JAR puede utilizarse en sentencias CREATE FUNCTION y CREATE PROCEDURE subsiguientes.

Autorización: El privilegio que contiene el ID de autorización de la sentencia CALL debe incluir como mínimo uno de los siguientes valores para las tablas de catálogo SYSJAROBJECTS y SYSJARCONTENTS:

- Las siguientes autorizaciones de sistema:
  - Los privilegios INSERT y SELECT en la tabla
  - La autorización de sistema *EXECUTE sobre la biblioteca QSYS2
- Autorización administrativa

El privilegio que contiene el ID de autorización de la sentencia CALL también debe incluir las siguientes autorizaciones:

- Acceso de lectura (*R) sobre el archivo JAR especificado en el parámetro *jar-url* que se instala.

- Acceso de escritura, ejecución y lectura (*RWX) sobre el directorio donde está instalado el archivo JAR. Este directorio es /QIBM/UserData/OS400/SQLLib/Function/jar/*esquema*, donde *esquema* es el esquema de *jar-id*.

No puede utilizarse autorización adoptada para estas autorizaciones.

Sintaxis SQL:

```
>>-CALL--SQLJ.INSTALL_JAR-- (--'url-jar'--,--'id-jar'--,--despliegue--)-->

>-------------------------------------------------------------><
```

Descripción:

url-jar El URL que contiene el archivo JAR que debe instalarse o sustituirse. El único esquema de URL soportado es 'file:'.

id-jar El identificador de JAR de la base de datos que debe asociarse con el archivo especificado por el *url-jar*. El *id-jar* utiliza la denominación SQL y el archivo JAR se instala en el esquema o biblioteca especificados por el calificador implícito o explícito.

despliegue
Valor utilizado para describir la acción de instalación (install_action) del archivo del descriptor de despliegue. Si este entero es un valor no cero, las acciones de instalación (install_actions) de un archivo de descriptor de despliegue deben realizarse al final del procedimiento (install_jar). La versión actual de DB2 UDB para iSeries sólo soporta el valor cero.

Notas de utilización: Cuando se instala un archivo JAR, DB2 UDB para iSeries registra el archivo JAR en el catálogo de sistema SYSJAROBJECTS. También extrae los nombres de los archivos de clase Java del archivo JAR y registra cada clase en el catálogo de sistema SYSJARCONTENTS. DB2 UDB para iSeries copia el archivo JAR en un subdirectorio jar/schema del directorio /QIBM/UserData/OS400/SQLLib/Function. DB2 UDB para iSeries da a la nueva copia del archivo JAR el nombre que figura en la cláusula *id-jar*. Un archivo JAR instalado por DB2 UDB para iSeries en un subdirectorio de /QIBM/UserData/OS400/SQLLib/Function/jar no debe cambiarse. En lugar de ello, deben utilizarse los mandatos SQL CALL SQLJ.REMOVE_JAR y CALL SQLJ.REPLACE_JAR para eliminar o sustituir un archivo JAR instalado.

Ejemplo: El mandato siguiente se emite desde una sesión interactiva SQL.

```
CALL SQLJ.INSTALL_JAR('file:/home/db2inst/classes/Proc.jar' , 'myproc_jar', 0)
```

El archivo Proc.jar ubicado en el directorio file:/home/db2inst/classes/ se instala en DB2 UDB para iSeries con el nombre myproc_jar. Los mandatos SQL subsiguientes que utilizan el archivo Procedure.jar hacen referencia a él con el nombre myproc_jar.

SQLJ.REMOVE_JAR: El procedimiento almacenado SQLJ.REMOVE_JAR elimina un archivo JAR del sistema de bases de datos.

Autorización: El privilegio que contiene el ID de autorización de la sentencia CALL debe incluir como mínimo uno de los siguientes valores para las tablas de catálogo SYSJARCONTENTS y SYSJAROBJECTS:

- Las siguientes autorizaciones de sistema:
  - Los privilegios SELECT y DELETE en la tabla
  - La autorización de sistema *EXECUTE sobre la biblioteca QSYS2
- Autorización administrativa

El privilegio que contiene el ID de autorización de la sentencia CALL también debe incluir la siguiente autorización:

- La autorización *OBJMGT sobre el archivo JAR que se elimina. El archivo JAR se denomina /QIBM/UserData/OS400/SQLLib/Function/jar/schema/jarfile.

No puede utilizarse autorización adoptada para esta autorización.

Sintaxis:

```
>>-CALL--SQLJ.REMOVE_JAR--(--'id-jar'--,--deshacer despliegue--)----------><
```

Descripción:

id-jar Identificador JAR del archivo JAR que debe eliminarse de la base de datos.

deshacer despliegue
Valor utilizado para describir la acción de eliminación (remove_action) del archivo del descriptor de despliegue. Si este entero es un valor no cero, las acciones de eliminación (remove_actions) de un archivo de descriptor de despliegue deben realizarse al final del procedimiento install_jar. La versión actual de DB2 UDB para iSeries sólo soporta el valor cero.

Ejemplo: El mandato siguiente se emite desde una sesión interactiva SQL:

```
CALL SQLJ.REMOVE_JAR('myProc_jar', 0)
```

El archivo JAR myProc_jar se elimina de la base de datos.

SQLJ.REPLACE_JAR: El procedimiento almacenado SQLJ.REPLACE_JAR sustituye un archivo JAR en el sistema de bases de datos.

Autorización: El privilegio que contiene el ID de autorización de la sentencia CALL debe incluir como mínimo uno de los siguientes valores para las tablas de catálogo SYSJAROBJECTS y SYSJARCONTENTS:

- Las siguientes autorizaciones de sistema:
  - Los privilegios SELECT, INSERT y DELETE en la tabla
  - La autorización de sistema *EXECUTE sobre la biblioteca QSYS2
- Autorización administrativa

El privilegio que contiene el ID de autorización de la sentencia CALL también debe incluir las siguientes autorizaciones:

- Acceso de lectura (*R) sobre el archivo JAR especificado en el parámetro *jar-url* que se instala.
- La autorización *OBJMGT sobre el archivo JAR que se elimina. El archivo JAR se denomina /QIBM/UserData/OS400/SQLLib/Function/jar/schema/jarfile.

No puede utilizarse autorización adoptada para estas autorizaciones.

Sintaxis:

```
>>-CALL--SQLJ.REPLACE_JAR--(--'url-jar'--,--'id-jar'--)--------><
```

Descripción:

url-jar El URL que contiene el archivo JAR que debe sustituirse. El único esquema de URL soportado es 'file:'.

id-jar El identificador de JAR de la base de datos que debe asociarse con el archivo especificado por el *url-jar*. El *id-jar* utiliza la denominación SQL y el archivo JAR se instala en el esquema o biblioteca especificados por el calificador implícito o explícito.

Notas de utilización: El procedimiento almacenado SQLJ.REPLACE_JAR sustituye un archivo JAR que se ha instalado anteriormente en la base de datos mediante SQLJ.INSTALL_JAR.

Ejemplo: El mandato siguiente se emite desde una sesión interactiva SQL:

```
CALL SQLJ.REPLACE_JAR('file:/home/db2inst/classes/Proc.jar' , 'myproc_jar')
```

El archivo JAR actual al que hace referencia el *id-jar* myproc_jar se sustituye por el archivo Proc.jar ubicado en el directorio file:/home/db2inst/classes/.

SQLJ.UPDATEJARINFO: SQLJ.UPDATEJARINFO actualiza la columna CLASS_SOURCE de la tabla de catálogo SYSJARCONTENTS. Este procedimiento no forma parte del estándar SQLJ, pero lo utiliza el constructor de procedimientos almacenados de DB2 UDB para iSeries.

Autorización: El privilegio que contiene el ID de autorización de la sentencia CALL debe incluir como mínimo uno de los siguientes valores para la tabla de catálogo SYSJARCONTENTS:

- Las siguientes autorizaciones de sistema:
  - Los privilegios SELECT y UPDATEINSERT en la tabla
  - La autorización de sistema *EXECUTE sobre la biblioteca QSYS2
- Autorización administrativa

El usuario que ejecuta la sentencia CALL también debe tener las siguientes autorizaciones:

- Acceso de lectura (*R) sobre el archivo JAR especificado en el parámetro *jar-url*. Acceso de lectura (*R) sobre el archivo JAR que se instala.
- Acceso de escritura, ejecución y lectura (*RWX) sobre el directorio donde está instalado el archivo JAR. Este directorio es /QIBM/UserData/OS400/SQLLib/Function/jar/*esquema*, donde *esquema* es el esquema de *id-jar*.

No puede utilizarse autorización adoptada para estas autorizaciones.

Sintaxis:

```
>>-CALL--SQLJ.UPDATEJARINFO--(--'id-jar'--,--'id-clase'--,--'url-jar'--)-->

>-------------------------------------------------------------><
```

Descripción:

id-jar El identificador JAR de la base de datos que debe actualizarse.

id-clase
El nombre de clase calificado por paquete de la clase que debe actualizarse.

url-jar El URL que contiene el archivo de clase con el que debe actualizarse el archivo JAR. El único esquema de URL soportado es 'file:'.

Ejemplo: El mandato siguiente se emite desde una sesión interactiva SQL:

```
CALL SQLJ.UPDATEJARINFO('myproc_jar', 'mypackage.myclass',
                        'file:/home/user/mypackage/myclass.class')
```

El archivo JAR asociado con el *id-jar* myproc_jar se actualiza con una versión nueva de la clase mypackage.myclass. La versión nueva de la clase se obtiene del archivo /home/user/mypackage/myclass.class.

SQLJ.RECOVERJAR: El procedimiento SQLJ.RECOVERJAR toma el archivo JAR almacenado en el catálogo SYSJAROBJECTS y lo restaura en el archivo /QIBM/UserData/OS400/SQLLib/Function/jar/*jarschema*/*jar_id*.jar.

Autorización: El privilegio que contiene el ID de autorización de la sentencia CALL debe incluir como mínimo uno de los siguientes valores para la tabla de catálogo SYSJAROBJECTS:

- Las siguientes autorizaciones de sistema:
  - Los privilegios SELECT y UPDATEINSERT en la tabla
  - La autorización de sistema *EXECUTE sobre la biblioteca QSYS2
- Autorización administrativa

El usuario que ejecuta la sentencia CALL también debe tener las siguientes autorizaciones:

- Acceso de escritura, ejecución y lectura (*RWX) sobre el directorio donde está instalado el archivo JAR. Este directorio es /QIBM/UserData/OS400/SQLLib/Function/jar/*esquema*, donde *esquema* es el esquema de *id-jar*.
- La autorización *OBJMGT sobre el archivo JAR que se elimina. El archivo JAR se denomina /QIBM/UserData/OS400/SQLLib/Function/jar/schema/jarfile.

Sintaxis:

```
>>-CALL--SQLJ.RECOVERJAR--(--'id-jar'--)---------------------><
```

Descripción:

id-jar El identificador JAR de la base de datos que debe convertirse.

Ejemplo: El mandato siguiente se emite desde una sesión interactiva SQL:

```
CALL SQLJ.UPDATEJARINFO('myproc_jar')
```

El archivo JAR asociado con myproc_jar se actualiza con el contenido de la tabla SYSJARCONTENT. El archivo se copia en /QIBM/UserData/OS400/SQLLib/Function/jar/jar_schema myproc_jar.jar.

SQLJ.REFRESH_CLASSES: El procedimiento almacenado SQLJ.REFRESH_CLASSES provoca que vuelvan a cargarse las clases definidas por usuario utilizadas por los procedimientos almacenados Java o las UDF Java en la conexión de base de datos actual. A este procedimiento almacenado deben llamarlo las conexiones de base de datos existentes para obtener cambios realizados por una llamada al procedimiento almacenado SQLJ.REPLACE_JAR.

Autorización: NONE

Sintaxis:

```
>>-CALL--SQLJ.REFRESH_CLASSES-- ()-->
>-------------------------------------------------------------><
```

Ejemplo: Llame a un procedimiento almacenado Java, MYPROCEDURE, que utiliza una clase de un archivo jar registrado con el id de jar MYJAR:

```
CALL MYPROCEDURE()
```

Sustituya el archivo jar utilizando la siguiente llamada:

```
CALL SQLJ.REPLACE_JAR('MYJAR', '/tmp/newjarfile.jar')
```

Para hacer que las siguientes llamadas al procedimiento almacenado MYPROCEDURE utilicen el archivo jar actualizado, debe llamarse a SQLJ.REFRESH_CLASSES:

```
CALL SQLJ.REFRESH_CLASSES()
```

Vuelva a llamar al procedimiento almacenado. Se utilizan los archivos de clase actualizados al llamar al procedimiento.

```
CALL MYPROCEDURE()
```

Convenciones de pase de parámetros para procedimientos almacenados y UDF Java

La tabla siguiente ofrece una lista de la representación de los tipos de datos SQL en los procedimientos almacenados y las UDF Java.

Tipo de datos SQL	Estilo de parámetro Java JAVA	Estilo de parámetro Java DB2GENERAL
SMALLINT	short	short
INTEGER	int	int
BIGINT	long	long
DECIMAL(p,s)	BigDecimal	BigDecimal
NUMERIC(p,s)	BigDecimal	BigDecimal
REAL o FLOAT(p)	float	float
DOUBLE PRECISION, FLOAT o FLOAT(p)	double	double
CHARACTER(n)	Serie	Serie
CHARACTER(n) FOR BIT DATA	byte[]	com.ibm.db2.app.Blob
VARCHAR(n)	Serie	Serie
VARCHAR(n) FOR BIT DATA	byte[]	com.ibm.db2.app.Blob
GRAPHIC(n)	Serie	Serie
VARGRAPHIC(n)	Serie	Serie
DATE	Fecha	Serie
TIME	Hora	Serie
TIMESTAMP	Indicación de la hora	Serie
Variable de indicador	-	-
CLOB	-	com.ibm.db2.app.Clob
BLOB	-	com.ibm.db2.app.Blob
DBCLOB	-	com.ibm.db2.app.Clob
DataLink	-	-

Java con otros lenguajes de programación

Con Java, existen varias maneras de llamar al código escrito en un lenguaje distinto de Java.

Interfaz Java nativa

Una de las formas de llamar a código escrito en otro lenguaje consiste en implementar métodos Java seleccionados como 'métodos nativos'. Los métodos nativos son procedimientos, escritos en otro lenguaje, que proporcionan la implementación real de un método Java. Los métodos nativos pueden acceder a la máquina virtual Java utilizando la interfaz Java nativa (JNI). Estos métodos nativos se ejecutan en la hebra Java, que es una hebra del kernel, por lo que han de ser seguros en ejecución multihebra. Una función es segura en ejecución multihebra si puede iniciarse simultáneamente en varias hebras dentro de un mismo proceso. Asimismo, lo es si, y solo si, lo son también todas las funciones a las que llama.

Los métodos nativos constituyen el "puente" que permite acceder a las funciones del sistema no soportadas directamente en Java o que permite intercambiar información con el código de usuario existente. A la hora de utilizar métodos nativos, conviene ser precavido porque el código al que se llama puede no ser seguro en ejecución multihebra. Consulte la sección Utilizar la interfaz Java nativa para métodos nativos para obtener más información acerca de JNI y los métodos nativos ILE.

La API de invocación Java

El hecho de utilizar la API de invocación Java, que también forma parte de la especificación JNI (interfaz Java nativa), permite que una aplicación no Java pueda emplear la máquina virtual Java. Permite asimismo emplear código Java como ampliación de la aplicación.

Métodos nativos OS/400 PASE

La máquina virtual Java (JVM) de iSeries ahora admite el uso de métodos nativos en ejecución en el entorno OS/400 PASE. Los métodos nativos OS/400 PASE para Java permiten transportar fácilmente las aplicaciones Java que se ejecutan en AIX al servidor iSeries. Puede copiar los archivos de clase y las bibliotecas de métodos nativos de AIX en el sistema de archivos integrado del iSeries y ejecutarlos desde cualquier indicador de mandatos de CL (Control Language), Qshell o sesión de terminal OS/400 PASE.

Métodos nativos de teraespacio

La máquina virtual Java (JVM) ahora da soporte al uso de los métodos nativos del modelo de almacenamiento de teraespacio. El modelo de almacenamiento de teraespacio proporciona un entorno de procesos extensos con dirección local para programas ILE. Utilizar el teraespacio le permite llevar código de método nativo desde otros sistemas operativos a OS/400 con pocos o ningún cambio en el código fuente.

java.lang.Runtime.exec()

Para llamar a programas o mandatos desde dentro de un programa Java, puede utilizar java.lang.Runtime.exec(). El método exec() inicia otro proceso en el que se puede ejecutar cualquier mandato o programa de iSeries. En este modelo, para la comunicación entre procesos se puede utilizar la corriente de entrada, de salida y de error estándar del proceso hijo.

Comunicación entre procesos

Una opción es utilizar sockets para la comunicación entre el proceso padre y el proceso hijo.

También se pueden utilizar archivos continuos para la comunicación entre programas. O bien, consulte los ejemplos de comunicación entre procesos para obtener una visión general de las opciones existentes para comunicarse con programas que se ejecutan en otro proceso.

Para llamar a Java desde otros lenguajes, consulte el Ejemplo: Llamar a Java desde C o el Ejemplo: Llamar a Java desde RPG para obtener más información.

También puede utilizar IBM Toolbox para Java para llamar a programas y mandatos existentes en el servidor iSeries. Para la comunicación entre procesos con IBM Toolbox para Java se suelen utilizar las colas de datos y los mensajes de iSeries.

Nota: la utilización de Runtime.exec(), IBM Toolbox para Java o JNI puede poner en peligro la portabilidad del programa Java. Debe evitar el uso de estos métodos en un entorno Java ″puro″.

Utilizar la interfaz nativa Java para métodos nativos

Los métodos nativos deben utilizarse únicamente en aquellos casos en que Java puro no responde a las necesidades de programación. Debe limitar el uso de métodos nativos y emplearlos solo en las circunstancias siguientes:

- Para acceder a funciones del sistema que no están disponibles por medio de Java puro.
- Para implementar métodos extremadamente sensibles al rendimiento que pueden beneficiarse en gran medida de una implementación nativa.
- Para intercambiar información con las interfaces de programas de aplicación (API) existentes que permitan a Java llamar a otras API.

Las siguientes instrucciones corresponden al uso de la interfaz Java nativa (JNI) con el lenguaje C. Para obtener información sobre el uso de JNI con el lenguaje RPG, consulte la siguiente documentación:

Capítulo 11 de la publicación WebSphere Development Studio: ILE RPG Programmer's Guide, SC09-2507

Si desea utilizar la interfaz Java nativa (JNI) para los métodos nativos, siga estos pasos:

1. Diseñe la clase especificando, por medio de la sintaxis estándar del lenguaje Java, qué métodos son nativos.
2. Escoja un nombre de biblioteca y programa para el programa de servicio (*SRVPGM) que contiene las implementaciones de método nativo. Cuando codifique la llamada a método System.loadLibrary() en el inicializador estático de la clase, especifique el nombre del programa de servicio.
3. Utilice la herramienta `javac` para compilar el fuente Java y obtener un archivo de clase.
4. Utilice la herramienta `javah` para crear el archivo de cabecera (.h). Este contiene los prototipos exactos para crear las implementaciones de método nativo. La opción -d especifica el directorio en el que debe crearse el archivo de cabecera.
5. Copie el archivo de cabecera del sistema de archivos integrado en un miembro de un archivo fuente; para ello, utilice el mandato Copiar desde archivo continuo (CPYFRMSTMF). Debe copiar el archivo de cabecera en un miembro de archivo fuente para que el compilador C pueda utilizarlo. Emplee el nuevo soporte de archivos continuos del mandato Crear programa ILE C/400 enlazado (CRTCMOD) para dejar los archivos fuente y de cabecera C en el sistema de archivos integrado. Para obtener más información acerca del mandato CRTCMOD y la utilización de archivos continuos, consulte la publicación WebSphere Development Studio: ILE C/C++ Programmer's Guide, SC09-2712

   .
6. Escriba el código de método nativo. En Consideraciones entorno a las hebras y los métodos nativos Java hallará información detallada sobre los lenguajes y las funciones que se utilizan para los métodos nativos.
   a. Incluya el archivo de cabecera creado en los pasos anteriores.
   b. Correlacione de manera exacta los prototipos del archivo de cabecera.
   c. Convierta las series al formato ASCII (American Standard Code for Information Interchange) si deben pasarse series a la máquina virtual Java. En Codificaciones de caracteres Java hallará más información.
7. Si el método nativo ha de interaccionar con la máquina virtual Java, utilice las funciones que se proporcionan con JNI.
8. 

   Compile el código fuente C, utilizando el mandato CRTCMOD, en un objeto módulo (*MODULE).

9. Enlace uno o varios objetos módulo para crear un programa de servicio (*SRVPGM); para ello, utilice el mandato Crear programa de servicio (CRTSRVPGM). El nombre de este programa de servicio debe coincidir con el nombre que ha proporcionado en el código Java que se halla en las llamadas a función System.load() o System.loadLibrary().
10. Si ha utilizado la llamada System.loadLibrary() en el código Java, realice una de las siguientes tareas adecuadas para el J2SDK que esté ejecutando:
    - Incluir la lista de bibliotecas necesarias en la variable de entorno LIBPATH. Puede cambiar la variable de entorno LIBPATH en QShell y desde la línea de mandatos de iSeries.
      - En el indicador de mandatos de Qshell, escriba:
        ```
        export LIBPATH=/QSYS.LIB/MYLIB.LIB
        java -Djava.version=1.4 myclass
        ```
      - O bien, en la línea de mandatos, escriba:
        ```
        ADDENVVAR LIBPATH '/QSYS.LIB/MYLIB.LIB'
        JAVA PROP((java.version 1.4)) myclass
        ```
    - También puede suministrar la lista en la propiedad **java.library.path**. Puede cambiar la propiedad java.library.path en QShell y desde la línea de mandatos de iSeries.
      - En el indicador de mandatos de Qshell, escriba:
        ```
        java -Djava.library.path=/QSYS.LIB/MYLIB.LIB -Djava.version=1.4 myclass
        ```
      - O bien, en la línea de mandatos de iSeries, escriba:
        ```
        JAVA PROP((java.library.path '/QSYS.LIB/MYLIB.LIB') (java.version '1.4')) myclass
        ```

    Donde */QSYS.LIB/MYLIB.LIB* es la biblioteca que desea cargar utilizando la llamada System.loadLibrary(), y *myclass* es el nombre de la aplicación Java.
11. 

    La sintaxis de la vía de acceso para System.load(String path) puede ser cualquiera de las siguientes:
    - /qsys.lib/sysNMsp.srvpgm (para *SRVPGM QSYS/SYSNMSP)
    - /qsys.lib/mylib.lib/myNMsp.srvpgm (para *SRVPGM MYLIB/MYNMSP)
    - un enlace simbólico, por ejemplo /home/mydir/myNMsp.srvpgm que enlace con /qsys.lib/mylib.lib/myNMsp.srvpgm

      Nota: esto equivale a utilizar el método System.loadLibrary("myNMsp").

    Nota: El nombre de vía de acceso suele ser un literal de serie entre comillas. Por ejemplo, podría utilizar el siguiente código:

    ```
    System.load("/qsys.lib/mylib.lib/myNMsp.srvpgm")
    ```
12. El parámetro libname para System.loadLibrary(String libname) suele ser un literal de serie entre comillas que identifica la biblioteca de método nativo. El sistema utiliza la lista de bibliotecas actual y las variables de entorno LIBPATH y PASE_LIBPATH para buscar un programa de servicio o un ejecutable OS/400 PASE PASE que coincida con el nombre de biblioteca. Por ejemplo, `loadLibrary("myNMsp")` da como resultado la búsqueda de un *SRVPGM denominado MYNMSP o un ejecutable OS/400 PASE denominado libmyNMsp.a o libmyMNsp.so.

Para obtener una descripción completa de JNI, consulte Java Native Interface by Sun Microsystems, Inc., y The Source for Java Technology java.sun.com

.

Consulte los Ejemplos: utilizar la interfaz nativa Java para métodos nativos para obtener un ejemplo de utilización de JNI para métodos nativos.

API de invocación Java

La API de invocación, que forma parte de la interfaz Java nativa (JNI), permite al código no Java crear una máquina virtual Java, así como cargar y utilizar clases Java. Esta función permite a un programa multihebra utilizar las clases Java que se ejecutan en múltiples hebras de una sola máquina virtual Java.

IBM Developer Kit para Java da soporte a la API de invocación Java para los siguientes tipos de llamantes:

- Un programa ILE o un programa de servicio creado para `STGMDL(*SNGLVL)` y `DTAMDL(*P128)`
- Un programa ILE o un programa de servicio creado para `STGMDL(*TERASPACE)` y `DTAMDL(*LLP64)`
- Un ejecutable OS/400 PASE creado para AIX de 32 bits o de 64 bits

La aplicación controla la máquina virtual Java. La aplicación puede crear la máquina virtual Java, llamar a métodos Java (de forma parecida a cómo llama una aplicación a las subrutinas) y destruir la máquina virtual Java. Una vez creada, la máquina virtual Java está preparada para ejecutarse dentro del proceso hasta que la aplicación la destruye de manera explícita. Mientras se destruye, la máquina virtual Java realiza operaciones de borrado como, por ejemplo, ejecutar finalizadores, finalizar las hebras de la máquina virtual Java y liberar los recursos de la máquina virtual Java.

Con una máquina virtual Java preparada para ejecutarse, una aplicación escrita en lenguajes ILE, tales como C y RPG, puede llamar a la máquina virtual Java para que realice cualquier función.

También puede regresar de la máquina virtual Java a la aplicación C, llamar de nuevo a la máquina virtual Java y así sucesivamente. La máquina virtual Java se crea una vez y no hace falta crearla nuevamente antes de llamarla para que ejecute código Java (poco o mucho).

Cuando se utiliza la API de invocación para ejecutar programas Java, el destino de STDOUT y STDERR se controla por medio de una variable de entorno llamada QIBM_USE_DESCRIPTOR_STDIO. Si está establecida en Y o I (por ejemplo, QIBM_USE_DESCRIPTOR_STDIO=Y), la máquina virtual Java utiliza descriptores de archivo para STDIN (fd 0), STDOUT (fd 1) y STDERR (fd 2). En este caso, el programa debe establecer los descriptores de archivo en valores válidos abriéndolos como los tres primeros archivos o conductos del trabajo. Al primer archivo abierto en el trabajo se le da 0 como fd, al segundo 1 y al tercero 2. Para los trabajos iniciados con la API de engendramiento, estos descriptores se pueden preasignar mediante una correlación de descriptores de archivo (consulte la documentación de la API de engendramiento). Si la variable de entorno QIBM_USE_DESCRIPTOR_STDIO no está establecida o bien lo está en cualquier otro valor, no se utilizan descriptores de archivo para STDIN, STDOUT y STDERR. En lugar de ello, se direcciona STDOUT y STDERR a un archivo en spool propiedad del trabajo actual y la utilización de STDIN da como resultado una excepción de E/S.

Si desea obtener un ejemplo que utiliza la API de llamada, consulte el Ejemplo: API de llamada Java. Consulte la sección Funciones de la API de llamada para obtener detalles acerca de las funciones de la API de llamada soportadas por IBM Developer Kit para Java.

Funciones de la API de invocación: IBM Developer Kit para Java da soporte a estas funciones de la API de invocación.

Nota: antes de usar esta API, debe asegurarse de que está en un trabajo con capacidad multihebra. Consulte la sección Aplicaciones multihebra para obtener más información acerca de los trabajos con capacidad multihebra.

- **JNI_GetCreatedJavaVMs**

  Devuelve información sobre todas las máquina virtuales Java que se han creado.

  Aunque esta API está diseñada para devolver información para múltiples máquinas virtuales Java (JVM), solamente puede existir una JVM para un proceso. Por consiguiente, esta API devolverá un máximo de una JVM.

  Firma:

  ```
  jint JNI_GetCreatedJavaVMs(JavaVM **vmBuf,
                jsize bufLen,
                jsize *nVMs);
  ```

  vmBuf es un área de salida cuyo tamaño viene determinado por bufLen, que es el número de punteros. Cada máquina virtual Java tiene una estructura JavaVM asociada que está definida en java.h.

  Esta API almacena un puntero que señala hacia la estructura JavaVM que está asociada a cada una de las máquinas virtuales Java creadas en vmBuf, a menos que vmBuf sea 0.

  Los punteros que señalan a estructuras JavaVM se almacenan en el orden de las máquinas virtuales Java correspondientes que se crean. nVMs devuelve el número de máquinas virtuales que hay creadas actualmente. El servidor iSeries da soporte a la creación de más de una máquina virtual Java, por lo que cabe esperar un valor superior a uno. Esta información, junto con el tamaño de vmBuf, determina si se devuelven o no los punteros que señalan hacia las estructuras JavaVM de cada una de las máquinas virtuales Java creadas.

- **JNI_CreateJavaVM**

  Permite al usuario crear una máquina virtual Java y utilizarla después en una aplicación.

  Firma:

  ```
  jint JNI_CreateJavaVM(JavaVM **p_vm,
                void **p_env,
                void *vm_args);
  ```

  p_vm es la dirección de un puntero de JavaVM para la máquina virtual Java de nueva creación. Hay otras API de invocación de JNI que utilizan p_vm para identificar la máquina virtual Java. p_env es la dirección de un puntero de Entorno JNI para la máquina virtual Java de nueva creación. Señala hacia una tabla de funciones de JNI que inician dichas funciones. vm_args es una estructura que contiene los parámetros de inicialización de la máquina virtual Java.

  Si se inicia un mandato Ejecutar Java (RUNJVA) o JAVA y se especifica una propiedad que tenga un parámetro de mandato equivalente, este tiene preferencia. Se hace caso omiso de la propiedad. Por ejemplo, el parámetro os400.optimization no se tiene en cuenta en el mandato siguiente:

  ```
  JAVA CLASS(Hello) PROP((os400.optimization 0))
  ```

  Para obtener una lista de las propiedades exclusivas de 0S/400 soportadas por la API JNI_CreateJavaVM, consulte el apartado Propiedades Java del sistema.

  Nota: Java en el servidor iSeries da soporte a la creación de una sola máquina virtual Java (JVM) dentro de un trabajo o proceso individual. Para obtener más información, consulte el apartado Soporte para múltiples máquinas virtuales Java.

- **DestroyJavaVM**

  Destruye la máquina virtual Java.

Firma:

```
jint DestroyJavaVM(JavaVM *vm)
```

Cuando se crea la máquina virtual Java, vm es el puntero de JavaVM devuelto.

- **AttachCurrentThread**

  Conecta una hebra con una máquina virtual Java para que la hebra pueda utilizar los servicios de la máquina virtual Java.

  Firma:

```
jint AttachCurrentThread(JavaVM *vm,
                         void **p_env,
                         void *thr_args);
```

  El puntero de JavaVM, vm, identifica la máquina virtual Java a la que se conecta la hebra. p_env es el puntero que señala hacia la ubicación en la que está situado el puntero de interfaz JNI de la hebra actual. thr_args contiene argumentos de conexión de hebra específicos de la VM.

- **DetachCurrentThread**

  Firma:

```
jint DetachCurrentThread(JavaVM *vm);
```

  vm identifica la máquina virtual Java de la que se desconecta la hebra.

Para obtener una descripción completa de las funciones de la API de invocación, consulte Java Native Interface Specification by Sun Microsystems, Inc., o The Source for Java Technology java.sun.com

.

Soporte para varias máquinas virtuales Java: A partir de la V5R3, Java en el servidor iSeries ya no da soporte a la creación de más de una máquina virtual Java (JVM) dentro de un trabajo o proceso individual. Esta restricción afecta solamente a aquellos usuarios que crean las JVM utilizando la API de invocación de interfaz Java nativa (JNI). Este cambio en el soporte no afecta a cómo utiliza el mandato java para ejecutar los programas Java.

No puede llamar a JNI_CreateJavaVM() satisfactoriamente más de una vez en un trabajo y JNI_GetCreatedJavaVMs() no puede devolver más de una JVM en una lista de resultados.

El soporte para crear solamente una JVM individual dentro de un solo trabajo o proceso sigue los estándares de la implementación de referencia de Java de Sun Microsystems, Inc.

Métodos nativos Java y consideraciones acerca de las hebras

Puede utilizar métodos nativos para acceder a funciones que no están disponibles en Java.

Para utilizar mejor Java junto con los métodos nativos, es necesario tener claros los conceptos siguientes:

- Una hebra Java, tanto si la ha creado Java como si es una hebra nativa conectada, tiene inhabilitadas todas las excepciones de coma flotante. Si la hebra ejecuta un método nativo que vuelve a habilitar las excepciones de coma flotante, Java no las desactivará por segunda vez. Si la aplicación de usuario no las inhabilita antes de regresar para ejecutar el código Java, es posible que el comportamiento de este no sea el correcto si se produce una excepción de coma flotante. Cuando una hebra nativa se desconecta de la máquina virtual Java, su máscara de excepción de coma flotante se restaura en el valor que estaba en vigor en el momento de conectarse.
- Cuando una hebra nativa se conecta a la máquina virtual Java, esta cambia la prioridad de las hebras, si conviene, para ajustarse a los esquemas de prioridad de uno a diez que define Java. Cuando la hebra

se desconecta, la prioridad se restaura. Después de conectarse, la hebra puede cambiar la prioridad de hebra utilizando una interfaz de método nativo (por ejemplo, una API POSIX). Java no cambiará la prioridad de hebra en las transiciones de regreso a la máquina virtual Java.

- El componente API de invocación de la interfaz Java nativa (JNI) permite a un usuario intercalar una máquina virtual Java en su aplicación. Si una aplicación crea una máquina virtual Java y ésta finaliza de manera anómala, se indica la excepción MCH74A5 de iSeries, "Máquina virtual Java terminada", a la hebra inicial del proceso si dicha hebra estaba conectada a la máquina virtual Java en el momento de finalizar ésta. La máquina virtual Java podría finalizar anormalmente por cualquiera de las razones siguientes:
  - El usuario llama al método java.lang.System.exit().
  - Finaliza una hebra que la máquina virtual Java necesita.
  - Se produce un error interno en la máquina virtual Java.

  Este comportamiento es distinto al de la mayoría de las plataformas Java. En ellas, el proceso que crea automáticamente la máquina virtual Java finaliza de manera brusca tan pronto como finaliza la máquina virtual Java. La aplicación, si supervisa y maneja una excepción MCH74A5 indicada, puede seguir ejecutándose. De lo contrario, el proceso finaliza si la excepción queda sin manejar. Si se añade el código que se encarga de la excepción MCH74A5 específica del servidor iSeries, podría verse mermado el grado de portabilidad de la aplicación a otras plataformas.

Dado que los métodos nativos se ejecutan siempre en un proceso multihebra, el código que contienen debe ser seguro en ejecución multihebra. Este hecho impone a los lenguajes y a las funciones que se utilizan para los métodos nativos las siguientes restricciones:

- No se debe utilizar ILE CL para métodos nativos, ya que este lenguaje no es seguro en ejecución multihebra. Para ejecutar mandatos CL seguros en ejecución multihebra, puede utilizar la función system() del lenguaje C o el método java.lang.Runtime.exec().
  - Para ejecutar mandatos CL seguros en ejecución multihebra desde un método nativo C o C++, utilice la función system() del lenguaje C.
  - Para ejecutar mandatos CL seguros en ejecución multihebra directamente desde Java, utilice el método java.lang.Runtime.exec().
- Se puede utilizar ILE C, ILE C++, ILE COBOL e ILE RPG para escribir un método nativo, pero todas las funciones a las que se llame desde dentro del método nativo deben ser seguras en ejecución multihebra.

  Nota: El soporte en tiempo de compilación para escribir métodos nativos solo se proporciona actualmente en los lenguajes C, C++ y RPG. Escribir métodos nativos en otros lenguajes, si bien es posible, puede resultar mucho más complicado.

  Atención:
  No todas las funciones estándar C, C++, COBOL o RPG son seguras en ejecución multihebra.
- Las funciones exit() y abort() de C y C++ no deben utilizarse nunca dentro de un método nativo. Estas funciones provocan la detención de todo el proceso que se ejecuta en la máquina virtual Java. Esto incluye todas las hebras del proceso, sean o no originarias de Java.

  Nota: la función exit() a la que se hace referencia es la función de C y C++, y no es igual que el método java.lang.Runtime.exit().

Para obtener más información acerca de las hebras en el servidor iSeries, consulte la sección Aplicaciones multihebra.

Los métodos nativos y la interfaz nativa Java (JNI)

Los métodos nativos son métodos Java que se inician en un lenguaje distinto de Java. Pueden acceder a interfaces de programas de aplicación (API) y a funciones específicas del sistema que no están disponibles directamente en Java.

La utilización de métodos nativos limita la portabilidad de una aplicación porque en ellos interviene código específico del sistema. Un método nativo puede consistir en sentencias de código nativo nuevas o en sentencias de código nativo que llaman a código nativo existente.

Una vez que haya decidido que se necesita un método nativo, es posible que éste tenga que interaccionar con la máquina virtual Java en la que se ejecuta. La interfaz Java nativa (JNI) hace más fácil esta interoperatividad de una manera neutral por lo que a la plataforma se refiere.

JNI es un conjunto de interfaces que permiten a un método nativo interaccionar con la máquina virtual Java de muchas maneras. Por ejemplo, JNI incluye interfaces que crean objetos nuevos y llaman a métodos, que obtienen campos y los establecen, que procesan excepciones y manipulan series y matrices.

Para obtener una descripción completa de JNI, consulte Java Native Interface by Sun Microsystems, Inc., o The Source for Java Technology java.sun.com

.

Las series en los métodos nativos

Muchas funciones de la interfaz nativa Java (JNI) aceptan series de estilo de lenguaje C como parámetros. Por ejemplo, la función FindClass() de JNI acepta un parámetro de tipo serie que especifica el nombre totalmente calificado de un archivo de clase. Si se encuentra el archivo de clase, la función FindClass lo cargará, y al llamador de FindClass se le devolverá una referencia al archivo de clase.

Todas las funciones de JNI esperan que los parámetros de tipo serie estén codificados en UTF-8. Para obtener detalles acerca de UTF-8 puede consultar la especificación JNI, pero en la mayoría de los casos es suficiente con observar que los caracteres ASCII (American Standard Code for Information Interchange) de 7 bits son equivalentes a su representación en UTF-8. Los caracteres ASCII de 7 bits son en realidad caracteres de 8 bits, pero su primer bit siempre es 0. Por lo tanto, la mayoría de las series ASCII C ya tienen en realidad el formato UTF-8.

El compilador C del servidor iSeries opera en EBCDIC (extended binary-coded decimal interchange code) por omisión, por lo que puede suministrar series a las funciones de JNI en UTF-8. Existen dos maneras de hacerlo. Se pueden utilizar series literales o bien series dinámicas. Las series literales son aquellas cuyo valor es conocido en el momento de compilar el código fuente. Las series dinámicas son aquellas cuyo valor no se conoce durante la compilación, sino que se calcula realmente durante la ejecución.

Series literales en métodos nativos: Resulta más fácil codificar las series literales en UTF-8 si la serie está compuesta por caracteres con una representación ASCII (American Standard for Information Interchange) de 7 bits. Si la serie puede representarse en ASCII, como ocurre con la mayoría, puede ir entre sentencias 'pragma' que modifiquen la página de códigos actual del compilador. Entonces, el compilador almacenará internamente la serie en el formato UTF-8 que JNI requiere. Si la serie no puede representarse en ASCII, es más fácil tratar la serie EBCDIC original como si fuese una serie dinámica y procesarla con iconv() antes de pasarla a JNI. Para obtener más información acerca de las series dinámicas, consulte el apartado Series dinámicas.

Por ejemplo, para buscar una clase denominada `java/lang/String`, el código será el siguiente:

```
#pragma convert (819)
 myClass = (*env)->FindClass(env,"java/lang/String");
#pragma convert (0)
```

La primera sentencia pragma, con el número 819, informa al compilador que debe almacenar todas las series entrecomilladas posteriores (series literales) en formato ASCII. La segunda sentencia pragma, con el número 0, indica al compilador que para las series entrecomilladas debe volver a la página de códigos

por omisión del compilador, que es normalmente la página de códigos EBCDIC 37. Así, incluyendo la llamada entre sentencias pragma, se cumple el requisito de JNI de que todos los parámetros estén codificados en UTF-8.

Atención: tenga cuidado con las sustituciones de texto. Por ejemplo, si el código es:

```
#pragma convert (819)
#define MyString "java/lang/String"
#pragma convert (0)
myClass = (*env)->FindClass(env,MyString);
```

La serie resultante es EBCDIC porque durante la compilación se sustituye el valor de MyString en la llamada a FindClass. En el momento de producirse esta sustitución, la sentencia pragma número 819 no ha entrado en vigor. Por tanto, las series literales no se almacenan en ASCII.

Convertir series dinámicas a y desde EBCDIC, Unicode y UTF-8: Para manipular variables de tipo serie calculadas en tiempo de ejecución, puede ser necesario convertir las series a, o desde, EBCDIC, Unicode y UTF-8.

La API del sistema que proporciona la función de conversión de página de códigos es iconv(). Para utilizar iconv(), siga estos pasos:

1. Cree un descriptor de conversión con `QtqIconvOpen()`.
2. Llame a `iconv()` para que utilice el descriptor con el fin de convertir en una serie.
3. Cierre el descriptor mediante `iconv_close`.

En el ejemplo 3 de la utilización de Java Native Interface para ejemplos de métodos nativos, la rutina crea, utiliza y a continuación destruye el descriptor de conversión iconv dentro de la rutina. Esta estrategia evita los problemas que plantea la utilización multihebra del descriptor iconv_t, pero en el caso de código sensible al rendimiento es mejor crear un descriptor de conversión en almacenamiento estático y moderar el acceso múltiple a él utilizando una exclusión mutua (mutex) u otro recurso de sincronización.

Métodos nativos IBM OS/400 PASE para Java

La máquina virtual Java (JVM) de iSeries ahora admite el uso de métodos nativos en ejecución en el entorno OS/400 PASE. Antes de la versión V5R2, la JVM de iSeries nativa sólo empleaba métodos nativos ILE. El soporte para métodos nativos OS/400 PASE incluye:

- Pleno uso de la interfaz nativa Java (JNI) de iSeries nativa desde métodos nativos OS/400 PASE
- Posibilidad de llamar a métodos nativos OS/400 PASE desde la JVM de iSeries nativa

Este nuevo soporte permite transportar fácilmente las aplicaciones Java que se ejecutan en AIX al servidor iSeries. Puede copiar los archivos de clase y las bibliotecas de métodos nativos de AIX en el sistema de archivos integrado del iSeries y ejecutarlos desde cualquier indicador de mandatos de CL (Control Language), Qshell o sesión de terminal OS/400 PASE.

Para obtener más información sobre cómo emplear los métodos nativos IBM OS/400 PASE para Java, consulte los temas siguientes:

Variables de entorno Java de OS/400 PASE
Obtenga información sobre las variables de entorno que debe definir antes de utilizar los métodos nativos OS/400 PASE. Estas variables de entorno gestionan los entornos de ejecución OS/400 PASE y JVM.

Códigos de error Java de OS/400 PASE
Para ayudarle a resolver los problemas relacionados con los métodos nativos OS/400 PASE, obtenga información sobre las condiciones de error que indican los mensajes de las anotaciones de trabajo de OS/400 y las excepciones de ejecución Java.

Gestionar bibliotecas de métodos nativos
Descubra los convenios de denominación de bibliotecas Java y el algoritmo de búsqueda de bibliotecas. Esta información es importante para gestionar varias versiones de una biblioteca de método nativo en el servidor iSeries.

Ejemplo: métodos nativos IBM OS/400 PASE para Java
Vea cómo ejecutar un sencillo programa Java que imprime el contenido de una serie Java. En lugar de acceder a la serie directamente desde el código Java, el ejemplo llama a un método nativo que, a continuación, llama de nuevo a Java mediante JNI para obtener el valor de la serie.

Esta información supone que se está familiarizado con OS/400 PASE. Para obtener más información, consulte el siguiente tema:

OS/400 PASE

Variables de entorno Java de OS/400 PASE

La máquina virtual Java (JVM) utiliza las siguientes variables para iniciar entornos OS/400 PASE. Debe establecer la variable QIBM_JAVA_PASE_STARTUP para ejecutar el ejemplo de método nativo IBM OS/400 PASE para Java.

Para obtener información sobre cómo establecer variables de entorno para el ejemplo, consulte el tema siguiente:

Variables de entorno para el ejemplo de IBM OS/400 PASE

QIBM_JAVA_PASE_STARTUP
Debe establecer esta variable de entorno si se cumplen las dos condiciones siguientes:

- Se utilizan los métodos nativos OS/400 PASE
- Se inicia Java desde un indicador de mandatos de iSeries o Qshell

La JVM utiliza esta variable de entorno para iniciar un entorno PASE. El valor de la variable identifica un programa de arranque de OS/400 PASE. El servidor iSeries incluye dos programas de arranque de OS/400 PASE:

- /usr/lib/start32: inicia un entorno de OS/400 PASE de 32 bits
- /usr/lib/start64: inicia un entorno de OS/400 PASE de 64 bits

El formato de bits de todos los objetos de biblioteca compartida empleados por un entorno OS/400 PASE debe coincidir con el formato de bits del entorno OS/400 PASE.

No puede utilizar esta variable al iniciar Java desde una sesión de terminal OS/400 PASE. Una sesión de terminal OS/400 PASE siempre utiliza un entorno OS/400 PASE de 32 bits. Las JVM iniciadas desde una sesión de terminal OS/400 PASE utilizan el mismo tipo de entorno PASE que la sesión de terminal.

QIBM_JAVA_PASE_CHILD_STARTUP
Establezca esta variable de entorno opcional si el entorno OS/400 PASE de una JVM secundaria debe ser diferente del entorno OS/400 PASE de la JVM primaria. Una llamada a Runtime.exec() en Java inicia una JVM secundaria (o hija).

Para obtener más información, consulte Utilizar QIBM_JAVA_PASE_CHILD_STARTUP.

QIBM_JAVA_PASE_ALLOW_PREV
Establezca esta variable de entorno opcional cuando desee utilizar el entorno OS/400 PASE actual, si ya existe uno. En ciertas situaciones, resulta difícil determinar si ya existe un entorno OS/400 PASE. Utilizar QIBM_JAVA_PASE_ALLOW_PREV y QIBM_JAVA_PASE_STARTUP en combinación permite a la JVM utilizar un OS/400 PASE ya existente o iniciar un nuevo entorno OS/400 PASE.

Para obtener más información, consulte Utilizar QIBM_JAVA_PASE_ALLOW_PREV.

Ejemplos: variables de entorno para el ejemplo de IBM OS/400 PASE: Para emplear el ejemplo de métodos nativos IBM OS/400 PASE para Java, debe establecer las siguientes variables de entorno.

PASE_LIBPATH

El servidor iSeries utiliza esta variable de entorno de OS/400 PASE para identificar la ubicación de las bibliotecas de métodos nativos OS/400 PASE. Puede establecer la vía de acceso en un único directorio o varios directorios. En el caso de especificar varios directorios, utilice un carácter de dos puntos (:) para separar las entradas. El servidor también puede emplear la variable de entorno LIBPATH.

Para obtener más información sobre cómo utilizar Java, las bibliotecas de métodos nativos y PASE_LIBPATH con este ejemplo, consulte el tema siguiente:

Utilizar Java, OS/400 PASE y bibliotecas de métodos nativos

PASE_THREAD_ATTACH

Al establecer esta variable de entorno de OS/400 PASE en Y se hace que una hebra ILE no iniciada por OS/400 PASE se conecte automáticamente a OS/400 PASE al llamar a un procedimiento de OS/400 PASE.

Para obtener más información sobre las variables de entorno de OS/400 PASE, consulte las entradas adecuadas del tema siguiente:

Trabajar con las variables de entorno de OS/400 PASE

QIBM_JAVA_PASE_STARTUP

La JVM utiliza esta variable de entorno para iniciar un entorno OS/400 PASE. El valor de la variable identifica un programa de arranque de OS/400 PASE.

Para obtener más información, consulte el siguiente tema:

Variables Java de OS/400 PASE

Utilizar QIBM_JAVA_PASE_CHILD_STARTUP: La variable de entorno QIBM_JAVA_PASE_CHILD_STARTUP indica el programa de arranque de OS/400 PASE para las JVM secundarias. Utilice QIBM_JAVA_PASE_CHILD_STARTUP cuando se cumplan todas las condiciones siguientes:

- La aplicación Java que desea ejecutar crea máquinas virtuales Java (JVM) mediante llamadas Java a Runtime.exec().
- Las JVM primaria y secundaria utilizan métodos nativos OS/400 PASE.
- El entorno OS/400 PASE de las JVM secundarias debe ser distinto del entorno OS/400 PASE de la JVM primaria.

Si todas las condiciones indicadas anteriormente son ciertas, lleve a cabo las acciones siguientes:

- Establezca la variable de entorno QIBM_JAVA_PASE_CHILD_STARTUP en el programa de arranque de OS/400 PASE de las JVM secundarias.
- Al iniciar la JVM primaria desde un indicador de mandatos de iSeries o Qshell, establezca la variable de entorno QIBM_JAVA_PASE_STARTUP en el programa de arranque de OS/400 PASE de la JVM primaria.

  Nota: al iniciar la JVM primaria desde una sesión de terminal OS/400 PASE, no establezca QIBM_JAVA_PASE_STARTUP.

El proceso de la JVM secundaria hereda la variable de entorno QIBM_JAVA_PASE_CHILD_STARTUP. Además, OS/400 establece la variable de entorno QIBM_JAVA_PASE_STARTUP del proceso de la JVM secundaria en el valor de la variable de entorno QIBM_JAVA_PASE_CHILD_STARTUP del proceso padre.

La tabla siguiente identifica los entornos OS/400 PASE resultantes (si existen) para las diversas combinaciones de definiciones y entornos de mandatos de QIBM_JAVA_PASE_STARTUP y QIBM_JAVA_PASE_CHILD_STARTUP:

Entorno de inicio			**Comportamiento resultante**	
Entorno de mandatos	**QIBM_JAVA _PASE_STARTUP**	**QIBM_JAVA_PASE _CHILD_STARTUP**	**Arranque de OS/400 PASE de JVM primaria**	**Arranque de OS/400 PASE de JVM secundaria**
CL o QSH	startX definido	startY definido	Utilizar startX	Utilizar startY
CL o QSH	startX definido	No definido	Utilizar startX	Utilizar startX
CL o QSH	No definido	startY definido	Ningún entorno OS/400 PASE	Utilizar startY
CL o QSH	No definido	No definido	Ningún entorno OS/400 PASE	Ningún entorno OS/400 PASE
Sesión de terminal OS/400 PASE	startX definido	startY definido	No permitido*	No permitido*
Sesión de terminal OS/400 PASE	startX definido	No definido	No permitido*	No permitido*
Sesión de terminal OS/400 PASE	No definido	startY definido	Utilizar entorno de sesión de terminal OS/400 PASE	Utilizar startY
Sesión de terminal OS/400 PASE	No definido	No definido	Utilizar entorno de sesión de terminal OS/400 PASE	Ningún entorno OS/400 PASE

* Las filas marcadas como No permitido indican situaciones en las que la variable de entorno QIBM_JAVA_PASE_STARTUP podría entrar en conflicto con la sesión de terminal OS/400 PASE. Debido al posible conflicto, el uso de QIBM_JAVA_PASE_STARTUP no está permitido desde una sesión de terminal OS/400 PASE.

Utilizar QIBM_JAVA_PASE_ALLOW_PREV: Establezca esta variable de entorno opcional cuando desee utilizar el entorno OS/400 PASE actual, si ya existe uno.

A veces resulta difícil determinar si ya existe un entorno OS/400 PASE. Utilizar QIBM_JAVA_PASE_ALLOW_PREV en combinación con QIBM_JAVA_PASE_STARTUP permite a la JVM determinar si debe utilizarse el entorno OS/400 PASE actual (si existe uno) o iniciar un nuevo entorno OS/400 PASE. Para utilizar estas dos variables de entorno en combinación, establézcalas con los siguientes valores:

- Establezca QIBM_JAVA_PASE_STARTUP en el programa de arranque por omisión
- Establezca QIBM_JAVA_PASE_ALLOW_PREV en 1

Por ejemplo, una aplicación que inicie un entorno OS/400 PASE opcionalmente, llamará al programa que inicia la JVM. En este caso, al utilizar los valores anteriores, el programa puede utilizar el entorno OS/400 PASE actual, si existe uno, o iniciar un nuevo entorno OS/400 PASE.

La siguiente tabla identifica los entornos OS/400 PASE que sean resultado de las diversas combinaciones de entorno OS/400 PASE y definiciones de QIBM_JAVA_PASE_STARTUP y QIBM_JAVA_PASE_ALLOW_PREV:

Entorno de inicio			**Comportamiento resultante**
Entorno OS/400 PASE	**QIBM_JAVA _PASE_STARTUP**	**QIBM_JAVA_PASE _ALLOW_PREV**	**Arranque de OS/400 PASE de JVM**
Ninguno	No definido	No definido*	Ningún entorno OS/400 PASE
Ninguno	No definido	Definido '1'	Ningún entorno OS/400 PASE
Ninguno	startX definido	No definido*	Utilizar startX
Ninguno	startX definido	Definido '1'	Utilizar startX
Iniciado	No definido	No definido*	Utilizar entorno OS/400 PASE existente
Iniciado	No definido	Definido '1'	Utilizar entorno OS/400 PASE existente
Iniciado	startX definido	No definido*	No permitido: error de JVM durante el arranque
Iniciado	startX definido	Definido '1'	Utilizar entorno OS/400 PASE existente

* No definido significa que no se ha incluido QIBM_JAVA_PASE_ALLOW_PREV o que tiene un valor que no es 1.

Las dos últimas filas de la tabla anterior indican situaciones en las que es de utilidad establecer QIBM_JAVA_PASE_ALLOW_PREV. La JVM comprueba QIBM_JAVA_PASE_ALLOW_PREV cuando ya existe un entorno OS/400 PASE y se ha definido QIBM_JAVA_PASE_STARTUP. De lo contrario, la JVM ignora QIBM_JAVA_PASE_ALLOW_PREV.

Las variables de entorno QIBM_JAVA_PASE_ALLOW_PREV y QIBM_JAVA_PASE_CHILD_STARTUP son independientes la una de la otra.

Códigos de error Java de OS/400 PASE

En la lista siguiente se describen los errores con que puede encontrarse en el arranque o la ejecución al utilizar los métodos nativos OS/400 PASE para Java.

Errores de arranque:

Para los errores de arranque, examine los mensajes de las anotaciones de trabajo correspondientes.

Errores de ejecución: Además de los errores de arranque, puede que aparezcan las excepciones Java PaseInternalError o PaseExit en la salida de Qshell de la JVM:

- PaseInternalError - Indica un error interno del sistema. Compruebe las entradas de las anotaciones del código interno bajo licencia.

  Para obtener más información sobre el código de error PaseInternalError, consulte Qp2CallPase.

- PaseExit - La aplicación OS/400 PASE ha llamado a la función exit() o el entorno OS/400 PASE ha finalizado de forma anormal. Compruebe las anotaciones de trabajo y las anotaciones del código interno bajo licencia para obtener más información.

Gestionar bibliotecas de métodos nativos

Para emplear bibliotecas de métodos nativos, especialmente si desea gestionar varias versiones de una biblioteca de método nativo en el servidor iSeries, debe entender tanto los convenios de denominación de bibliotecas Java como el algoritmo de búsqueda de bibliotecas.

OS/400 utiliza la primera biblioteca de método nativo que coincide con el nombre de la biblioteca que carga la máquina virtual Java (JVM). Para asegurarse de que OS/400 encuentra los métodos nativos correctos, debe evitar los conflictos de nombres de biblioteca y toda confusión acerca de qué biblioteca de método nativo utiliza la JVM.

Convenios de denominación de bibliotecas Java de OS/400 PASE y AIX: Si el código Java carga una biblioteca denominada Sample, el archivo ejecutable correspondiente debe denominarse libSample.a o libSample.so.

Orden de búsqueda de bibliotecas Java: Cuando se habilitan los métodos nativos OS/400 PASE para la JVM, el servidor utiliza tres listas distintas (en el orden siguiente) para crear una única vía de acceso de búsqueda de bibliotecas de métodos nativos:

1. Lista de bibliotecas de OS/400
2. Variable de entorno LIBPATH
3. Variable de entorno PASE_LIBPATH

Para efectuar la búsqueda, OS/400 convierte la lista de bibliotecas al formato del sistema de archivos integrado. Los objetos del sistema de archivos QSYS tienen nombres equivalentes en el sistema de archivos integrado, pero algunos objetos del sistema de archivos integrado no tienen nombres del sistema de archivos QSYS equivalentes. Como el cargador de bibliotecas busca los objetos tanto en el sistema de archivos QSYS como el sistema de archivos integrado, OS/400 utiliza el formato del sistema de archivos integrado para buscar las bibliotecas de métodos nativos.

La tabla siguiente muestra cómo OS/400 convierte las entradas de la lista de bibliotecas al formato del sistema de archivos integrado:

Entrada de lista de bibliotecas	Formato del sistema de archivos integrado
QSYS	/qsys.lib
QSYS2	/qsys.lib/qsys2.lib
QGPL	/qsys.lib/qgpl.lib
QTEMP	/qsys.lib/qtemp.lib

Ejemplo: buscar la biblioteca Sample2

En el ejemplo siguiente, LIBPATH se establece en /home/user1/lib32:/samples/lib32 y PASE_LIBPATH se establece en /QOpenSys/samples/lib.

La tabla siguiente, cuando se lee de principio a fin, indica la vía de acceso de búsqueda completa:

Origen	Directorios del sistema de archivos integrado
Lista de bibliotecas	/qsys.lib /qsys.lib/qsys2.lib /qsys.lib/qgpl.lib /qsys.lib/qtemp.lib

Origen	Directorios del sistema de archivos integrado
LIBPATH	/home/user1/lib32 /samples/lib32
PASE_LIBPATH	/QOpenSys/samples/lib

Nota: los caracteres en mayúsculas y minúsculas sólo son significativos en la vía de acceso /QOpenSys.

Para buscar la biblioteca Sample2, el cargador de bibliotecas Java busca los candidatos de archivos en el orden siguiente:

1. /qsys.lib/sample2.srvpgm
2. /qsys.lib/libSample2.a
3. /qsys.lib/libSample2.so
1. /qsys.lib/qsys2.lib/sample2.srvpgm
2. /qsys.lib/qsys2.lib/libSample2.a
3. /qsys.lib/qsys2.lib/libSample2.so
1. /qsys.lib/qgpl.lib/sample2.srvpgm
2. /qsys.lib/qgpl.lib/libSample2.a
3. /qsys.lib/qgpl.lib/libSample2.so
1. /qsys.lib/qtemp.lib/sample2.srvpgm
2. /qsys.lib/qtemp.lib/libSample2.a
3. /qsys.lib/qtemp.lib/libSample2.so
1. /home/user1/lib32/sample2.srvpgm
2. /home/user1/lib32/libSample2.a
3. /home/user1/lib32/libSample2.so
1. /samples/lib32/sample2.srvpgm
2. /samples/lib32/libSample2.a
3. /samples/lib32/libSample2.so
1. /QOpenSys/samples/lib/SAMPLE2.srvpgm
2. /QOpenSys/samples/lib/libSample2.a
3. /QOpenSys/samples/lib/libSample2.so

OS/400 carga el primer candidato de la lista que existe realmente en la JVM como una biblioteca de método nativo. Aunque en la búsqueda se encuentren candidatos como '/qsys.lib/libSample2.a' y '/qsys.lib/libSample2.so', no es posible crear archivos del sistema de archivos integrado o enlaces simbólicos en los directorios /qsys.lib. Por consiguiente, aunque OS/400 busque estos archivos candidatos, nunca los encontrará en los directorios del sistema de archivos integrado que empiezan por /qsys.lib.

Sin embargo, puede crear enlaces simbólicos arbitrarios de otros directorios del sistema de archivos integrado a objetos OS/400 del sistema de archivos QSYS. En consecuencia, entre los candidatos de archivos válidos se encuentran archivos como /home/user1/lib32/sample2.srvpgm.

Métodos nativos del modelo de almacenamiento en teraespacio para Java

La máquina virtual Java (JVM) ahora da soporte al uso de los métodos nativos del modelo de almacenamiento de teraespacio. El modelo de almacenamiento de teraespacio proporciona un entorno de

procesos extensos con dirección local para programas ILE. Utilizar el teraespacio le permite llevar código de método nativo desde otros sistemas operativos a OS/400 con pocos o ningún cambio en el código fuente.

Para conocer detalles sobre la programación con el modelo de almacenamiento en teraespacio, consulte la siguiente información:

Capítulo 4 de ILE Concepts

Capítulo 17 de WebSphere Development Studio ILE C/C++ Programmer's Guide

El concepto para los métodos nativos Java creados para el modelo de almacenamiento en teraespacio es muy similar al de los métodos nativos que utilizan almacenamiento de un solo nivel. La JVM pasa a los métodos nativos de teraespacio un puntero que señala al entorno Java Native Interface (JNI) que los métodos puede utilizar para llamar a funciones JNI.

Para los métodos nativos del modelo de almacenamiento en teraespacio, la JVM proporciona implementaciones de funciones JNI que utilizan el modelo de almacenamiento en teraespacio y punteros de 8 bytes.

Crear métodos nativos de teraespacio

Para crear satisfactoriamente un método nativo del modelo de almacenamiento en teraespacio, el mandato de creación del módulo de teraespacio debe utilizar las siguientes opciones:

```
TERASPACE(*YES) STGMDL(*TERASPACE) DTAMDL(*LLP64)
```

La siguiente opción (`*TSIFC`), para utilizar funciones de almacenamiento en teraespacio, es opcional:

```
TERASPACE(*YES *TSIFC)
```

Nota: Cuando no utilice `DTAMDL(*LLP64)` al utilizar métodos nativos Java del modelo de almacenamiento en teraespacio, llamar a un método nativo generará una excepción de ejecución.

Crear programas de servicio de teraespacio que utilicen métodos nativos

Para poder crear un programa de servicio del modelo de almacenamiento en teraespacio, utilice la siguiente opción en el mandato de lenguaje de control (CL) Crear programa de servicio (CRTSRVPGM):

```
CRTSRVPGM STGMDL(*TERASPACE)
```

Además, es altamente recomendable utilizar la opción `ACTGRP(*CALLER)`, que permite a la JVM activar todos los programas de servicio de métodos nativos del modelo de almacenamiento en teraespacio en el mismo grupo de activación de teraespacio. Utilizar un grupo de activación de teraespacio de este modo puede ser de importancia para que los métodos nativos puedan manejar las excepciones de forma eficaz.

Para obtener detalles adicionales sobre la activación de programas y los grupos de activación, consulte la siguiente información:

Capítulo 3 de ILE Concepts

Utilizar las API de invocación de Java con métodos nativos de teraespacio

Utilice la función GetEnv de la API de invocación cuando el puntero del entorno JNI no coincida con el modelo de almacenamiento del programa de servicio. La función GetEnv de la API de invocación siempre devuelve el puntero de entorno JNI correcto. Para obtener más información, consulte las siguientes páginas:

La API de invocación Java

Mejoras de JNI

La JVM da soporte a métodos nativos del modelo de almacenamiento en teraespacio y de un solo nivel, pero los dos modelos de almacenamiento utilizan entornos JNI distintos. Dado que los dos modelos de almacenamiento utilizan entornos JNI distintos, no pase el puntero de entorno JNI como un parámetro entre métodos nativos en los dos modelos de almacenamiento.

Comparación entre el entorno de lenguajes integrados y Java

En un servidor iSeries, el entorno Java está separado del entorno de lenguajes integrados (ILE). Java no es un lenguaje ILE y no puede enlazarse con los módulos de objeto ILE para crear programas o programas de servicio en un servidor iSeries.

ILE	Java
Los miembros que forman parte de la biblioteca o de la estructura de archivos de un servidor iSeries almacenan los archivos de código fuente.	Los archivos continuos del sistema de archivos integrado contienen el código fuente.
El programa de utilidad para entrada del fuente (SEU) edita archivos fuente EBCDIC.	Los archivos fuente ASCII (American Standard Code for Information Interchange) se editan normalmente con un editor de estación de trabajo.
El resultado de la compilación de archivos fuente son módulos de código objeto, que se almacenan en bibliotecas de un servidor iSeries.	El resultado de la compilación del código fuente son archivos de clase, que se almacenan en el sistema de archivos integrado.
Los módulos objeto están unidos estáticamente entre sí en programas o programas de servicio.	Las clases se cargan dinámicamente según convenga en tiempo de ejecución.
Se puede llamar directamente a funciones escritas en otros lenguajes de programación ILE.	Para llamar a otros lenguajes desde Java, se debe utilizar la interfaz Java nativa (JNI).
Los lenguajes ILE se compilan y ejecutan siempre en forma de instrucciones de lenguaje máquina.	Los programas Java pueden ser interpretados o compilados.

Utilizar java.lang.Runtime.exec()

Utilice el método java.lang.Runtime.exec para llamar a programas o mandatos desde dentro de un programa Java. Al utilizar el método java.lang.Runtime.exec() se crean uno o varios trabajos adicionales habilitados para hebras. Los trabajos adicionales procesan la serie de mandatos que se pasa en el método.

Nota: El método java.lang.Runtime.exec ejecuta los programas en un trabajo aparte, que es distinto a la función C system(). La función C system ejecuta programas en el mismo trabajo.

El proceso real que se produce depende de los siguientes elementos:

- La clase de mandato que pase en java.lang.Runtime.exec()
- El valor de la propiedad del sistema os400.runtime.exec

Procesar distintos tipos de mandatos

La tabla siguiente indica cómo java.lang.Runtime.exec() procesa distintas clases de mandatos y muestra los efectos de la propiedad del sistema os400.runtime.exec.

	Valor de la propiedad del sistema os400.runtime.exec	
Tipo de mandato	**EXEC (valor por omisión)**	**QSHELL**
mandato java	Inicia un segundo trabajo que ejecuta la JVM. La JVM inicia un tercer trabajo que ejecuta la aplicación Java.	Inicia un segundo trabajo que ejecuta Qshell, el intérprete de shell. Qshell inicia un tercer trabajo para ejecutar la aplicación Java, el programa o el mandato.
programa	Inicia un segundo trabajo que ejecuta un programa ejecutable (programa OS/400 o programa OS/400 PASE).	
mandato CL	Inicia un segundo trabajo que ejecuta un programa OS/400. El programa OS/400 ejecuta el mandato CL en el segundo trabajo.	

Nota: Al llamar a un mandato CL o un programa CL, asegúrese de que el CCSID del trabajo contiene los caracteres que pasa como parámetros al mandato llamado.

El proceso del segundo o del tercer trabajo se ejecuta concurrentemente con cualquier máquina virtual Java (JVM) del trabajo original. Cualquier proceso de salida o conclusión que tenga lugar en dichos trabajos no afecta a la JVM original.

propiedad del sistema os400.runtime.exec

Puede establecer el valor de la propiedad del sistema os400.runtime.exec como EXEC (el valor por omisión) o QSHELL. El valor de os400.runtime.exec determina si java.lang.Runtime.exec() utiliza la interfaz EXEC o Qshell.

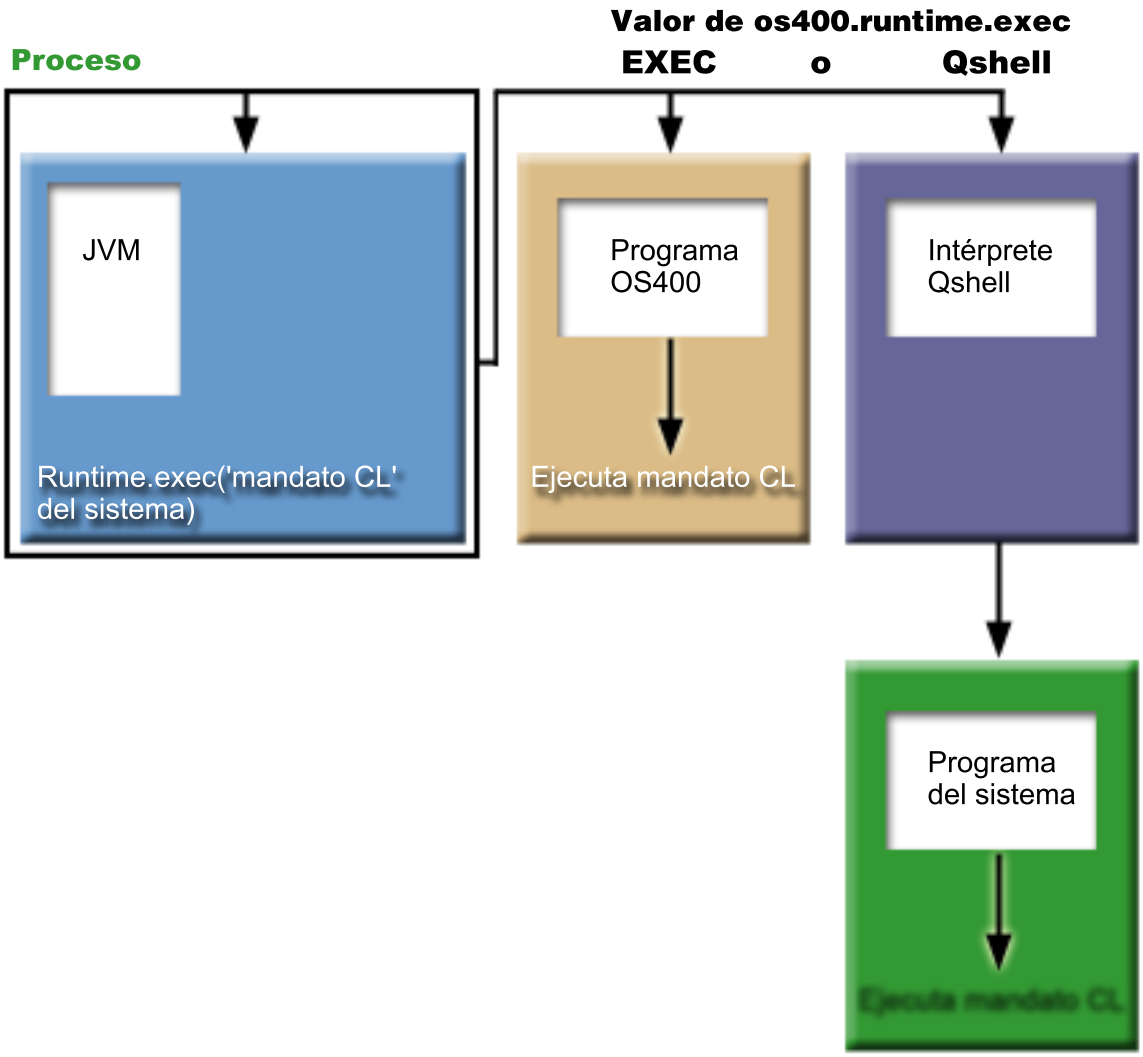
Utilizar un valor de EXEC en lugar de QSHELL tiene las siguientes ventajas:

- El programa Java que llama a java.lang.Runtime.exec() es más portable
- Utilizar java.lang.Runtime.exec() para llamar a un mandato CL emplea menos recursos del sistema

Deberá utilizar java.lang.Runtime.exec() para ejecutar Qshell solamente cuando lo requiera la compatibilidad a la inversa. Utilizar java.lang.Runtime.exec() para ejecutar Qshell requiere que establezca os400.runtime.exec como QSHELL.

La siguiente ilustración muestra cómo utilizar un valor de QSHELL lanza un tercer trabajo, que consume recursos del sistema adicionales. Recuerde que utilizar un valor de QSHELL disminuye la portabilidad del programa Java.

Figura 1. Utilizar un valor de QSHELL para la propiedad del sistema os400.runtime.exec



Además, al utilizar un valor de QSHELL, pasar un mandato CL a java.lang.Runtime.exec() requiere una sintaxis específica. Para obtener más información, consulte el ejemplo siguiente para llamar a un mandato CL (al final de esta página).

Para obtener información sobre cómo establecer os400.runtime.exec, consulte la Lista de propiedades del sistema Java.

Ejemplos: Llamar a mandatos con java.lang.Runtime.exec()

Para conocer maneras de utilizar java.lang.Runtime.exec() para ejecutar distintos tipos de mandatos, vea los siguientes ejemplos:

- Ejemplo: Llamar a otro programa Java con java.lang.Runtime.exec()
- Ejemplo: Llamar a un programa CL con java.lang.Runtime.exec()
- Ejemplo: Llamar a un mandato CL con java.lang.Runtime.exec()

Comunicaciones entre procesos

A la hora de comunicar con programas que se ejecutan en otro proceso, existen diversas opciones.

Una opción es utilizar sockets para la comunicación entre procesos. Un programa puede actuar a modo de programa servidor, a la escucha de una entrada procedente del programa cliente en una conexión por socket. El programa cliente se conecta al servidor por medio de un socket. Una vez establecida la conexión por socket, cualquiera de los dos programas puede enviar o recibir información.

Otra opción es utilizar archivos continuos para la comunicación entre programas. Para ello, se utilizan las clases System.in, System.out y System.err.

Una tercera opción es utilizar IBM Toolbox para Java, que proporciona colas de datos y objetos de mensaje de iSeries.

También puede llamar a Java desde otros lenguajes. Consulte el Ejemplo: Llamar a Java desde C y el Ejemplo: Llamar a Java desde RPG para obtener más información.

Utilizar sockets para la comunicación entre procesos

Las corrientes por sockets comunican entre sí programas que se están ejecutando en procesos aparte. Los programas pueden iniciarse independientemente o mediante el método java.lang.Runtime.exec() desde el programa principal (main) de Java. Si un programa está escrito en un lenguaje distinto de Java, hay que asegurarse de que tiene lugar la conversión ASCII (American Standard Code for Information Interchange) o EBCDIC (extended binary-coded decimal interchange code). En Codificaciones de caracteres Java hallará información más detallada.

Para obtener un ejemplo de utilización de sockets, consulte el Ejemplo: utilizar sockets para la comunicación entre procesos.

Utilizar corrientes de entrada y de salida para la comunicación entre procesos

Las corrientes de entrada y de salida comunican entre sí programas que se ejecutan en procesos aparte. El método java.lang.Runtime.exec() ejecuta un programa. El programa padre puede obtener handles para acceder a las corrientes de entrada y de salida del proceso hijo y escribir o leer en ellas. Si el programa hijo está escrito en un lenguaje distinto de Java, es preciso asegurarse de que tiene lugar la conversión ASCII (American Standard Code for Information Interchange) o EBCDIC (Extended Binary-Coded Decimal Interchange Code). En Codificaciones de caracteres Java hallará información más detallada.

Si desea ver un ejemplo que utiliza corrientes de entrada y de salida, consulte el Ejemplo: utilizar corrientes de entrada y de salida para la comunicación entre procesos.

Plataforma Java

La **plataforma Java** es el entorno para desarrollar y gestionar applets y aplicaciones Java. Consta de tres componentes principales: el lenguaje Java, los paquetes Java y la máquina virtual Java. El lenguaje y los paquetes Java son parecidos a C++ y a sus bibliotecas de clases. Los paquetes Java contienen clases, que están disponibles en cualquier implementación compatible con Java. La API (interfaz de programación de aplicaciones) debe ser la misma en cualquier sistema que soporte Java.

Java difiere de un lenguaje tradicional, como por ejemplo C++, en la forma de compilar y ejecutar. En un entorno de programación tradicional, el usuario escribe y compila el código fuente de un programa en código de objeto para un sistema operativo y un hardware específico. El código de objeto se enlaza con otros módulos de código de objeto para crear un programa en ejecución. El código es específico con respecto a un conjunto determinado de hardware de sistema y no funciona en otros sistemas si no se realizan cambios. Esta figura muestra el entorno de despliegue de un lenguaje tradicional.

Para utilizar la plataforma Java de forma eficaz, consulte las siguientes secciones:

Applets y aplicaciones Java
Puede escribir el applet Java e incluirlo en una página HTML, de forma muy parecida a la inclusión de una imagen. Al utilizar un navegador habilitado para Java para ver una página HTML que contiene un applet, el código del applet se transfiere al sistema y la máquina virtual Java del navegador lo ejecuta. También puede escribir una aplicación Java que no requiera la utilización de un navegador Web.

Máquina virtual Java
Puede incorporar la máquina virtual Java a un navegador Web o a un sistema operativo, como por ejemplo IBM Operating System/400 (OS/400). La máquina virtual Java consta del intérprete de Java y del entorno de ejecución Java. El intérprete realiza la tarea de interpretar el archivo de clase y ejecutar las instrucciones Java en una plataforma de hardware determinada. La máquina virtual Java es el componente que permite escribir código Java, compilarlo una sola vez y ejecutarlo en cualquier plataforma.

Archivos JAR y de clase Java
El entorno Java difiere de otros entornos de programación en que el compilador Java no genera código de máquina para un conjunto de instrucciones específicas de hardware. En lugar de ello, el compilador Java convierte el código fuente en instrucciones de la máquina virtual Java, almacenadas por los archivos de clase Java. Puede utilizar archivos JAR para almacenar los archivos de clase. El archivo de clase no está destinado a una plataforma de hardware específica, sino a la arquitectura de la máquina virtual Java.

Hebras Java
Java es un lenguaje de programación multihebra; por ello, más de una hebra puede estar ejecutándose simultáneamente en la máquina virtual Java. Las hebras Java proporcionan un medio de que el programa Java realice varias tareas al mismo tiempo.

Java Development Kit
Java Development Kit (JDK) es un software distribuido por Sun Microsystems, Inc. para los programadores de Java. Incluye el intérprete de Java, las clases Java y herramientas de desarrollo Java. Consulte la siguiente información acerca de JDK:

- Paquetes Java
- Herramientas Java

Applets y aplicaciones Java

Un **applet** es un programa Java diseñado para incluirse en un documento Web HTML. El documento HTML contiene identificadores, que especifican el nombre del applet Java y su URL (Localizador Universal de Recursos). El URL es la ubicación en la que residen los bytecodes del applet en Internet. Cuando se visualiza un documento HTML que contiene un identificador de applet Java, un navegador Web habilitado para Java descarga los bytecodes Java de Internet y utiliza la máquina virtual Java para procesar el código desde el documento Web. Estos applets Java son los que permiten que las páginas Web contengan gráficos animados o información interactiva.

Para obtener más información, consulte Writing Applets

, la guía de aprendizaje de Sun Microsystems para applets Java. Incluye una visión general de los applets, instrucciones para escribir applets y algunos problemas comunes acerca de los applets.

Las **aplicaciones** son programas autónomos que no requieren la utilización de un navegador. Las aplicaciones Java se ejecutan iniciando el intérprete de Java desde la línea de mandatos y especificando el

archivo que contiene la aplicación compilada. Generalmente, las aplicaciones residen en el sistema en el que se despliegan. Las aplicaciones acceden a los recursos del sistema, y están restringidas por el modelo de seguridad de Java.

Máquina virtual Java

La máquina virtual Java es un entorno de ejecución que puede añadirse a un navegador Web o a cualquier sistema operativo, como por ejemplo IBM Operating System/400 (OS/400). La máquina virtual Java ejecuta instrucciones generadas por un compilador Java. Consta de un intérprete de bytecode y un sistema de ejecución que permiten ejecutar los archivos de clase Java (page 182) en cualquier plataforma, independientemente de la plataforma en la que se desarrollaron originariamente.

El cargador de clases y el gestor de seguridad, que forman parte de la ejecución Java, aíslan el código que proviene de otra plataforma. También pueden restringir los recursos del sistema a los que accede cada una de las clases que se cargan.

Nota: las aplicaciones Java no quedan restringidas; sólo los applets lo están. Las aplicaciones pueden acceder libremente a los recursos del sistema y utilizar métodos nativos. La mayoría de los programas de IBM Developer Kit para Java son aplicaciones.

Puede utilizar el mandato Crear programa Java (CRTJVAPGM) para asegurarse de que el código cumple los requisitos de seguridad impuestos por la ejecución Java para verificar los bytecodes. Esto incluye forzar restricciones de tipos, comprobar conversiones de datos, garantizar que no se produzcan desbordamientos o subdesbordamientos de la pila de parámetros y comprobar las violaciones de acceso. Sin embargo, muchas veces no es necesario comprobar explícitamente los bytecodes. Si no utiliza el mandato CRTJVAPGM de antemano, las comprobaciones se producen durante la primera utilización de una clase. Una vez verificados los bytecodes, el intérprete decodifica los bytecodes y ejecuta las instrucciones de máquina necesarias para realizar las operaciones deseadas.

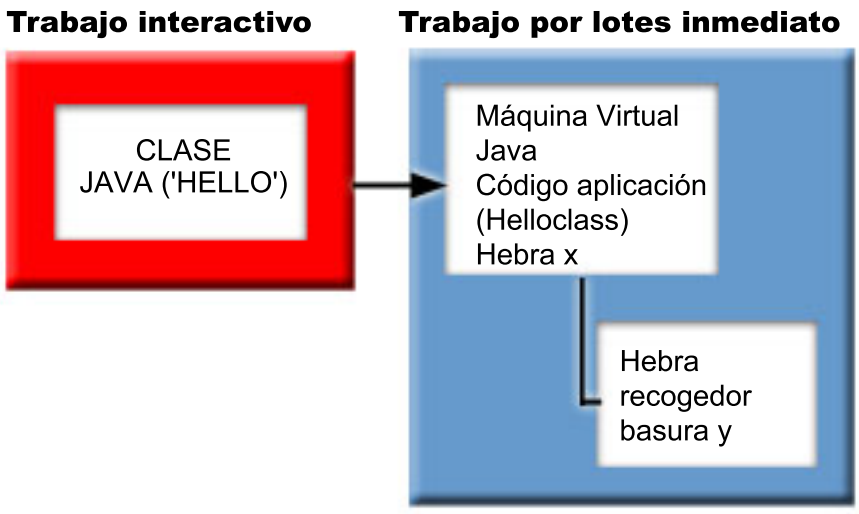
Nota: El “Intérprete de Java” en la página 181 sólo se utiliza si especifica OPTIMIZE(*INTERPRET) o INTERPRET(*YES).

Además de cargar y ejecutar los bytecodes, la máquina virtual Java incluye un recolector de basura que gestiona la memoria. La recogida de basura se ejecuta al mismo tiempo que la carga e interpretación de los bytecodes.

Entorno de ejecución Java

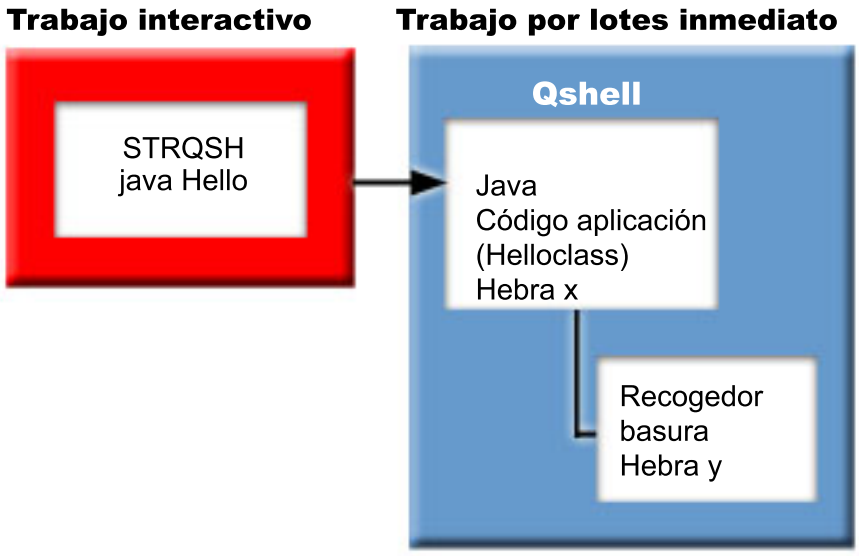
El entorno de ejecución Java se inicia cada vez que se especifica el mandato Ejecutar Java (RUNJVA) o el mandato JAVA en la línea de mandatos de iSeries. Dado que el entorno Java es multihebra, es necesario ejecutar la máquina virtual Java en un trabajo que dé soporte a las hebras, como puede ser un trabajo inmediato de proceso por lotes (BCI). Como se ilustra en la Figura 1, una vez iniciada la máquina virtual Java, pueden iniciarse más hebras en el trabajo en las que se ejecutará el recogedor de basura.

Figura 1: El entorno Java típico al utilizar el mandato CL RUNJVA o JAVA



También es posible iniciar el entorno de ejecución Java con el mandato `java` de Qshell desde el intérprete de Qshell. En este entorno, el intérprete de Qshell se ejecuta en un trabajo BCI asociado a un trabajo interactivo. El entorno de ejecución Java se inicia en el trabajo que está ejecutando el intérprete de Qshell.

Figura 2: El entorno Java al utilizar el mandato java en Qshell



Cuando el entorno de ejecución Java se inicia desde un trabajo interactivo, aparece la pantalla de la shell Java. Esta pantalla proporciona una línea de entrada para incluir datos en la corriente System.in, así como para visualizar los datos que se escriben en la corriente System.out y en la corriente System.err.

Intérprete de Java

El intérprete de Java es el componente de la máquina virtual Java que interpreta los archivos de clase Java para una plataforma de hardware determinada. El intérprete de Java decodifica cada bytecode y ejecuta una serie de instrucciones de máquina para ese bytecode.

Archivos JAR y de clase Java

Un **archivo de archivado (JAR) Java** es un formato de archivo que combina muchos archivos en uno. Puede utilizar los archivos JAR como herramienta de archivado general y también para distribuir programas Java de todos los tipos, incluyendo applets. Los applets Java se bajan en un navegador en una sola transacción HTTP (Hypertext Transfer Protocol) en lugar de abriendo una conexión nueva para cada componente. Este método de bajada aumenta la velocidad con la que el applet se carga en una página Web y empieza a funcionar.

JAR es el único formato de archivado independiente de plataforma. JAR también es el único formato que maneja archivos de audio e imagen, así como archivos de clase. JAR es un formato de estándar abierto, totalmente ampliable, escrito en Java.

El formato JAR también da soporte a la compresión, lo cual reduce el tamaño del archivo y disminuye el tiempo de bajada. Además, el autor de un applet puede firmar digitalmente las entradas individuales de un archivo JAR para autenticar su origen.

Para actualizar las clases de archivos JAR, consulte la Herramienta jar Java.

Los **archivos de clase Java** son archivos continuos que se producen cuando el compilador Java compila un archivo fuente. El archivo de clase contiene tablas que describen cada uno de los campos y métodos de la clase. También contiene los bytecodes de cada método, datos estáticos y descripciones utilizadas para representar objetos Java.

Hebras Java

Una **hebra** es una única corriente independiente que se ejecuta dentro de un programa. Java es un lenguaje de programación multihebra, por lo que puede ejecutarse simultáneamente más de una hebra en la máquina virtual Java. Las hebras Java proporcionan un medio de que el programa Java realice varias tareas al mismo tiempo. Una hebra es esencialmente un flujo de control de un programa.

Las hebras son construcciones de programación moderna que se utilizan para dar soporte a programas concurrentes y para mejorar el rendimiento y la escalabilidad de las aplicaciones. La mayoría de los lenguajes de programación soportan las hebras mediante la utilización de bibliotecas de programación de complementos. Java soporta las hebras como interfaces de programas de aplicación (API) incorporadas.

Nota: la utilización de hebras proporciona el soporte necesario para aumentar la interactividad, consiguiendo con ello reducir la espera en el teclado al ejecutarse más tareas en paralelo. Sin embargo, el programa no es necesariamente más interactivo sólo porque utilice hebras.

Las hebras son el mecanismo de espera en interacciones de larga ejecución, mientras permiten que el programa maneje otras tareas. Las hebras tienen la capacidad de dar soporte a varios flujos mediante la misma corriente de código. A veces se las denomina **procesos ligeros**. El lenguaje Java incluye soporte directo para hebras. Sin embargo, por diseño, no soporta la entrada y salida asíncrona sin bloques con interrupciones ni la espera múltiple.

Las hebras permiten el desarrollo de programas paralelos que se escalan adecuadamente en un entorno en el que una máquina tenga varios procesadores. Si se construyen apropiadamente, también proporcionan un modelo para el manejo de varias transacciones y usuarios.

Puede utilizar las hebras en un programa Java en diversas situaciones. Algunos programas deben ser capaces de sumarse a varias actividades y continuar siendo capaces de responder a la entrada adicional por parte del usuario. Por ejemplo, un navegador Web debe ser capaz de responder a la entrada del usuario mientras reproduce un sonido.

Las hebras también pueden utilizar métodos asíncronos. Cuando el usuario llama a un segundo método, no es necesario que espere a que finalice el primer método para que el segundo continúe con su propia actividad.

Existen también muchas razones para no utilizar hebras. Si un programa utiliza de manera inherente una lógica secuencial, una sola hebra puede realizar la secuencia completa. En este caso, la utilización de varias hebras produce un programa complejo sin ninguna ventaja. La creación e inicio de una hebra conlleva un trabajo considerable. Si una operación sólo implica unas pocas sentencias, resulta más rápido manejarla con una sola hebra. Esto puede ser cierto aún en el caso de que la operación sea conceptualmente asíncrona. Cuando varias hebras comparten objetos, éstos deben sincronizarse para coordinar el acceso de las hebras y conservar la coherencia. La sincronización añade complejidad a un programa, resulta difícil ajustarlo para un rendimiento óptimo y puede ser una fuente de errores de programación.

Para obtener más información acerca de las hebras, consulte la sección Desarrollar aplicaciones multihebra.

Sun Microsystems, Inc. Java Development Kit

Java Development Kit (JDK) es un software distribuido por Sun Microsystems, Inc. destinado a los programadores de Java. Incluye el intérprete de Java, las clases Java y herramientas de desarrollo Java: compilador, depurador, desensamblador, visor de applets, generador de archivo de apéndice y generador de documentación.

JDK permite escribir aplicaciones que se desarrollan una sola vez y se ejecutan en cualquier lugar de cualquier máquina virtual Java. Las aplicaciones Java desarrolladas con JDK en un sistema pueden utilizarse en otro sistema sin cambiar ni recompilar el código. Los archivos de clase Java son portables a cualquier máquina virtual Java estándar.

Para encontrar más información acerca del JDK actual, compruebe la versión de IBM Developer Kit para Java del servidor iSeries.

Puede comprobar la versión de IBM Developer Kit para Java por omisión de la máquina virtual Java del servidor iSeries especificando cualquiera de los siguientes mandatos:

- `java -version` en el indicador de mandatos de Qshell.
- `RUNJVA CLASS(*VERSION)` en la línea de mandatos CL.

A continuación, busque la misma versión de JDK de Sun Microsystems, Inc. en The Source for Java Technology java.sun.com

para obtener documentación específica. IBM Developer Kit para Java es una implementación compatible de Sun Microsystems, Inc. Java Technology, y por tanto debe estar familiarizado con su documentación de JDK.

Consulte los siguientes temas para obtener más información:

- Soporte para varios Java Development Kits (JDK) proporciona información acerca de la utilización de varias máquinas virtuales Java.
- Métodos nativos y la interfaz Java nativa define qué es un método nativo y lo que puede hacer. Este tema también describe brevemente la interfaz Java nativa.

Paquetes Java

Un paquete Java es una forma de agrupar interfaces y clases relacionadas en Java. Los paquetes Java son parecidos a las bibliotecas de clases disponibles en otros lenguajes.

Los paquetes Java, que proporcionan las API Java, están disponibles como parte de Java Development Kit (JDK) de Sun Microsystems. Inc. Para obtener una lista completa de paquetes Java e información sobre las API de Java, consulte Paquetes de Java 2 Platform.

Herramientas Java

Para obtener una lista completa de las herramientas suministradas por Java Development Kit de Sun Microsystems, Inc., consulte la guía de consulta de herramientas de Sun Microsystems, Inc. Para obtener más información acerca de cada una de las herramientas soportadas por IBM Developer Kit para Java, consulte la sección Herramientas Java soportadas por IBM Developer Kit para Java.

Temas avanzados

A continuación se indican los temas avanzados de IBM Developer Kit para Java:

Clases, paquetes y directorios
Cada clase Java forma parte de un paquete. El nombre del paquete está relacionado con la estructura de directorios en la que reside la clase.

Los archivos del sistema de archivos integrado
El sistema de archivos integrado almacena los archivos JAR, los archivos ZIP, los archivos fuente y los archivos de clase relacionados con Java en una estructura jerárquica de archivos.

Autorizaciones de archivo
Para ejecutar o depurar un programa Java, es necesario que el archivo de clase, JAR o ZIP tenga autorización de lectura. Puede obtener más información acerca de las autorizaciones de archivo que requieren diversos mandatos CL.

Trabajo por lotes
Los programas Java pueden ejecutarse en un trabajo por lotes mediante el mandato Someter trabajo (SBMJOB). Puede obtener más información acerca del mandato SBMJOB y acerca de cómo puede verificar que el trabajo por lotes pueda ejecutar más de un trabajo.

Clases, paquetes y directorios Java

Cada clase Java forma parte de un paquete. La primera sentencia de un archivo fuente Java indica la clase y el paquete que la contienen. Si el archivo fuente no contiene la sentencia package, significa que la clase forma parte de un paquete por omisión cuyo nombre no se indica.

El nombre del paquete está relacionado con la estructura de directorios en la que reside la clase. El sistema de archivos integrado da soporte a las clases Java en una estructura jerárquica de archivos parecida a la que existe en la mayoría de los sistemas UNIX y PC. Las clases Java deben almacenarse en un directorio que tenga una vía de acceso relativa que coincida con el nombre de paquete de la clase. Por ejemplo, observe la siguiente clase Java:

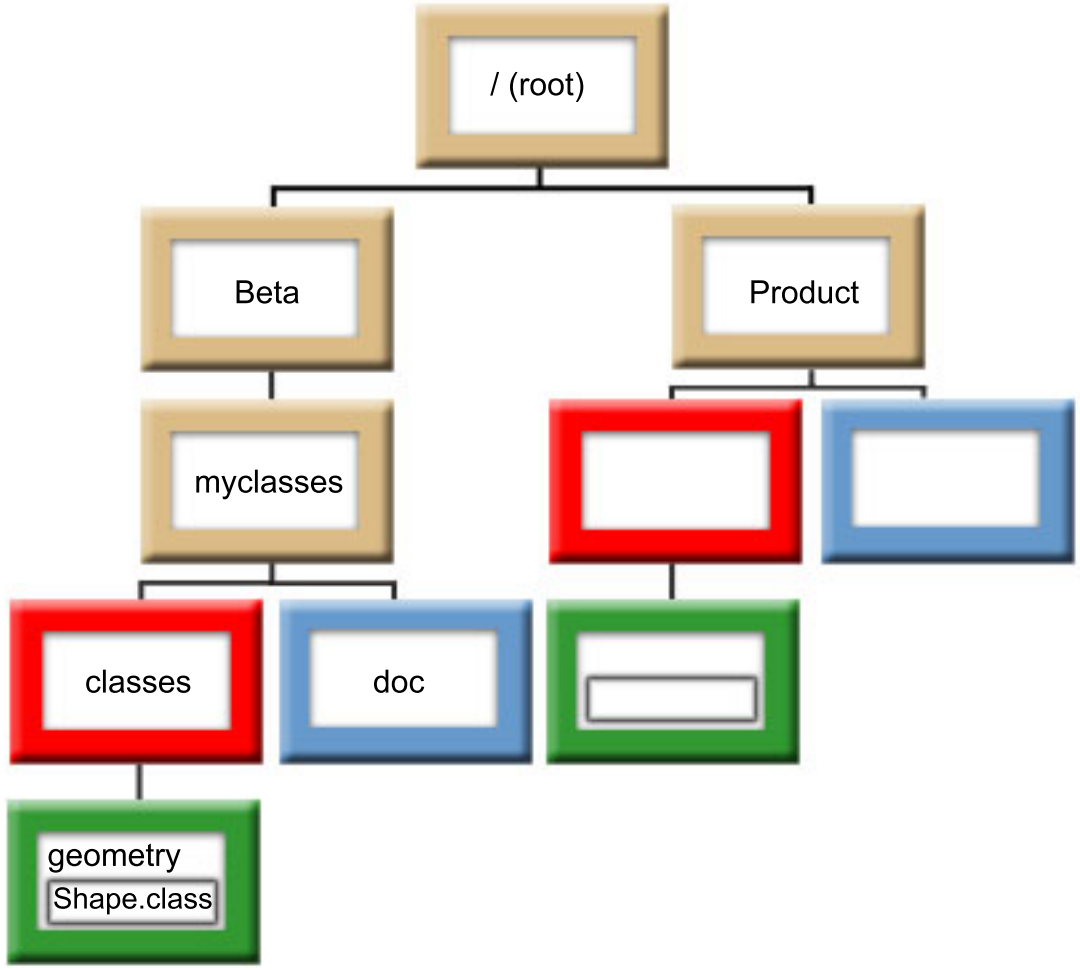
```
     package classes.geometry;
import java.awt.Dimension;

public class Shape {

Dimension metrics;

   // El código de la implementación de la clase Shape estaría aquí ...

     }
```

La sentencia package del código anterior indica que la clase Shape forma parte del paquete classes.geometry. Para que la unidad ejecutable Java encuentre la clase Shape, debe almacenar la clase Shape en la estructura de directorios relativa classes/geometry.

Nota: El nombre de paquete se corresponde con el nombre de directorio relativo en el que reside la clase. El cargador de clases de la máquina virtual Java busca la clase agregando el nombre de vía de acceso relativa a cada uno de los directorios que especifique en la vía de acceso de clases. El cargador de clases de la máquina virtual Java también puede buscar la clase realizando una búsqueda en los archivos ZIP o JAR especificados en la vía de acceso de clases.

Por ejemplo, si almacena la clase Shape en el directorio /Product/classes/geometry del sistema de archivos "root" (/), deberá especificar /Product en la vía de acceso de clases.

Figura 1: Ejemplo de estructura de directorios para clases Java del mismo nombre en paquetes distintos



Nota: en la estructura de directorios pueden existir varias versiones de la clase Shape. Para utilizar la versión Beta de la clase Shape, coloque /Beta/myclasses dentro de la vía de acceso de clases antes de cualquier otro directorio o archivo ZIP que contenga la clase Shape.

El compilador Java utiliza la vía de acceso de clases Java, el nombre de paquete y la estructura de directorios para buscar los paquetes y las clases cuando compila el código fuente Java. Para obtener más información, consulte la vía de acceso de clases Java.

Los archivos del sistema de archivos integrado

IBM Developer Kit para Java da soporte al uso de sistemas de archivos seguros en ejecución multihebra en el sistema de archivos integrado para almacenar y trabajar con los archivos de clase relacionados con Java, los archivos fuente, los archivos ZIP y los archivos JAR. Para obtener más información sobre los sistemas de archivos seguros en ejecución multihebra y una comparación de los sistemas de archivos, consulte lo siguiente:

Consideraciones sobre sistemas de archivos para programación multihebra

Comparación de sistemas de archivos

Autorizaciones de archivo Java en el sistema de archivos integrado

Para ejecutar o depurar un programa Java, es necesario que el archivo de clase, JAR o ZIP tenga autorización de lectura (*R). Los directorios necesitan la autorización de lectura y la de ejecución (*RX).

Para utilizar el mandato Crear programa Java (CRTJVAPGM) con el fin de optimizar un programa, el archivo de clase, el archivo JAR o el archivo ZIP debe tener autorización de lectura (*R) y el directorio debe tener autorización de ejecución (*X). Si utiliza un patrón dentro del nombre de archivo de clase, el directorio debe tener autorización de lectura y de ejecución (*RX).

Para suprimir un programa Java con el mandato Suprimir programa Java (DLTJVAPGM), hay que tener autorización de lectura y escritura (*RW) sobre el archivo de clase, y el directorio debe tener autorización de ejecución (*X). Si utiliza un patrón dentro del nombre de archivo de clase, el directorio debe tener autorización de lectura y de ejecución (*RX).

Para visualizar un programa Java con el mandato Visualizar programa Java (DSPJVAPGM), hay que tener autorización de lectura (*R) sobre el archivo de clase, y el directorio debe tener la autorización de ejecución (*X).

Nota: para un usuario que posea la autorización QSECOFR, siempre parecerá que los archivos y los directorios que carecen de autorización de ejecución (*X) tienen dicha autorización. Distintos usuarios pueden obtener resultados diferentes en determinadas situaciones, aunque aparentemente todos ellos tengan el mismo tipo de acceso a los mismos archivos. Es importante saberlo cuando se ejecutan scripts de shell mediante el intérprete de Qshell o java.Runtime.exec().

Por ejemplo, un usuario escribe un programa Java que utiliza java.Runtime.exec() para llamar a un script de shell y lo prueba utilizando un ID de usuario que tiene la autorización QSECOFR. Si la modalidad de archivo del script de shell tiene autorización de lectura y de escritura (*RW), el sistema de archivos integrado permitirá que lo ejecute el ID de usuario que tiene la autorización QSECOFR. Sin embargo, podría ser que un usuario sin autorización QSECOFR intentase ejecutar el mismo programa Java y que el sistema de archivos integrado le indicase al código de java.Runtime.exec() que el script de shell no puede ejecutarse porque falta *X. En este caso, java.Runtime.exec() lanza una excepción de entrada y salida.

También puede asignar autorizaciones a archivos nuevos creados por programas Java en un sistema de archivos integrado. Utilizando la propiedad del sistema os400.file.create.auth para los archivos y os400.dir.create.auth para los directorios, puede utilizarse cualquier combinación de autorizaciones de lectura, escritura y ejecución.

Para obtener más información, consulte las secciones Las API de programa y mandato CL o Sistema de archivos integrado.

Ejecutar Java en un trabajo de proceso por lotes

Para ejecutar programas Java en un trabajo de proceso por lotes, hay que utilizar el mandato Someter trabajo (SBMJOB). En esta modalidad, la pantalla Entrada de mandato de Qshell de Java no está disponible para manejar las corrientes System.in, System.out y System.err.

Puede redirigir estas corrientes a otros archivos. Por omisión, las corrientes System.out y System.err se envían a un archivo en spool. El trabajo de proceso por lotes, que da como resultado una excepción de entrada y salida para las peticiones de lectura procedentes de System.in, es el propietario del archivo en spool. System.in, System.out y System.err se pueden redirigir dentro del programa Java. Para ello también se pueden utilizar las propiedades os400.stdin, os400.stdout y os400.stderr del sistema.

Nota: SBMJOB establece el directorio de trabajo actual (CWD) en el directorio HOME especificado en el perfil de usuario.

Ejemplo: ejecutar Java en un trabajo de proceso por lotes

```
SBMJOB CMD(JAVA Hello OPTION(*VERBOSE)) CPYENVVAR(*YES)
```

La ejecución del mandato JAVA en el ejemplo anterior genera un segundo trabajo. Por lo tanto, el subsistema en el que se ejecuta el trabajo debe tener capacidad para ejecutar más de un trabajo.

Para verificar que el trabajo de proceso por lotes tiene capacidad para ejecutar más de un trabajo, siga estos pasos:

1. En la línea de mandatos CL, escriba `DSPSBSD(MYSBSD)`, donde `MYSBSD` es la descripción de subsistema del trabajo de proceso por lotes.
2. Elija la opción 6, Entradas de cola de trabajos.
3. Observe el campo Máx. activo correspondiente a la cola de trabajos.

   Si el valor del campo Máx. activo es igual o menor que 1 y no es *NOMAX, entre lo siguiente en la línea de mandatos CL:

   ```
   CHGJOBQE SBSD(MYSBSD) JOBQ(MYJOBQ) MAXACT(*NOMAX)
   ```

   Donde:

   - `MYSBSD` es la descripción de subsistema, y
   - `MYJOBQ` es la cola de trabajos.

Ejecutar la aplicación Java en un sistema principal que no tiene una interfaz gráfica de usuario

Si desea ejecutar la aplicación Java en un sistema principal que no tiene una interfaz gráfica de usuario (GUI), como por ejemplo en un servidor iSeries, puede utilizar NAWT (Native Abstract Window Toolkit).

Utilice NAWT para proporcionar a las aplicaciones y servlets Java la posibilidad de utilizar las funciones de gráficos de AWT de Java 2 Software Development Kit's (J2SDK), Standard Edition. Para obtener más información, consulte el apartado Native Abstract Windowing Toolkit (NAWT)

NAWT (Native Abstract Windowing Toolkit)

Native Abstract Windowing Toolkit (NAWT) proporciona a las aplicaciones y los servlets Java la posibilidad de utilizar la función de gráficos AWT (Abstract Windowing Toolkit) ofrecida por Java 2 Software Development Kit (J2SDK), Standard Edition.

Nota: NAWT no da soporte actualmente a fonts y juegos de caracteres específicos de entornos locales y de idiomas. Al utilizar NAWT, asegúrese de que está en conformidad con los siguientes requisitos:

- Utilice solamente caracteres que estén definidos en el juego de caracteres ISO8859-1.
- Utilice el archivo font.properties. El archivo font.properties reside en el directorio /QIBM/ProdData/Java400/jdknn/lib, donde *nn* es el número de versión del J2SDK que está utilizando. Concretamente, no utilice ninguno de los archivos font.properties.*xxx*, donde *xxx* es un idioma u otro calificador.

Normalmente, NAWT utiliza X Window System como motor de gráficos subyacente. Para utilizar X Window System, necesita un servidor X. Un servidor X es una aplicación autónoma que acepta conexiones y peticiones de programas cliente X. En este caso, la infraestructura NAWT subyacente es el programa cliente X.

El servidor X recomendado es el servidor AT&T Virtual Network Computing (VNC).El servidor VNC se adapta bien a los servidores iSeries ya que no necesita un ratón, un teclado y un monitor con capacidad de gráficos dedicados. IBM proporciona una versión del servidor VNC que se ejecuta en el entorno OS/400 PASE (Portable Application Solutions Environment). OS/400 PASE es un entorno similar a UNIX que le permite ejecutar la mayoría de ejecutables binarios compilados para el sistema operativo IBM AIX. OS/400 PASE se instala como parte de OS/400 Versión 5 Release 2.

Al ejecutar el servidor VNC en OS/400 PASE, el servidor iSeries realiza todos los cálculos de gráficos de NAWT, por lo que no requiere un servidor de gráficos externo. La siguiente información de NAWT y J2SDK describe cómo obtener y poner a punto el servidor VNC en OS/400 PASE.

Para obtener más información sobre la instalación y uso de NAWT, consulte lo siguiente:

Niveles de soporte de NAWT
Lea sobre los niveles de soporte de NAWT disponibles con las distintas versiones de J2SDK. Utilice esta información como ayuda para valorar sus necesidades gráficas y seleccionar la versión de J2SDK que necesita ejecutar.

Instalar y utilizar NAWT
Utilice las instrucciones paso a paso para instalar NAWT y VNC. Aprenda los pasos necesarios que debe completar cada vez que desee utilizar NAWT.

Consejos para la utilización de VNC
Averigüe cómo utilizar mandatos de lenguaje de control (CL) de OS/400 para iniciar y detener un servidor VNC y para visualizar información sobre los servidores VNC en ejecución actualmente.

Consejos para el uso de NAWT con WebSphere Application Server
Aprenda a poner a punto NAWT para que lo utilicen los programas Java de gráficos que se ejecuten bajo WebSphere Application Server.

Dado que ejecutar NAWT requiere utilizar OS/400 PASE y VNC, puede interesarle aprender más sobre estas aplicaciones. Para obtener más información, consulte lo siguiente:

OS/400 PASE

Virtual Network Computing

Niveles de soporte de NAWT

La versión de Java 2 Software Development Kit (J2SDK), Standard Edition que utilice afectará a las opciones disponibles para soporte de Native Abstract Windowing Toolkit (NAWT). Antes de instalar NAWT, es necesario comprender qué tipo de soporte cumple sus necesidades.

NAWT y J2SDK, versión 1.3: Para J2SDK versión 1.3, NAWT da soporte solamente a aplicaciones Java gráficas que no necesitan interacción directa del usuario. Este nivel de soporte es adecuado para aplicaciones Java, servlets y paquetes de gráficos que generan datos de imágenes (codificados como JPEG, GIF y otros) en los servidores iSeries.

NAWT y J2SDK, versión 1.4: Para J2SDK versión 1.4, NAWT da soporte a todas las funciones de Java de Abstract Windowing Toolkit (AWT), incluidas las interfaces gráficas de usuario (GUI) interactivas y el entorno AWT headless Java.

Para obtener más información sobre el soporte de NAWT disponible al ejecutar J2SDK, versión 1.4, consulte lo siguiente:

Soporte completo de GUI

Soporte de AWT headless

Instalar y utilizar Native Abstract Windowing Toolkit

La versión de Java 2 Software Development Kit (J2SDK), Standard Edition que utilice y el nivel de soporte de Native Abstract Windowing Toolkit (NAWT) que necesite afectará a cómo instalará NAWT. Antes de instalar NAWT, es necesario comprender los distintos niveles de soporte gráfico que NAWT ofrece. Para obtener más información, consulte Niveles de soporte de NAWT.

Instalar y utilizar NAWT: Después de valorar sus necesidades gráficas y determinar qué versión de J2SDK desea ejecutar, utilice las siguientes instrucciones para instalar y utilizar NAWT:

J2SDK, versión 1.3

J2SDK, versión 1.4, soporte completo de GUI

J2SDK, versión 1.4, soporte de AWT headless

NAWT y OS/400 PASE: NAWT inicia el entorno OS/400 PASE automáticamente pero se inicia por omisión en modalidad de 32 bits. Si necesita que OS/400 PASE se ejecute en modalidad de 64 bits, deberá establecer la variable de entorno QIBM_JAVA_PASE_STARTUP antes de iniciar la JVM. Para obtener más información, consulte Variable de entornos de Java OS/400 PASE.

Consejos para la utilización de VNC

Esta sección ofrece consejos adicionales para utilizar VNC (Virtual Network Computing).

Iniciar un servidor de pantalla VNC desde un programa CL: El ejemplo siguiente presenta una forma de establecer la variable de entorno DISPLAY e iniciar VNC automáticamente utilizando mandatos de lenguaje de control (CL):

```
    CALL QP2SHELL PARM('/QOpenSys/QIBM/ProdData/DeveloperTools/vnc/vncserver_java' ':n')
ADDENVVAR ENVVAR(DISPLAY) VALUE('nombresistema:n')
```

donde:

- *nombresistema* es el nombre de sistema principal o dirección IP del sistema iSeries en el que se está ejecutando VNC
- *n* es el valor numérico que representa el número de pantalla que desea iniciar

Nota: En el ejemplo se presupone que no está ejecutando la pantalla :*n* y que ha creado satisfactoriamente el archivo de contraseñas de VNC necesario. Para obtener más información sobre cómo crear un archivo de contraseñas, consulte la página siguiente:

Crear un archivo de contraseñas de VNC

Detener un servidor de pantalla VNC desde un programa CL: El siguiente código muestra una manera de detener un servidor VNC desde un programa CL:

```
CALL QP2SHELL PARM('/QOpenSys/QIBM/ProdData/DeveloperTools/vnc/vncserver_java' '-kill' ':n')
```

donde *n* es el valor numérico que representa el número de pantalla que desea terminar.

Buscar servidores de pantalla VNC en ejecución: Para determinar qué servidores VNC están ejecutándose actualmente en un sistema iSeries, si hay alguno, complete los siguientes pasos:

1. Desde una línea de mandatos de OS/400, inicie una shell PASE:
   ```
   CALL QP2TERM
   ```
2. Desde el indicador de la shell PASE, utilice el mandatops de PASE para listar los servidores VNC:
   ```
   ps gaxuw | grep Xvnc
   ```

La salida resultante de este mandato revelará qué servidores VNC están en ejecución, con el siguiente formato:

```
john  418  0.9  0.0  5020  0  - A  Jan 31 222:26
  /QOpenSys/QIBM/ProdData/DeveloperTools/vnc/Xvnc :1 -desktop X -httpd
jane   96  0.2  0.0   384  0  - A  Jan 30 83:54
  /QOpenSys/QIBM/ProdData/DeveloperTools/vnc/Xvnc :2 -desktop X -httpd
```

Donde:

- La primera columna es el perfil que ha iniciado el servidor.
- La segunda columna es el ID de proceso PASE del servidor.
- La información que empieza por */QOpensys/* es el mandato que ha iniciado el servidor VNC (incluidos los argumentos). El número de pantalla suele ser el primer elemento de la lista de argumentos del mandato Xvnc.

  Nota: el proceso Xvnc, mostrado en la salida del ejemplo anterior, es el nombre del propio programa servidor VNC. Xvnc se inicia cuando se ejecuta el script vncserver_java, el cuál prepara el entorno y los parámetros para Xvnc y, a continuación, inicia Xvnc.

Consejos para el uso de NAWT con WebSphere Application Server

Antes de leer la siguiente información, asegúrese de que comprende cómo debe instalar y utilizar Native Abstract Windowing Toolkit (NAWT) en el servidor iSeries. Concretamente, debe saber cómo utilizar NAWT con la versión de Java 2 Software Development Kit (J2SDK) y el release de OS/400 que utiliza.

Garantizar las comunicaciones seguras: Al utilizar WebSphere Application Server y NAWT, debe habilitar comunicaciones seguras entre el servidor Virtual Network Computing (VNC) y WebSphere Application Server.

Un método denominado comprobación de autorización X garantiza las comunicaciones seguras entre WebSphere Application Server y el servidor VNC.

El proceso de iniciar el servidor VNC crea un archivo .Xauthority que contiene información de clave cifrada. Las comunicaciones seguras entre WebSphere Application y VNC **REQUIEREN** que tanto WebSphere Application Server como VNC tengan acceso a la información de clave cifrada en el archivo .Xauthority.

Utilizar la comprobación de autorización X: Utilice uno de los siguientes métodos para utilizar la comprobación de autorización X:

Ejecutar WebSphere Application Server y VNC utilizando el mismo perfil

Una manera de garantizar las comunicaciones seguras entre WebSphere Application y el servidor VNC es ejecutar WebSphere Application Server desde el mismo perfil que utilice para iniciar el servidor VNC. Para ejecutar WebSphere Application y VNC con el mismo perfil, debe cambiar el perfil de usuario bajo el que se ejecuta el servidor de aplicaciones.

Para cambiar el perfil de usuario para el servidor de aplicaciones desde al perfil de usuario por omisión (QEJBSVR) a un perfil distinto, debe realizar las siguientes acciones:

1. Utilice la consola administrativa de WebSphere Application Server para cambiar la configuración del servidor de aplicaciones
2. Utilice iSeries Navigator para habilitar el nuevo perfil

Para obtener información sobre cómo utilizar la consola administrativa de WebSphere Application Server y iSeries Navigator, consulte la siguiente documentación:

WebSphere Application Server

Gestionar usuarios y grupos con Management Central

Ejecutar WebSphere Application Server y VNC utilizando perfiles distintos

Cuando desee que WebSphere Application Server y VNC utilicen perfiles distintos, puede garantizar comunicaciones seguras haciendo que WebSphere Application Server utilice el archivo .Xauthority.

Para habilitar WebSphere Application Server para que utilice el archivo .Xauthority, complete los pasos siguientes:

1. Cree un nuevo archivo .Xauthority (o actualice un archivo .Xauthority ya existente) iniciando el servidor VNC desde su perfil de usuario. Desde una línea de mandatos de lenguaje de control (CL) de OS/400, teclee el siguiente mandato y pulse **INTRO**:

   ```
   CALL QP2SHELL PARM('/QOpenSys/QIBM/ProdData/DeveloperTools/vnc/vncserver_java' ':n')
   ```

   donde *n* es el número de pantalla (un valor numérico en el rango de 1 a 99).

   Nota: El archivo .Xauthority reside en el directorio para el perfil bajo el que ejecuta el servidor VNC.
2. Utilice los siguientes mandatos CL para otorgar al perfil bajo el que ejecuta WebSphere Application Server la autorización para leer el archivo .Xauthority:

   ```
   CHGAUT OBJ('/home') USER(WASprofile) DTAAUT(*RX)
   CHGAUT OBJ('/home/VNCprofile') USER(WASprofile) DTAAUT(*RX)
   CHGAUT OBJ('/home/VNCprofile/.Xauthority') USER(WASprofile) DTAAUT(*R)
   ```

   donde *VNCprofile* y *WASprofile* son los perfiles adecuados bajo los que ejecuta el servidor VNC y WebSphere Application Server.
3. Desde la consola administrativa de WebSphere Application Server, defina las variables de entorno DISPLAY y XAUTHORITY para su aplicación:
   - Para DISPLAY, utilice: *system*`:`*n* o `localhost:`*n*

     donde *system* es el nombre o dirección IP del sistema iSeries y *n* es el número de pantalla que ha utilizado para iniciar el servidor VNC.
   - Para XAUTHORITY, utilice: `/home/`*VNCprofile*`/.Xauthority`

     donde *VNCprofile* es el perfil que ha iniciado el servidor VNC.

4. Active los cambios en la configuración reiniciando WebSphere Application Server.

Para obtener información sobre cómo utilizar la consola administrativa de WebSphere Application Server, consulte la siguiente documentación:

WebSphere Application Server

Seguridad Java

La mayoría de los programas Java que se ejecutan en un servidor iSeries son aplicaciones, no applets, y por lo tanto el modelo de seguridad de "cajón de arena" (sandbox) no representa ninguna restricción para ellas. Desde el punto de vista de la seguridad, las aplicaciones Java están sujetas a las mismas restricciones de seguridad que cualquier otro programa de un servidor iSeries. Para ejecutar un programa Java en un servidor iSeries, debe tener autorización sobre el archivo de clase del sistema de archivos integrado. El programa, una vez iniciado, se ejecuta con la autorización del usuario.

Se puede utilizar la autorización adoptada para acceder a los objetos con la autorización del usuario que ejecuta el programa y la autorización del propietario del programa. La autorización adoptada otorga temporalmente a un usuario autorización sobre objetos a los que, en principio, no habría tenido permiso para acceder. En el tema dedicado al mandato Crear programa Java (CRTJVAPGM) hallará información detallada sobre los dos nuevos parámetros de autorización adoptada, que son USRPRF y USEADPAUT.

IBM Developer Kit para Java proporciona las siguientes características de seguridad para las aplicaciones Java:

El modelo de seguridad Java
El cargador y el verificador de bytecodes, dentro de la máquina virtual Java, también proporcionan una medida de seguridad Java mediante la utilización del modelo de seguridad Java. Al igual que sucede con los applets, el cargador y el verificador de bytecodes comprueban que los bytecodes sean válidos y que los tipos de datos se utilicen correctamente. También comprueban que se acceda correctamente a los registros y a la memoria y que la pila no sufra desbordamientos ni subdesbordamientos. Estas comprobaciones garantizan que la máquina virtual Java puede ejecutar con total seguridad la clase sin poner en peligro la integridad del sistema.

Ampliación criptográfica Java
La implementación de Java Cryptography Extension (JCE) del servidor iSeries es compatible con la implementación de Sun Microsystems, Inc. Esta documentación describe los aspectos exclusivos de la implementación iSeries. se presupone que el usuario está familiarizado con la documentación general de JCE.

Java Secure Socket Extension
Java Secure Socket Extension (JSSE) es la implementación Java del protocolo de capa de sockets segura (SSL). JSSE utiliza SSL y el protocolo TLS (Transport Layer Security) para permitir a los clientes y servidores efectuar comunicaciones seguras sobre TCP/IP. Esta documentación describe los aspectos exclusivos de la implementación iSeries de JSSE. Se supone que el usuario está familiarizado con la documentación general de JSSE.

Servicio de autenticación y autorización Java
El Servicio de autenticación y autorización Java (JAAS) es otro elemento de seguridad soportado por IBM Developer Kit para Java. Actualmente, Java 2 Software Development Kit (J2SDK), Standard Edition proporciona controles de acceso que se basan en dónde se originó el código y en quién lo firmó (controles de acceso basados en el origen del código). Sin embargo, a J2SDK le falta la

capacidad de forzar controles de acceso adicionales basados en quién ejecuta el código. JAAS proporciona una infraestructura que añade este soporte al modelo de seguridad de Java 2.

Java Generic Security Service
Java Generic Security Service (JGSS) es otro elemento de seguridad soportado por IBM Developer Kit para Java. JGSS proporciona una interfaz genérica para la mensajería segura entre aplicaciones. JGSS da soporte a distintos mecanismos de seguridad basados en claves secretas, claves públicas u otras tecnologías de seguridad.

Nota: para J2SDK, versión 1.4, JAAS, JCE, JGSS y JSSE forman parte del JDK básico y no se consideran ampliaciones. En versiones anteriores de JDK, estos elementos de seguridad eran ampliaciones.

El modelo de seguridad Java

El usuario puede transferir applets Java desde cualquier sistema; por consiguiente, existen mecanismos de seguridad dentro de la máquina virtual Java que suministran protección contra applets malintencionados. El sistema de ejecución Java verifica los bytecodes a medida que la máquina virtual Java los va cargando. Esto garantiza que son bytecodes válidos y que el código no viola ninguna de las restricciones que la máquina virtual Java impone a los applets Java. Las restricciones afectan a las operaciones que pueden realizar los applets, a la forma en que estos pueden acceder a la memoria y a la manera en que pueden utilizar la máquina virtual Java. Se imponen restricciones con el fin de impedir que un applet Java pueda acceder al sistema operativo subyacente o a los datos del sistema. Este modelo de seguridad se denomina ″cajón de arena″ (sandbox), pues según este modelo los applets Java solo tienen libertad para ″jugar″ en el cajón de arena reservado para ellos.

El modelo de seguridad ″cajón de arena″ es una combinación formada por el cargador de clases, el verificador de archivos de clase y la clase java.lang.SecurityManager.

Para obtener más información acerca de la seguridad, consulte la documentación de Seguridad de Sun Microsystems, Inc. y la sección Proteger aplicaciones con SSL.

Ampliación de criptografía Java

La ampliación de criptografía Java (JCE) 1.2 es una ampliación estándar de Java 2 Software Development Kit (J2SDK), Standard Edition. La implementación de JCE de un servidor iSeries es compatible con la implementación de Sun Microsystems, Inc. Esta documentación describe los aspectos exclusivos de la implementación iSeries. Se presupone que el usuario está familiarizado con la documentación general de las ampliaciones JCE. Para que le resulte más fácil trabajar con esa información y con la de iSeries, se suministra un enlace que le remite a la documentación de JCE de Sun

.

En el servidor iSeries, el nivel de cifrado se controla mediante el producto Cryptographic Access Provider. Este producto está disponible en dos versiones, 5722-AC2 y 5722-AC3. El producto 5722-AC3 permite todos los algoritmos de cifrado. El producto 5722-AC2 no permite Triple-DES y limita los algoritmos simétricos a 56 bits y los algoritmos asimétricos a 1024 bits.

Con la salvedad de las restricciones impuestas en 5722-AC2 que hemos mencionado, IBM JCE Provider da soporte a estos algoritmos:

- DES
- Triple-DES
- RC2
- RC4
- Blowfish

- RSA
- Diffie-Hellman
- DSA
- Mars
- MD2
- MD5
- SHA-1
- Seal

Además, proporciona asimismo un generador de números aleatorios.

Si desea utilizar IBM JCE con Java 1.2, edite el archivo /QIBM/ProdData/Java400/jdk12/lib/security/java.security. A continuación se muestra la sección del archivo que debe cambiarse.

```
#
# Para utilizar el proveedor de seguridad IBMJCE, es necesario:
# 1) Instalar un producto IBM Cryptographic Access Provider
# 2) Eliminar el carácter de comentario de la segunda entrada
#    de proveedor, como se muestra más abajo.
#
# Lista de proveedores y su orden de preferencia:
#
security.provider.1=sun.security.provider.Sun
#security.provider.2=com.ibm.crypto.provider.IBMJCE
```

Si desea utilizar IBM JCE con Java 1.3, edite el archivo /QIBM/ProdData/OS400/Java400/jdk/lib/security/java.security. A continuación se muestra la sección del archivo que debe cambiarse.

```
#
# Para utilizar el proveedor de seguridad IBMJCE, es necesario:
# 1) Instalar un producto IBM Cryptographic Access Provider
# 2) Eliminar el carácter de comentario de la tercera entrada
#    de proveedor, como se muestra más abajo.
#
# Lista de proveedores y su orden de preferencia:
#
security.provider.1=sun.security.provider.Sun
security.provider.2=com.sun.rsajca.Provider
#security.provider.3=com.ibm.crypto.provider.IBMJCE
```

En ambos casos, tan sólo se trata de suprimir un carácter.

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

Java Secure Socket Extension

Java Secure Socket Extension (JSSE) es la implementación Java del protocolo de capa de sockets segura (SSL). JSSE utiliza SSL y el protocolo TLS (Transport Layer Security) para permitir a los clientes y servidores efectuar comunicaciones seguras sobre TCP/IP.

JSSE proporciona las funciones siguientes:

- Cifrado de datos
- Autenticación de ID de usuarios remotos
- Autenticación de nombres de sistemas remotos
- Autenticación de cliente/servidor

- Integridad de mensajes

Integrado en Java 2 Software Development Kit, Standard Edition (J2SDK), versión 1.4, JSSE proporciona más funciones que SSL en solitario. Para obtener más información, consulte los siguientes temas:

Utilizar SSL (JSSE, versión 1.0.8)
SSL proporciona los medios para autenticar un servidor y un cliente con el fin de ofrecer privacidad e integridad de datos. Todas las comunicaciones SSL comienzan con un ″reconocimiento″ entre el servidor y el cliente. Durante el reconocimiento, SSL negocia la suite de cifrado que utilizarán el cliente y el servidor para comunicarse entre sí. La suite de cifrado es una combinación de diversas funciones de seguridad disponibles a través de SSL. Solamente puede utilizar SSL con J2SDK, versión 1.3.

Utilizar JSSE, versión 1.4
JSSE es como una infraestructura que abstrae los mecanismos subyacentes de SSL y TLS. Mediante la abstracción de la complejidad y las peculiaridades de los protocolos subyacentes, JSSE permite a los programadores utilizar comunicaciones cifradas seguras al tiempo que se minimizan las posibles vulneraciones de seguridad. Esta información sólo hace referencia al uso de JSSE en servidores iSeries que ejecutan J2SDK, versión 1.4.

Nota: esta información hace referencia a la versión de JSSE que ahora viene empaquetada en J2SDK, versión 1.4. Para versiones anteriores de JSSE, consulte Java Secure Socket Extension on the Sun Java Web site

.

Utilizar SSL (JSSE, versión 1.0.8)

Puede emplear Java Secure Socket Extension (JSSE, versión 1.0.8), que es la implementación Java del protocolo de capa de sockets segura (SSL), para dotar de más seguridad a la aplicación Java. Para mejorar la seguridad de la aplicación, SSL hace lo siguiente:

- Protege los datos de comunicación mediante cifrado.
- Autentica los ID de usuarios remotos.
- Autentica los nombres de sistemas remotos.

Nota: SSL utiliza un certificado digital para cifrar la comunicación por sockets de la aplicación Java. Los certificados digitales son un estándar de Internet para identificar aplicaciones, usuarios y sistemas seguros. Puede controlar los certificados digitales mediante IBM Digital Certificate Manager. Para obtener más información, consulte IBM Digital Certificate Manager.

Para conseguir que la aplicación Java sea más segura con SSL:

- Prepare el servidor iSeries para que soporte SSL.
- Diseñe la aplicación Java para que utilice SSL; para ello:
  - Si todavía no utiliza fábricas de sockets, cambie el código de socket Java para que utilice las fábricas de sockets.
  - Cambie el código Java para que utilice SSL.
- Utilice un certificado digital para conseguir que la aplicación Java sea más segura; para ello:
  1. Seleccione un tipo de certificado digital.
  2. Utilice el certificado digital al ejecutar la aplicación.

También puede registrar la aplicación Java como aplicación segura; para ello, utilice la API QsyRegisterAppForCertUse. Para obtener más información, consulte QsyRegisterAppForCertUse.

Para obtener más información acerca de la versión Java de SSL, consulte Java Secure Socket Extension

Preparar el servidor iSeries para el soporte de capa de sockets segura: Para preparar el sistema con el fin de que utilice la capa de sockets segura (SSL), tendrá que instalar el programa bajo licencia Digital Certificate Manager (que es el gestor de certificados digitales):

- 5722-SS1 OS/400 - Digital Certificate Manager

También es necesario instalar uno de estos programas bajo licencia Cryptographic Access Provider:

- 5722-AC1 Cryptographic Access Provider de 40 bits
- 5722-AC2 Cryptographic Access Provider de 56 bits
- 5722-AC3 Cryptographic Access Provider de 128 bits

Además, debe asegurarse de que puede acceder a un certificado digital, o bien crear uno, en el sistema. Para obtener más información acerca de la gestión de certificados digitales e Internet, consulte la sección Iniciación a IBM Digital Certificate Manager.

Productos Cryptographic Access Provider: Los productos Cryptographic Access Provider ofrecen numerosas suites de cifrado al sistema. Una suite de cifrado es una combinación formada por distintas características de seguridad. En esta lista se indica cuáles son las suites de cifrado que ofrece cada producto Cryptographic Access Provider:

5722-AC1 Cryptographic Access Provider de 40 bits
SSL_RSA_WITH_NULL_MD5
SSL_RSA_WITH_NULL_SHA
SSL_RSA_EXPORT_WITH_RC4_40_MD5
SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5

5722-AC2 Cryptographic Access Provider de 56 bits
SSL_RSA_WITH_NULL_MD5
SSL_RSA_WITH_NULL_SHA
SSL_RSA_WITH_DES_CBC_SHA
SSL_RSA_WITH_DES_CBC_MD5
SSL_RSA_EXPORT_WITH_RC4_40_MD5
SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5

5722-AC3 Cryptographic Access Provider de 128 bits
SSL_RSA_WITH_NULL_MD5
SSL_RSA_WITH_NULL_SHA
SSL_RSA_EXPORT_WITH_RC4_40_MD5
SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5
SSL_RSA_WITH_RC4_128_SHA
SSL_RSA_WITH_DES_CBC_SHA
SSL_RSA_WITH_3DES_EDE_CBC_SHA
SSL_RSA_WITH_RC4_128_MD5
SSL_RSA_WITH_RC2_CBC_128_MD5
SSL_RSA_WITH_DES_CBC_MD5
SSL_RSA_WITH_3DES_EDE_CBC_MD5

Puede que existan limitaciones con respecto al Cryptographic Access Provider que puede elegir, en función del país o región en el que se encuentre. Una vez cargado un producto Cryptographic Access Provider, se puede utilizar cualquiera de las suites de cifrado que en él se ofrecen.

Cambiar el código Java para que utilice fábricas de sockets: Para utilizar SSL (capa de sockets segura) con el código existente, primero debe cambiar el código de forma que utilice fábricas de sockets.

Para cambiar el código de forma que utilice fábricas de sockets, lleve a cabo los siguientes pasos:

1. Añada la línea siguiente al programa con el fin de importar la clase SocketFactory:
   ```
   import javax.net.*;
   ```
2. Añada una línea que declare una instancia de un objeto SocketFactory. Por ejemplo:
   ```
   SocketFactory socketFactory
   ```
3. Inicialice la instancia de SocketFactory definiéndola como igual al método SocketFactory.getDefault(). Por ejemplo:
   ```
   socketFactory = SocketFactory.getDefault();
   ```

   La declaración completa de SocketFactory debe ser así:
   ```
   SocketFactory socketFactory = SocketFactory.getDefault();
   ```
4. Inicialice los sockets existentes. Llame al método createSocket(host,port) de SocketFactory en la fábrica de sockets para cada socket que declare.

   Las declaraciones de sockets deben ser así:
   ```
   Socket s = socketFactory.createSocket(host,port);
   ```

   Donde:
   - *s* es el socket que se está creando.
   - *socketFactory* es la instancia de SocketFactory creada en el paso 2.
   - *host* es una variable de tipo serie que representa el nombre de un servidor de sistema principal.
   - *port* es una variable de tipo entero que representa el número de puerto de la conexión por socket.

Una vez realizados todos los pasos anteriores, el código utilizará fábricas de sockets. No hay que hacer más cambios en el código. Todos los métodos a los que se llama y la sintaxis con sockets seguirán funcionando.

Consulte la sección Ejemplos: cambiar el código Java para que utilice fábricas de sockets de servidor para obtener un ejemplo de un programa cliente que se convierte para utilizar fábricas de sockets.

Consulte el Ejemplo: cambiar el código Java para que utilice fábricas de sockets de cliente para obtener un ejemplo de programa cliente que se convierte para utilizar fábricas de sockets.

Cambiar el código Java para que utilice la capa de sockets segura: Si el código utiliza fábricas de sockets para crear los sockets, puede añadir el soporte de capa de sockets segura (SSL) al programa. Si el código aún no utiliza fábricas de sockets, consulte la sección Cambiar el código Java para que utilice fábricas de sockets.

Para cambiar el código de forma que utilice SSL, lleve a cabo los siguientes pasos:

1. Importe javax.net.ssl.* para añadir el soporte SSL:
   ```
   import javax.net.ssl.*;
   ```
2. Declare una fábrica de sockets utilizando SSLSocketFactory para inicializarla:
   ```
   SocketFactory newSF = SSLSocketFactory.getDefault();
   ```
3. Utilice la nueva fábrica de sockets para inicializar los sockets de la misma manera que ha utilizado la anterior:
   ```
   Socket s = newSF.createSocket(args[0], serverPort);
   ```

Ahora el código ya utiliza el soporte SSL. No hay que hacer más cambios en el código.

Consulte los Ejemplos: cambiar el cliente Java para que utilice la capa de sockets segura y Ejemplos: cambiar el servidor Java para que utilice la capa de sockets segura para obtener código de ejemplo.

Seleccionar un certificado digital: A la hora de decidir cuál es el certificado digital que va a utilizar, debe tomar en consideración diversos factores. Se puede utilizar el certificado por omisión del sistema o bien especificar otro certificado.

Interesa utilizar el certificado por omisión del sistema si:
- No tiene requisitos específicos de seguridad para la aplicación Java.
- Se desconoce el tipo de seguridad que se necesita para la aplicación Java.
- El certificado por omisión del sistema cumple los requisitos de seguridad de la aplicación Java.

Nota: si decide que desea utilizar el certificado por omisión del sistema, consulte con el administrador del sistema para asegurarse de que se ha creado un certificado por omisión del sistema. Para obtener más información acerca de la gestión de certificados digitales, consulte la sección Iniciación a IBM Digital Certificate Manager.

Si no desea utilizar el certificado por omisión del sistema, tendrá que elegir otro certificado. Puede optar por dos tipos de certificados:
- Un **certificado de usuario**, que identifica al usuario de la aplicación.
- Un **certificado del sistema**, que identifica el sistema en el que se ejecuta la aplicación.

Debe utilizar un certificado de usuario si:
- la aplicación se ejecuta como aplicación cliente.
- desea que el certificado identifique al usuario que trabaja con la aplicación.

Debe utilizar un certificado de sistema si:
- la aplicación se ejecuta como aplicación de servidor.
- desea que el certificado identifique en qué sistema se ejecuta la aplicación.

Una vez que sepa cuál es el tipo de certificado que necesita, podrá elegir cualquiera de los certificados digitales que haya en los contenedores de certificados a los que pueda acceder.

Utilizar el certificado digital al ejecutar la aplicación Java: Para emplear SSL (capa de sockets segura), debe ejecutar la aplicación Java utilizando un certificado digital.

Para especificar el certificado digital que debe utilizarse, emplee las siguientes propiedades:
- os400.certificateContainer
- os400.certificateLabel

Por ejemplo, si desea ejecutar la aplicación Java MyClass.class mediante el certificado digital MYCERTIFICATE, y MYCERTIFICATE está en el contenedor de certificados digitales YOURDCC, el mandato `java` es el siguiente:

```
java -Dos400.certificateContainer=YOURDCC
     -Dos400.certificateLabel=MYCERTIFICATE MyClass
```

Si aún no ha decidido el certificado digital que desea utilizar, consulte la sección Seleccionar un certificado digital. También puede decidir que va a utilizar el certificado por omisión del sistema, que está almacenado en el contenedor de certificados por omisión del sistema.

Para utilizar el certificado digital por omisión del sistema, no es necesario que especifique ningún certificado ni ningún contenedor de certificados en ninguna parte. La aplicación Java utilizará el certificado digital por omisión del sistema de forma automática.

Para obtener más información acerca de la gestión de certificados digitales e Internet, consulte la sección Iniciación a IBM Digital Certificate Manager.

Los certificados digitales y la propiedad -os400.certificateLabel: Los certificados digitales son un estándar de Internet para identificar aplicaciones, usuarios y sistemas seguros. Los certificados digitales se almacenan en contenedores de certificados digitales. Si desea utilizar el certificado por omisión de un contenedor de certificados digitales, no es necesario que especifique una etiqueta de certificado. Si desea utilizar un certificado digital determinado, es necesario que especifique la etiqueta de dicho certificado en el mandato `java` utilizando esta propiedad:

```
os400.certificateLabel=
```

Por ejemplo, si el nombre del certificado que desea utilizar es MYCERTIFICATE, el mandato `java` que entre será así:

```
java -Dos400.certificateLabel=MYCERTIFICATE MyClass
```

En este ejemplo, la aplicación Java MyClass utiliza el certificado MYCERTIFICATE. MYCERTIFICATE debe estar en el contenedor de certificados por omisión del sistema para que MyClass pueda utilizarlo.

Los contenedores de certificados digitales y la propiedad -os400.certificateContainer: Los contenedores de certificados digitales almacenan certificados digitales. Si desea utilizar el contenedor de certificados por omisión del sistema iSeries, no es necesario que especifique un contenedor de certificados. Si desea utilizar un contenedor de certificados digitales determinado, es necesario que especifique dicho contenedor de certificados digitales en el mandato `java` utilizando esta propiedad:

```
os400.certificateContainer=
```

Por ejemplo, si el nombre del contenedor de certificados que contiene el certificado digital que desea utilizar se denomina MYDCC, el mandato `java` que debe entrar sería:

```
java -Dos400.certificateContainer=MYDCC MyClass
```

En este ejemplo, la aplicación Java llamada MyClass.class se ejecutaría en el sistema utilizando el certificado digital por omisión que se halla en el contenedor de certificados digitales llamado MYDCC. Los sockets creados en la aplicación utilizarían el certificado por omisión de MYDCC para identificarse y hacer que las comunicaciones sean seguras.

Si deseara utilizar el certificado digital MYCERTIFICATE del contenedor de certificados digitales, entraría un mandato `java` como el siguiente:

```
java -Dos400.certificateContainer=MYDCC
     -Dos400.certificateLabel=MYCERTIFICATE MyClass
```

Utilizar Java Secure Socket Extension, versión 1.4

Java Secure Socket Extension (JSSE) utiliza el protocolo de capa de sockets segura (SSL) y el protocolo TLS (Transport Layer Security) para proporcionar comunicaciones cifradas seguras entre los clientes y servidores.

La implementación por parte de IBM de JSSE se denomina IBM JSSE. IBM JSSE incluye un proveedor iSeries JSSE nativo y un proveedor Java JSSE puro.

Para obtener más información sobre cómo configurar el servidor iSeries para dar soporte a JSSE, utilice los enlaces siguientes:

Configurar el servidor para dar soporte a JSSE
Descubra cómo configurar el servidor iSeries para utilizar IBM JSSE. La información incluye los requisitos de software, cómo cambiar los proveedores JSSE y las propiedades de seguridad y las propiedades del sistema necesarias.

Utilizar el proveedor iSeries JSSE nativo
Aprenda a utilizar las implementaciones iSeries nativas de la clase de JSSE KeyStore y la clase SSLConfiguration.

Ejemplos de JSSE
Utilice los programas de ejemplo para descubrir cómo puede emplear JSSE en las aplicaciones. El código fuente Java de ejemplo muestra cómo los clientes y servidores pueden emplear objetos SSLContext tanto en clientes como en servidores a fin de crear un entorno de comunicaciones seguras.

Configurar el servidor iSeries para dar soporte a JSSE: Si utiliza Java 2 Software Development Kit (J2SDK), versión 1.4 en el servidor iSeries, JSSE ya está configurado. La configuración por omisión utiliza el proveedor iSeries JSSE nativo.

Requisitos de software: Para utilizar JSSE con J2SDK, versión 1.4, debe tener instalado en el servidor iSeries IBM Cryptographic Access Provider de 128 bits (5722-AC3). Para obtener más información, consulte Productos Cryptographic Access Provider.

Cambiar los proveedores JSSE: Puede configurar JSSE para utilizar el proveedor Java JSSE puro en lugar del proveedor iSeries JSSE nativo. Cambiando algunas propiedades de seguridad de JSSE y propiedades del sistema Java específicas, puede conmutar entre los dos proveedores. Para obtener más información, consulte los siguientes temas:

- Proveedores JSSE
- Propiedades de seguridad de JSSE
- Propiedades del sistema Java

Gestores de seguridad: Si ejecuta la aplicación JSSE con un gestor de seguridad Java habilitado, puede que tenga que establecer los permisos de red disponibles. Para obtener más información, consulte SSLPermission en Permissions in the Java 2 SDK

Proveedores JSSE:

IBM JSSE incluye un proveedor iSeries JSSE nativo y dos proveedores Java JSSE puros. El proveedor que elija utilizar dependerá de las necesidades de la aplicación.

Los tres proveedores cumplen la especificación de la interfaz JSSE. Pueden comunicarse entre sí y con cualquier otra implementación de SSL o TLS, incluso implementaciones que no sean Java.

Proveedor Java JSSE puro: El proveedor Java JSSE puro presenta las características siguientes:

- Funciona con cualquier tipo de objeto KeyStore para controlar y configurar certificados digitales (por ejemplo, JKS, PKCS12, etc.).
- Le permite utilizar juntos cualquier combinación de componentes JSSE de diversas implementaciones

IBMJSSE es el nombre de proveedor de la implementación Java pura. Debe pasar este nombre de proveedor, utilizando las mayúsculas y minúsculas correctas, al método java.security.Security.getProvider() o los diversos métodos getInstance() para varias de las clases de JSSE.

Proveedor JSSE FIPS 140-2 Java puro: El proveedor JSSE FIPS 140-2 Java puro ofrece las siguientes características:

- Está en conformidad con FIPS (Federal Information Processing Standards) 140-2 para Módulos criptográficos
- Funciona con cualquier tipo de objeto KeyStore para controlar y configurar certificados digitales

Nota: El proveedor JSSE FIPS 140-2 Java puro no permite que componentes de ninguna otra implementación se conecten a su implementación.

IBMJSSEFIPS es el nombre de proveedor de la implementación JSSE FIPS 140-2 Java puro. Debe pasar este nombre de proveedor, utilizando las mayúsculas y minúsculas correctas, al método java.security.Securirty.getProvider() o a los diversos métodosgetInstance() para varias de las clases de JSSE.

Proveedor iSeries JSSE nativo: El proveedor iSeries JSSE nativo presenta las características siguientes:

- Utiliza el soporte SSL de iSeries nativo
- 

  Permite el uso del Gestor de certificados digitales para configurar y controlar certificados digitales. Esto se proporciona mediante un tipo exclusivo de KeyStore de iSeries (IbmISeriesKeyStore).
- Ofrece un rendimiento óptimo
- Le permite utilizar juntos cualquier combinación de componentes JSSE de diversas implementaciones. Sin embargo, para conseguir el mejor rendimiento posible, utilice solamente componentes iSeries JSSE nativo.

IbmISeriesSslProvider es el nombre de la implementación iSeries nativa. Debe pasar este nombre de proveedor, utilizando las mayúsculas y minúsculas correctas, al método java.security.Security.getProvider() o los diversos métodos getInstance() para varias de las clases de JSSE.

Cambiar el proveedor JSSE por omisión: Puede cambiar el proveedor JSSE por omisión efectuando las modificaciones adecuadas en las propiedades de seguridad. Para obtener más información, consulte el siguiente tema:

- Propiedades de seguridad de JSSE

Tras cambiar el proveedor JSSE, compruebe que las propiedades del sistema especifican la configuración correcta para la información de certificados digitales (almacén de claves) que requiere el nuevo proveedor. Para obtener más información, consulte el siguiente tema:

- Propiedades del sistema Java

Propiedades de seguridad de JSSE: Una máquina virtual Java (JVM) utiliza muchas propiedades de seguridad importantes que se establecen editando el archivo de propiedades de seguridad maestro de Java. Este archivo, denominado java.security, normalmente se encuentra en el directorio /QIBM/ProdData/Java400/jdk14/lib/security del servidor iSeries.

La lista siguiente describe varias propiedades de seguridad relevantes para utilizar JSSE. Utilice las descripciones a modo de guía para editar el archivo java.security.

security.provider.<entero>

El proveedor JSSE que desea utilizar. Estáticamente también registra las clases de proveedor criptográfico. Especifique los distintos proveedores JSSE exactamente igual que en el ejemplo siguiente:

```
security.provider.5=com.ibm.as400.ibmonly.net.ssl.Provider
     security.provider.6=com.ibm.jsse.IBMJSSEProvider
security.provider.7=com.ibm.fips.jsse.IBMJSSEFIPSProvider
```

ssl.KeyManagerFactory.algorithm

Especifica el algoritmo de KeyManagerFactory por omisión. Para el proveedor iSeries JSSE nativo, utilice lo siguiente:

```
ssl.KeyManagerFactory.algorithm=IbmISeriesX509
```

Para el proveedor Java JSSE puro, utilice lo siguiente:

```
ssl.KeyManagerFactory.algorithm=IbmX509
```

Para obtener más información, consulte el javadoc acerca de javax.net.ssl.KeyManagerFactory.

ssl.TrustManagerFactory.algorithm

Especifica el algoritmo de TrustManagerFactory por omisión. Para el proveedor iSeries JSSE nativo, utilice lo siguiente:

```
ssl.TrustManagerFactory.algorithm=IbmISeriesX509
```

Para el proveedor Java JSSE puro, utilice lo siguiente:

```
ssl.TrustManagerFactory.algorithm=IbmX509
```

Para obtener más información, consulte el javadoc acerca de javax.net.ssl.TrustManagerFactory.

ssl.SocketFactory.provider

Especifica la fábrica de sockets SSL por omisión. Para el proveedor iSeries JSSE nativo, utilice lo siguiente:

```
ssl.SocketFactory.provider=com.ibm.as400.ibmonly.net.ssl.SSLSocketFactoryImpl
```

Para el proveedor Java JSSE puro, utilice lo siguiente:

```
ssl.SocketFactory.provider=com.ibm.jsse.JSSESocketFactory
```

Para obtener más información, consulte el javadoc acerca de javax.net.ssl.SSLSocketFactory.

ssl.ServerSocketFactory.provider

Especifica la fábrica de sockets de servidor SSL por omisión. Para el proveedor iSeries JSSE nativo, utilice lo siguiente:

```
ssl.ServerSocketFactory.provider=com.ibm.as400.ibmonly.net.ssl.SSLServerSocketFactoryImpl
```

Para el proveedor Java JSSE puro, utilice lo siguiente:

```
ssl.ServerSocketFactory.provider=com.ibm.jsse.JSSEServerSocketFactory
```

Para obtener más información, consulte el javadoc acerca de javax.net.ssl.SSLServerSocketFactory.

Propiedades del sistema Java de JSSE: Si desea emplear JSSE en las aplicaciones, debe especificar varias propiedades del sistema que los objetos SSLContext por omisión necesitan para proporcionar una confirmación de la configuración. Algunas de las propiedades hacen referencia a ambos proveedores, mientras que otras sólo son válidas para el proveedor iSeries nativo.

Al utilizar el proveedor iSeries JSSE nativo, si no especifica ninguna de las propiedades, os400.certificateContainer toma el valor por omisión *SYSTEM, que significa que JSSE utiliza la entrada por omisión del almacén de certificados del sistema.

Propiedades que funcionan para ambos proveedores: Las propiedades siguientes son válidas para ambos proveedores JSSE. En cada descripción se indica la propiedad por omisión, si corresponde.

javax.net.ssl.trustStore

El nombre del archivo que contiene el objeto KeyStore que desea que utilice el TrustManager por omisión. El valor por omisión es jssecacerts, o cacerts (si no existe jssecacerts).

javax.net.ssl.trustStoreType

El tipo de objeto KeyStore que desea que utilice el TrustManager por omisión. El valor por omisión es el valor devuelto por el método KeyStore.getDefaultType.

javax.net.ssl.trustStorePassword

La contraseña del objeto KeyStore que desea que utilice el TrustManager por omisión.

javax.net.ssl.keyStore

El nombre del archivo que contiene el objeto KeyStore que desea que utilice el KeyManager por omisión.

javax.net.ssl.keyStoreType

El tipo de objeto KeyStore que desea que utilice el KeyManager por omisión. El valor por omisión es el valor devuelto por el método KeyStore.getDefaultType.

javax.net.ssl.keyStorePassword

La contraseña del objeto KeyStore que desea que utilice el KeyManager por omisión.

Propiedades que funcionan únicamente para el proveedor iSeries JSSE nativo: Las propiedades siguientes sólo son válidas para el proveedor iSeries JSSE nativo.

os400.secureApplication

El identificador de la aplicación. JSSE sólo utiliza esta propiedad cuando no se ha especificado ninguna de las propiedades siguientes:

- javax.net.ssl.keyStore
- javax.net.ssl.keyStorePassword
- javax.net.ssl.keyStoreType
- 
  javax.net.ssl.trustStore
- javax.net.ssl.trustStorePassword
- javax.ssl.net.trustStoreType

os400.certificateContainer

El nombre del archivo de claves que desea utilizar. JSSE sólo utiliza esta propiedad cuando no se ha especificado ninguna de las propiedades siguientes:

- javax.net.ssl.keyStore
- javax.net.ssl.keyStorePassword
- javax.net.ssl.keyStoreType

-

  javax.net.ssl.trustStore
- javax.net.ssl.trustStorePassword
- javax.ssl.net.trustStoreType

- os400.secureApplication

os400.certificateLabel

La etiqueta de archivo de claves que desea utilizar.

JSSE sólo utiliza esta propiedad cuando no se ha especificado ninguna de las propiedades siguientes:
- javax.net.ssl.keyStore
- javax.net.ssl.keyStorePassword
- javax.net.ssl.trustStore
- javax.net.ssl.trustStorePassword
- javax.ssl.net.trustStoreType
- os400.secureApplication

Información adicional: Para obtener más información sobre las propiedades del sistema, consulte los temas siguientes:
- Propiedades del sistema Java para J2SDK, versión 1.4, en los servidores iSeries
- System Properties en el sitio Web de Java de Sun

Utilizar el proveedor iSeries JSSE nativo: El proveedor iSeries JSSE nativo ofrece el conjunto completo de interfaces y clases de JSSE. Para emplear el proveedor iSeries nativo de forma eficaz, consulte la siguiente información:
- "Valores de protocolo para el método SSLContext.getInstance"
- "Implementación nativa de KeyStore" en la página 205
- "Recomendaciones al utilizar el proveedor iSeries nativo" en la página 205
- Información de javadoc para SSLConfiguration

Valores de protocolo para el método SSLContext.getInstance: La tabla siguiente identifica y describe los valores de protocolo para el método SSLContext.getInstance del proveedor iSeries JSSE nativo.

Valor de protocolo	Protocolos SSL soportados
SSL	SSL versión 2, SSL versión 3 y TLS versión 1
SSLv2	SSL versión 2
SSLv3	SSL versión 3
TLS	SSL versión 2, SSL versión 3 y TLS versión 1
TLSv1	TLS versión 1

Valor de protocolo	Protocolos SSL soportados
SSL_TLS	SSL versión 2, SSL versión 3 y TLS versión 1

Implementación nativa de KeyStore: El proveedor iSeries nativo ofrece una implementación de la clase KeyStore de tipo IbmISeriesKeyStore. Esta implementación de almacén de claves proporciona una envoltura alrededor del soporte de gestor de certificados digitales. El contenido del almacén de claves se basa en un identificador de aplicación específico o un archivo de claves, una contraseña y una etiqueta. JSSE carga las entradas del almacén de claves del gestor de certificados digitales. Para cargar las entradas, JSSE utiliza el identificador de aplicación adecuado o la información del archivo de claves cuando la aplicación intenta por primera vez acceder a las entradas del almacén de claves o la información del archivo de claves. No es posible modificar el almacén de claves y todos los cambios de configuración deben efectuarse mediante el Gestor de certificados digitales.

Para obtener más información sobre cómo utilizar el gestor de certificados digitales, consulte el siguiente tema:

Gestor de certificados digitales

Recomendaciones al utilizar el proveedor iSeries nativo: Las siguientes son algunas recomendaciones para hacer que el proveedor iSeries nativo se ejecute de la manera más eficaz posible.

- Para que el proveedor iSeries JSSE nativo funcione, la aplicación JSSE debe emplear únicamente componentes de la implementación nativa. Por ejemplo, la aplicación habilitada para JSSE iSeries nativa no puede emplear un objeto X509KeyManager creado utilizando el proveedor Java JSSE puro para inicializar correctamente un objeto SSLContext creado utilizando el proveedor iSeries JSSE nativo.
- Asimismo, debe inicializar las implementaciones de X509KeyManager y X509TrustManager en el proveedor iSeries nativo utilizando un objeto IbmISeriesKeyStore o com.ibm.as400.SSLConfiguration.

Nota: Las recomendaciones mencionadas pueden cambiar en releases posteriores, con lo que el proveedor iSeries JSSE nativo podría permitir conectar componentes no nativos (por ejemplo, JKS KeyStore o IbmX509 TrustManagerFactory).

Ejemplos: IBM Java Secure Sockets Extension: Los ejemplos de JSSE muestran cómo un cliente y un servidor pueden emplear el proveedor iSeries JSSE nativo para crear un contexto que haga posible las comunicaciones seguras.

Nota: ambos ejemplos utilizan el proveedor iSeries JSSE nativo, independientemente de las propiedades especificadas por el archivo java.security.

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

Ejemplo: cliente SSL que utiliza un objeto SSLContext

Este programa cliente de ejemplo utiliza un objeto SSLContext, que inicializa para emplear el ID de aplicación "MY_CLIENT_APP". Este programa utilizará la implementación iSeries nativa independientemente de lo que haya especificado en el archivo java.security.

Ejemplo: servidor SSL que utiliza un objeto SSLContext

El siguiente programa servidor utiliza un objeto SSLContext que inicializa con un archivo de almacén de claves creado anteriormente. El archivo de almacén de claves tiene el nombre `/home/keystore.file` y la contraseña de almacén de claves `password`.

El programa de ejemplo necesita el archivo de almacén de claves para crear un objeto IbmISeriesKeyStore. El objeto KeyStore debe especificar MY_SERVER_APP como identificador de la aplicación.

Para crear el archivo de almacén de claves, puede emplear cualquiera de los mandatos siguientes:

- Desde un indicador de mandatos de Qshell:

```
java com.ibm.as400.SSLConfiguration -create -keystore /home/keystore.file
  -storepass password -appid MY_SERVER_APP
```

  Para obtener más información sobre cómo utilizar mandatos Java con Qshell, consulte el siguiente tema:

  Qshell

- Desde un indicador de mandatos de iSeries:

```
RUNJVA CLASS(com.ibm.as400.SSLConfiguration) PARM('-create' '-keystore'
     '/home/keystore.file' '-storepass' 'password' '-appid' 'MY_SERVER_APP')
```

Servicio de autenticación y autorización Java

El servicio de autenticación y autorización Java (JAAS) es una ampliación estándar de Java 2 Software Development Kit (J2SDK), Standard Edition. Actualmente, J2SDK proporciona controles de acceso que se basan en dónde se originó el código y en quién lo firmó (controles de acceso basados en el origen del código). Sin embargo, a J2SDK le falta la capacidad de forzar controles de acceso adicionales basados en quién ejecuta el código. JAAS proporciona una infraestructura que añade este soporte al modelo de seguridad de Java 2.

IBM y Sun Microsystems, Inc. utilizan la API JAAS como ampliación de J2SDK, versión 1.3. IBM y Sun están introduciendo esta extensión para permitir la asociación de un usuario o identidad específicos con la hebra Java actual. Para ello se utilizan los métodos javax.security.auth.Subject y, opcionalmente, con la hebra del sistema operativo subyacente, se utilizan los métodos com.ibm.security.auth.ThreadSubject.

Nota: En J2SDK, versión 1.4, JAAS ya no es una ampliación, sino que forma parte de SDK básico.

La implementación de JAAS en el servidor iSeries es compatible con la implementación de Sun Microsystems, Inc. Esta documentación describe los aspectos exclusivos de la implementación iSeries. Se presupone que el usuario está familiarizado con la documentación general de las ampliaciones JAAS. Para facilitarle el trabajo con esa información y con la nuestra sobre iSeries, le proporcionamos estos enlaces:

- API Developers Guide, que es una guía informativa sobre cómo utilizar la API de JAAS en el desarrollo de software.
- JAAS LoginModule Developer's Guide se centra en los aspectos de autenticación de JAAS.
- JAAS API Specification, que incluye la información de tipo Javadoc sobre JAAS.

Si desea obtener más detalles sobre cómo se utiliza JAAS, seleccione cualquiera de estos temas:

- Preparar y configurar el servidor iSeries para JAAS
- Ejemplos de JAAS
- Javadoc JAAS específico del servidor iSeries

Preparar y configurar un servidor iSeries para el servicio de autenticación y autorización Java

Debe cumplir los requisitos de software y configurar el servidor iSeries para utilizar el servicio de autenticación y autorización Java (JAAS).

Requisitos de software para ejecutar JAAS 1.0 en un servidor iSeries

Instale los siguientes programas bajo licencia:

- Java 2 SDK, versión 1.4 (J2SDK)
- Para cambiar la identidad de la hebra de OS, es necesario el programa bajo licencia IBM Toolbox para Java (mod 4) (5722-JC1). Este programa contiene las clases ProfileTokenCredential necesarias para dar soporte al cambio de la identidad de hebra de OS de iSeries 400 y las clases de implementación nativas.

Configurar el sistema

Para configurar el sistema con objeto de que utilice JAAS, siga estos pasos:

1. Para J2SDK 1.3, añada al directorio de ampliaciones un enlace simbólico para el archivo jaas13.jar. El cargador de clases de ampliación debe cargar el archivo JAR. Para añadir el enlace, ejecute este mandato (debe ocupar una sola línea) en la línea de mandatos de iSeries:

   ```
   ADDLNK OBJ('/QIBM/ProdData/OS400/Java400/ext/jaas13.jar')
   NEWLNK('/QIBM/ProdData/Java400/jdk13/lib/ext/jaas13.jar')
   ```

   Nota: Para J2SDK 1.4, no es necesario añadir un enlace simbólico al directorio de ampliaciones. JAAS forma parte del SDK básico en esta versión.
2. En ${java.home}/lib/security se proporciona un archivo login.config por omisión que invoca com.ibm.as400.security.auth.login.BasicAuthenticationLoginModule. Este archivo login.config conecta al sujeto autenticado una credencial ProfileTokenCredential de un solo uso. Si desea emplear un archivo login.config propio que tenga distintas opciones, puede incluir la siguiente propiedad del sistema cuando invoque su aplicación:

   ```
   -Djava.security.auth.login.config=su archivo login.config
   ```
3. Añada al directorio de ampliaciones (ext) un enlace simbólico para el archivo jt400Native.jar. Así el cargador de clases de ampliación podrá cargar este archivo. El archivo jaas13.jar necesita este archivo JAR para las clases de implementación de credencial que forman parte de IBM Toolbox para Java. El cargador de clases de aplicación también puede cargar este archivo si se incluye en la variable CLASSPATH. Si este archivo se carga desde el directorio de vía de acceso de clases, no añada el enlace simbólico al directorio de ampliaciones.

   El hecho de enlazar simbólicamente el archivo jt400Native.jar con el directorio /QIBM/ProdData/Java400/jdk14/lib/ext obliga a todos los usuarios de J2SDK 1.4 del servidor a ejecutarse con esta versión de jt400Native.jar. Esto puede no ser conveniente si diversos usuarios necesitan distintas versiones de las clases de IBM Toolbox para Java. Existe la alternativa de colocar el archivo jt400Native.jar en la CLASSPATH de la aplicación, como ya se ha indicado. Aún hay otra opción, que consiste en añadir el enlace simbólico a su propio directorio y luego incluir ese directorio en la vía de acceso de clases del directorio de ampliaciones, especificando la propiedad java.ext.dirs del sistema en el momento de invocar la aplicación.

   Para enlazar el archivo jt400Native.jar al directorio /QIBM/ProdData/Java400/jdk13/lib/ext, ejecute este mandato en la línea de mandatos de iSeries para añadir el enlace:

   ```
   ADDLNK OBJ('/QIBM/ProdData/OS400/jt400/lib/jt400Native.jar')
      NEWLNK('/QIBM/ProdData/Java400/jdk13/lib/ext/jt400Native.jar')
   ```

   Para enlazar el archivo jt400Native.jar al directorio /QIBM/ProdData/Java400/jdk14/lib/ext, ejecute este mandato en la línea de mandatos de iSeries para añadir el enlace:

   ```
   ADDLNK OBJ('/QIBM/ProdData/OS400/jt400/lib/jt400Native.jar')
      NEWLNK('/QIBM/ProdData/Java400/jdk14/lib/ext/jt400Native.jar')
   ```

   Para enlazar el archivo jt400Native.jar a su propio directorio, haga lo siguiente:

   a. Para añadir el enlace, ejecute este mandato en la línea de mandatos de iSeries:

      ```
      ADDLNK OBJ('/QIBM/ProdData/OS400/jt400/lib/jt400Native.jar')
                  NEWLNK('su directorio de ampliaciones/jt400Native.jar')
      ```

b. Cuando llame al programa Java, emplee este patrón:

```
java -Djava.ext.dirs=su directorio de ampliaciones:directorios de
ampliaciones por omisión
```

Nota: Consulte la sección IBM Toolbox para Java para obtener información acerca de las clases de credenciales de iSeries. Pulse **Clases de seguridad**. Pulse **Servicios de autenticación**. Pulse la clase **ProfileTokenCredential**. Pulse **Paquete**.

4. Actualice los archivos de política de Java 2 para otorgar los debidos permisos sobre las ubicaciones reales de los archivos JAR de IBM Toolbox para Java. Aunque estos archivos pueden estar simbólicamente enlazados a los directorios de ampliaciones y a estos directorios se les otorgue java.security.AllPermission en el archivo ${java.home}/lib/security/java.policy, la autorización se basa en la ubicación real de los archivos JAR.

   Para utilizar satisfactoriamente las clases de credenciales de IBM Toolbox para Java, añada este fragmento de código al archivo de política de Java 2 de la aplicación:

```
grant  codeBase "file:/QIBM/ProdData/OS400/jt400/lib/jt400Native.jar"
 {
   permission javax.security.auth.AuthPermission "modifyThreadIdentity";
   permission java.lang.RuntimePermission "loadLibrary.*";
   permission java.lang.RuntimePermission "writeFileDescriptor";
   permission java.lang.RuntimePermission "readFileDescriptor";
 }
```

   También tendrá que añadir estos permisos para codeBase de la aplicación, ya que las operaciones efectuadas por los archivos JAR de IBM Toolbox para Java no se ejecutan en modalidad privilegiada.

   En API Developers Guide hallará información relacionada con los archivos de política de Java 2.

5. Asegúrese de que se han iniciado los servidores de sistema principal iSeries y que están funcionando. Las clases ProfileTokenCredential que residen en Toolbox (por ejemplo, jt400Native.jar) se emplean como credenciales conectadas al sujeto autenticado. Las clases de credenciales necesitan acceder a los servidores de sistema principal. Para verificar que los servidores se han iniciado y están funcionando, teclee lo siguiente en el indicador de mandatos de iSeries:
   - StrHostSVR *all
   - StrTcpSvr *DDM

   Si los servidores ya se habían iniciado, estos pasos no hacen nada. Si los servidores no se habían iniciado, lo harán ahora.

Ejemplos del servicio de autenticación y autorización Java

En esta información proporcionamos un enlace para acceder a algunos ejemplos del servicio de autenticación y autorización Java (JAAS) en un servidor iSeries. Junto con la documentación se incluyen dos ejemplos de JAAS, que se llaman HelloWorld y SampleThreadSubjectLogin. Si desea obtener instrucciones y el código fuente, pulse estos enlaces:

- HelloWorld
- SampleThreadSubjectLogin

IBM Java Generic Security Service (JGSS)

Java Generic Security Service (JGSS) proporciona una interfaz genérica para la autenticación y la mensajería segura. Bajo esta interfaz puede conectar distintos mecanismos de seguridad basados en claves secretas, claves públicas u otras tecnologías de seguridad.

Mediante la abstracción de la complejidad y las peculiaridades de los mecanismos de seguridad subyacentes en una interfaz estándar, JGSS aporta las siguientes ventajas al desarrollo de aplicaciones de red seguras:

- Puede desarrollar la aplicación para una única interfaz abstracta
- Puede emplear la aplicación con distintos mecanismos de seguridad sin efectuar ningún cambio

JGSS define los enlaces Java para GSS-API (Generic Security Service Application Programming Interface), que es una API criptográfica establecida por IETF (Internet Engineering Task Force) y adoptada por The X/Open Group.

La implementación por parte de IBM de JGSS se denomina IBM JGSS. IBM JGSS es una implementación de la infraestructura de GSS-API que utiliza Kerberos V5 como sistema de seguridad subyacente por omisión. También presenta un módulo de inicio de sesión del servicio de autenticación y autorización Java (JAAS) para crear y utilizar credenciales de Kerberos. Asimismo, puede hacer que JGSS efectúe comprobaciones de autorización de JAAS cuando utilice esas credenciales.

IBM JGSS incluye un proveedor iSeries JGSS nativo, un proveedor Java JGSS y versiones para Java de las herramientas de gestión de credenciales de Kerberos (kinit, ktab y klist).

Nota: el proveedor iSeries JGSS nativo utiliza la biblioteca de servicios de autenticación de red (NAS) de iSeries nativa. Al utilizar el proveedor nativo, debe emplear los programas de utilidad Kerberos de iSeries nativos. Para obtener más información, consulte Proveedores JGSS.

Para obtener más información sobre cómo utilizar JGSS, consulte los siguientes temas:

Conceptos de JGSS
Proporciona una introducción de los conceptos de JGSS, con una descripción de alto nivel de las operaciones de GSS-API y una breve explicación de los mecanismos de seguridad.

Configurar el servidor para utilizar JGSS
Descubra cómo configurar el servidor iSeries para utilizar IBM JGSS con Java 2 Software Development Kit, Standard Edition (J2SDK). Entre la información que se facilita se encuentra cómo identificar y establecer los permisos necesarios para emplear JGSS con un gestor de seguridad.

Ejecutar aplicaciones JGSS
Aprenda a ejecutar las aplicaciones JGSS en los servidores iSeries. La documentación incluye una descripción de los conceptos operativos e instrucciones para utilizar JAAS.

Desarrollar aplicaciones JGSS
Vea cómo debe utilizar JGSS para desarrollar aplicaciones seguras. Obtenga información sobre cómo generar señales de transporte, crear objetos JGSS, establecer un contexto y mucho más.

Información de consulta de javadoc de JGSS
Consulte la información de javadoc relacionada con las clases y los métodos del paquete de API org.ietf.jgss y las versiones Java de las herramientas de gestión de credenciales de Kerberos (kinit, ktab y klist).

Ejemplos de JGSS
Utilice los programas de ejemplo para descubrir cómo puede emplear JGSS en las aplicaciones. La documentación de ejemplo contiene el código fuente Java, las instrucciones para ejecutar los ejemplos, los archivos de configuración y política y mucho más.

Para obtener más información sobre la seguridad Java y el servicio de seguridad genérico, consulte la documentación siguiente:

-
- J2SDK Security enhancement

  de Sun Microsystems, Inc., con enlaces para obtener más información acerca de Java GSS-API
- IETF (Internet Engineering Task Force) RFC 2743 Generic Security Services Application Programming Interface Version 2, Update 1

- IETF RFC 2853 Generic Security Service API Version 2: Java Bindings

- The X/Open Group GSS-API Extensions for DCE

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

Conceptos de JGSS

Las operaciones de JGSS constan de cuatro fases diferenciadas, según el estándar de GSS-API (Generic Security Service Application Programming Interface):

1. Recopilación de credenciales para principales
2. Creación y establecimiento de un contexto de seguridad entre los principales de iguales que se comunican
3. Intercambio de mensajes seguros entre los iguales
4. Borrado y liberación de recursos

Asimismo, JGSS emplea la arquitectura JCA (Java Cryptographic Architecture) para ofrecer una conexión transparente de los distintos mecanismos de seguridad.

Consulte los enlaces siguientes para obtener descripciones de alto nivel de estos importantes conceptos de JGSS.

- Principales y credenciales
- Establecimiento del contexto
- Protección e intercambio de mensajes
- Borrado y liberación de recursos
- Mecanismos de seguridad

Principales y credenciales: La identidad con la que una aplicación participa en una comunicación segura JGSS con un igual se denomina principal. Un principal puede ser un usuario real o un servicio desatendido. Un principal adquiere credenciales específicas del mecanismo de seguridad como documento de identidad bajo ese mecanismo. Por ejemplo, al utilizar el mecanismo Kerberos, la credencial de un principal tiene el formato de un ticket de otorgación de tickets (TGT) emitido por un centro de distribución de claves (KDC) de Kerberos. En un entorno de varios mecanismos, una credencial de GSS-API puede contener varios elementos de credencial, cada uno de los cuales representa una credencial de mecanismo subyacente.

El estándar de GSS-API no establece cómo adquiere credenciales un principal y las implementaciones de GSS-API normalmente no proporcionan un método para la adquisición de credenciales. Un principal obtiene las credenciales antes de emplear GSS-API; GSS-API simplemente consulta al mecanismo de seguridad las credenciales en nombre del principal.

IBM JGSS incluye versiones para Java de las herramientas de gestión de credenciales de Kerberos kinit, ktab y klist. Asimismo, IBM JGSS amplía la especificación GSS-API estándar al proporcionar una interfaz de inicio de sesión de Kerberos opcional que utiliza JAAS. El proveedor Java JGSS puro da soporte a la interfaz de inicio de sesión opcional, no así el proveedor iSeries nativo. Para obtener más información, consulte los siguientes temas:

- Obtener credenciales de Kerberos

- Proveedores JGSS

Establecimiento del contexto: Tras adquirir las credenciales de seguridad, los dos iguales que se comunican establecen un contexto de seguridad mediante sus credenciales. Aunque los iguales establecen un único contexto conjunto, cada igual mantiene su propia copia local del contexto. El establecimiento del contexto supone que el igual iniciador se autentica para el igual aceptante. El iniciador puede solicitar la autenticación mutua, en cuyo caso el aceptante se autentica para el iniciador.

Una vez efectuado el establecimiento del contexto, el contexto establecido incluye información de estado (como, por ejemplo, las claves criptográficas compartidas) que hace posible el intercambio posterior de mensajes seguros entre los dos iguales.

Protección e intercambio de mensajes: Tras el establecimiento del contexto, los dos iguales están preparados para participar en intercambios de mensajes seguros. El originador del mensaje llama a su implementación de GSS-API local para codificar el mensaje, con lo que se garantiza la integridad del mensaje y, si se desea, la confidencialidad del mismo. A continuación la aplicación transporta la señal obtenida al igual.

La implementación de GSS-API local del igual utiliza la información del contexto establecido del modo siguiente:
- Verifica la integridad del mensaje
- Descifra el mensaje, si estaba cifrado

Borrado y liberación de recursos: Para liberar recursos, una aplicación JGSS suprime un contexto que ya no es necesario. Aunque una aplicación JGSS puede acceder a un contexto suprimido, todo intento de utilizarlo para el intercambio de mensajes generará una excepción.

Mecanismos de seguridad: La especificación GSS-API consiste en una infraestructura abstracta sobre uno o varios mecanismos de seguridad subyacentes. El modo de interactuar de la infraestructura con los mecanismos de seguridad subyacentes es específico de la implementación. Tales implementaciones se enmarcan en dos categorías generales:
- En un extremo, una implementación monolítica enlaza estrechamente la infraestructura con un único mecanismo. Este tipo de implementación impide el uso de otros mecanismos o incluso distintas implementaciones del mismo mecanismo.
- En el otro extremo, una implementación de gran modularidad ofrece facilidad de uso y flexibilidad. Este tipo de implementación ofrece la posibilidad de conectar distintos mecanismos de seguridad y sus implementaciones a la infraestructura de forma fácil y transparente.

IBM JGSS pertenece a la segunda categoría. Como implementación modular, IBM JGSS utiliza la infraestructura de proveedor definida por la arquitectura JCA (Java Cryptographic Architecture) y trata a cualquier mecanismo subyacente como un proveedor (JCA). Un proveedor JGSS proporciona una implementación concreta de un mecanismo de seguridad JGSS. Una aplicación puede crear instancias de varios mecanismos y utilizarlos.

Un proveedor puede dar soporte a varios mecanismos y JGSS facilita el uso de distintos mecanismos de seguridad. Sin embargo, la especificación GSS-API no proporciona un método para que dos iguales que se comunican elijan un mecanismo cuando hay disponibles varios mecanismos. Una forma de elegir un mecanismo consiste en empezar con el mecanismo SPNEGO (Simple And Protected GSS-API Negotiating), un pseudomecanismo que negocia un mecanismo real entre los dos iguales. IBM JGSS no incluye un mecanismo SPNEGO.

Para obtener más información sobre SPNEGO, consulte el documento de IETF (Internet Engineering Task Force) RFC 2478 The Simple and Protected GSS-API Negotiation Mechanism

Configurar el servidor iSeries para utilizar IBM JGSS

Cómo configurar el servidor iSeries para emplear JGSS depende de la versión de Java 2 Software Development Kit (J2SDK) que se ejecute en el servidor. Para obtener más información sobre cómo configurar el servidor iSeries para emplear JGSS, utilice los enlaces siguientes:

- Utilizar JGSS con J2SDK, versión 1.3
- Utilizar JGSS con J2SDK, versión 1.4
- Configurar JGSS para emplear el proveedor iSeries JGSS nativo

Configurar el servidor iSeries para utilizar JGSS con J2SDK, versión 1.3: Si utiliza Java 2 Software Development Kit (J2SDK), versión 1.3 en el servidor iSeries, debe preparar y configurar el servidor para utilizar JGSS. La configuración por omisión utiliza el proveedor Java JGSS puro.

Requisitos de software: Para utilizar JGSS con J2SDK, versión 1.3, el servidor debe tener instalado JAAS (Servicio de autenticación y autorización Java) 1.3.

Configurar el servidor para utilizar JGSS: Para configurar el servidor a fin de utilizar JGSS con J2SDK, versión 1.3, añada al directorio de ampliaciones un enlace simbólico para el archivo ibmjgssprovider.jar. El archivo ibmjgssprovider.jar contiene las clases de JGSS y el proveedor Java JGSS puro. La adición del enlace simbólico permite al cargador de clases de ampliación cargar el archivo ibmjgssprovider.jar.

Añadir el enlace simbólico

Para añadir el enlace simbólico, en una línea de mandatos de iSeries, escriba el mandato siguiente (en una sola línea) y pulse **INTRO**:

```
ADDLNK OBJ('/QIBM/ProdData/OS400/Java400/ext/ibmjgssprovider.jar')
NEWLNK('/QIBM/ProdData/Java400/jdk13/lib/ext/ibmjgssprovider.jar')
```

Nota: la política de Java 1.3 por omisión del servidor iSeries otorga los permisos adecuados a JGSS. Si tiene previsto crear su propio archivo java.policy, consulte la sección acerca de los permisos de JVM para ver una lista de los permisos que se pueden otorgar a ibmjgssprovider.jar.

Cambiar los proveedores JGSS: Tras configurar el servidor para utilizar JGSS, que utiliza el proveedor Java puro como valor por omisión, puede configurar JGSS para que emplee el proveedor iSeries JGSS nativo. A continuación, tras configurar JGSS para utilizar el proveedor nativo, puede conmutar fácilmente entre los dos proveedores. Para obtener más información, consulte los siguientes temas:

- Proveedores JGSS
- Configurar JGSS para emplear el proveedor iSeries JGSS nativo

Gestores de seguridad: Si ejecuta la aplicación IBM JGSS con un gestor de seguridad Java habilitado, consulte Utilizar un gestor de seguridad.

Configurar JGSS para emplear el proveedor iSeries JGSS nativo: IBM JGSS utiliza por omisión el proveedor Java puro. Tiene la posibilidad de emplear el proveedor iSeries JGSS nativo. Para obtener más información sobre los distintos proveedores, consulte Proveedores JGSS.

Requisitos de software: El proveedor iSeries JGSS nativo debe poder acceder a las clases de IBM Toolbox para Java. Si desea obtener instrucciones sobre cómo acceder a IBM Toolbox para Java, consulte Permitir al proveedor iSeries JGSS nativo acceder a IBM Toolbox para Java.

Asegúrese de que ha configurado el servicio de autenticación de red. Para obtener más información, consulte la sección sobre el servicio de autenticación de red.

Especificar el proveedor iSeries JGSS nativo: Antes de poder utilizar el proveedor iSeries JGSS nativo con J2SDK, versión 1.3, compruebe que ha configurado el servidor para utilizar JGSS. Para obtener más información, consulte Configurar el servidor iSeries para utilizar JGSS con J2SDK, versión 1.3. Si emplea J2SDK, versión 1.4, JGSS ya está configurado.

Nota: en las instrucciones siguientes, ${java.home} indica la vía de acceso de la ubicación de la versión de Java que utiliza en el servidor. Por ejemplo, si utiliza J2SDK, versión 1.4, ${java.home} es /QIBM/ProdData/Java400/jdk14. No olvide sustituir ${java.home} en los mandatos por la vía de acceso real del directorio inicial de Java.

Para configurar JGSS a fin de emplear el proveedor iSeries JGSS nativo, lleve a cabo estas tareas:

- Añadir al directorio de ampliaciones un enlace simbólico para el archivo JAR del proveedor iSeries nativo (page 213)
- Añadir el proveedor iSeries JGSS nativo a la lista de proveedores de seguridad del archivo java.security (page 213)

Añadir un enlace simbólico

Para añadir un enlace simbólico al directorio de ampliaciones para el archivo ibmjgssiseriesprovider.jar, en una línea de mandatos de iSeries, escriba el mandato siguiente (en una sola línea) y pulse **INTRO**:

```
ADDLNK OBJ('/QIBM/ProdData/OS400/Java400/ext/ibmjgssiseriesprovider.jar')
NEWLNK('${java.home}/lib/ext/ibmjgssiseriesprovider.jar')
```

Tras añadir un enlace simbólico al directorio de ampliaciones para el archivo ibmjgssiseriesprovider.jar, el cargador de clases de ampliación cargará el archivo JAR.

Añadir el proveedor a la lista de proveedores de seguridad

Añada el proveedor nativo a la lista de proveedores de seguridad del archivo java.security.

1. Abra ${java.home}/lib/security/java.security para editarlo.
2. Busque la lista de proveedores de seguridad. Debe encontrarse cerca del principio del archivo java.security y debe tener un aspecto parecido al siguiente:

```
     security.provider.1=sun.security.provider.Sun
security.provider.2=com.sun.rsajca.Provider
     security.provider.3=com.ibm.crypto.provider.IBMJCE
     security.provider.4=com.ibm.security.jgss.IBMJGSSProvider
```

3. Añada el proveedor iSeries JGSS nativo a la lista de proveedores de seguridad antes del proveedor Java original. Dicho de otro modo, añada com.ibm.iseries.security.jgss.IBMJGSSiSeriesProvider a la lista con un número inferior al de com.ibm.jgss.IBMJGSSProvider y, a continuación, actualice la posición de IBMJGSSProvider. Por ejemplo:

```
     security.provider.1=sun.security.provider.Sun
security.provider.2=com.sun.rsajca.Provider
     security.provider.3=com.ibm.crypto.provider.IBMJCE
     security.provider.4=com.ibm.iseries.security.jgss.IBMJGSSiSeriesProvider
     security.provider.5=com.ibm.security.jgss.IBMJGSSProvider
```

   Observe que IBMJGSSiSeriesProvider ha pasado a ser la cuarta entrada de la lista y IBMJGSSProvider se ha convertido en la quinta entrada. Asimismo, compruebe que los números de entrada de la lista de proveedores de seguridad son secuenciales y que cada entrada aumenta el número de entrada en un solo número.
4. Guarde y cierre el archivo java.security.

Configurar el servidor iSeries para utilizar JGSS con J2SDK, versión 1.4: Si utiliza Java 2 Software Development Kit (J2SDK), versión 1.4 en el servidor iSeries, JGSS ya está configurado. La configuración por omisión utiliza el proveedor Java JGSS puro.

Cambiar los proveedores JGSS: Puede configurar JGSS a fin de emplear el proveedor iSeries JGSS nativo en lugar del proveedor Java JGSS puro. A continuación, tras configurar JGSS para utilizar el proveedor nativo, puede conmutar fácilmente entre los dos proveedores. Para obtener más información, consulte los siguientes temas:

- Proveedores JGSS
- Configurar JGSS para emplear el proveedor iSeries JGSS nativo

Gestores de seguridad: Si ejecuta la aplicación JGSS con un gestor de seguridad Java habilitado, consulte Utilizar un gestor de seguridad.

Proveedores JGSS: IBM JGSS incluye un proveedor iSeries JGSS nativo y un proveedor Java JGSS puro. El proveedor que elija utilizar dependerá de las necesidades de la aplicación.

El proveedor Java JGSS puro presenta las características siguientes:

- Garantiza el mayor nivel de portabilidad de la aplicación
- Funciona con la interfaz de inicio de sesión de Kerberos JAAS opcional
- Es compatible con las herramientas de gestión de credenciales de Kerberos Java

El proveedor iSeries JGSS nativo presenta las características siguientes:

- Utiliza las bibliotecas Kerberos de iSeries nativas
- Es compatible con las herramientas de gestión de credenciales de Kerberos Qshell
- Las aplicaciones JGSS se ejecutan con mayor rapidez

Nota: ambos proveedores JGSS cumplen la especificación GSS-API y, por consiguiente, son compatibles entre sí. En otras palabras, una aplicación que utiliza el proveedor Java JGSS puro puede interactuar con una aplicación que emplea el proveedor iSeries JGSS nativo.

Cambiar el proveedor JGSS: **Nota:** si el servidor ejecuta J2SDK, versión 1.3, antes de cambiar al proveedor iSeries JGSS nativo, compruebe que ha configurado el servidor para utilizar JGSS. Para obtener más información, consulte los siguientes temas:

- Configurar el servidor iSeries para utilizar JGSS con J2SDK, versión 1.3
- Configurar JGSS para utilizar el proveedor JGSS nativo

Puede cambiar fácilmente el proveedor JGSS mediante una de las opciones siguientes:

- Edite la lista de proveedores de seguridad de `${java.home}/lib/security/java.security`.

  Nota: ${java.home} indica la vía de acceso de la ubicación de la versión de Java que utiliza en el servidor. Por ejemplo, si utiliza J2SDK, versión 1.3, ${java.home} es /QIBM/ProdData/Java400/jdk13.
- Especifique el nombre del proveedor en la aplicación JGSS mediante GSSManager.addProviderAtFront() o GSSManager.addProviderAtEnd(). Para obtener más información, consulte el javadoc de GSSManager.

Utilizar un gestor de seguridad: Si ejecuta la aplicación JGSS con un gestor de seguridad Java habilitado, debe comprobar que la aplicación y JGSS tienen los permisos necesarios. Para obtener más información sobre los permisos que necesita para utilizar JGSS, consulte los temas siguientes:

- Permisos de JVM
- Comprobaciones de permisos de JAAS

Permisos de JVM: Además de las comprobaciones de control de acceso efectuadas por JGSS, la máquina virtual Java (JVM) lleva a cabo comprobaciones de autorización al acceder a una variada gama de recursos, tales como archivos, propiedades Java, paquetes y sockets.

Para obtener más información sobre cómo utilizar los permisos de JVM, consulte la información sobre permisos en Java 2 SDK

.

La lista siguiente indica los permisos necesarios al utilizar las funciones de JAAS de JGSS o al emplear JGSS con un gestor de seguridad:

- javax.security.auth.AuthPermission ″modifyPrincipals″
- javax.security.auth.AuthPermission ″modifyPrivateCredentials″
- javax.security.auth.AuthPermission ″getSubject″
- javax.security.auth.PrivateCredentialPermission
  ″javax.security.auth.kerberos.KerberosKey javax.security.auth.kerberos.KerberosPrincipal \″*\″″, ″read″
- javax.security.auth.PrivateCredentialPermission
  ″javax.security.auth.kerberos.KerberosTicket javax.security.auth.kerberos.KerberosPrincipal \″*\″″, ″read″
- java.util.PropertyPermission ″com.ibm.security.jgss.debug″, ″read″
- java.util.PropertyPermission ″DEBUG″, ″read″
- java.util.PropertyPermission ″java.home″, ″read″
- java.util.PropertyPermission ″java.security.krb5.conf″, ″read″
- java.util.PropertyPermission ″java.security.krb5.kdc″, ″read″
- java.util.PropertyPermission ″java.security.krb5.realm″, ″read″
- java.util.PropertyPermission ″javax.security.auth.useSubjectCredsOnly″, ″read″
- java.util.PropertyPermission ″user.dir″, ″read″
- java.util.PropertyPermission ″user.home″, ″read″
- java.lang.RuntimePermission ″accessClassInPackage.sun.security.action″
- java.security.SecurityPermission ″putProviderProperty.IBMJGSSProvider″

Comprobaciones de permisos de JAAS: IBM JGSS efectúa comprobaciones de permisos en tiempo de ejecución cuando el programa habilitado para JAAS utiliza credenciales y accede a servicios. Puede inhabilitar esta función de JAAS opcional estableciendo la propiedad Java javax.security.auth.useSubjectCredsOnly en false. Además, JGSS lleva a cabo comprobaciones de permisos únicamente cuando la aplicación se ejecuta con un gestor de seguridad.

JGSS realiza comprobaciones de permisos en relación con la política Java que está en vigor para el contexto de control de acceso actual. JGSS lleva a cabo las siguientes comprobaciones de permisos específicos:

- javax.security.auth.kerberos.DelegationPermission
- javax.security.auth.kerberos.ServicePermission

Comprobación DelegationPermission: DelegationPermission permite el control por parte de la política de seguridad del uso de las funciones de reenvío de tickets y servicio proxy de Kerberos. Mediante estas funciones, un cliente puede permitir que un servicio actúe en nombre del cliente.

DelegationPermission toma dos argumentos, en el orden siguiente:

1. El principal de subordinado, que es el nombre del principal de servicio que actúa en nombre del cliente y bajo su autorización.
2. El nombre del servicio que el cliente desea permitir que el principal de subordinado utilice.

Ejemplo: utilizar la comprobación DelegationPermission

En el ejemplo siguiente, superSecureServer es el principal de subordinado y krbtgt/REALM.IBM.COM@REALM.IBM.COM es el servicio que se desea permitir que superSecureServer utilice en nombre del cliente. En este caso, el servicio es el ticket de otorgación de tickets del cliente, lo que significa que superSecureServer puede obtener un ticket para cualquier servicio en nombre del cliente.

```
permission javax.security.auth.kerberos.DelegationPermission
     "\"superSecureServer/host.ibm.com@REALM.IBM.COM\"
     \"krbtgt/REALM.IBM.COM@REALM.IBM.COM\"";
```

En el ejemplo anterior, DelegationPermission otorga al cliente permiso para obtener un nuevo ticket de otorgación de tickets del centro de distribución de claves (KDC) que sólo superSecureServer puede utilizar. Una vez que el cliente ha enviado el nuevo ticket de otorgación de tickets a superSecureServer, superSecureServer tiene la posibilidad de actuar en nombre del cliente.

El ejemplo siguiente permite al cliente obtener un nuevo ticket que permita a superSecureServer acceder únicamente al servicio ftp en nombre del cliente:

```
permission javax.security.auth.kerberos.DelegationPermission
     "\"superSecureServer/host.ibm.com@REALM.IBM.COM\"
     \"ftp/ftp.ibm.com@REALM.IBM.COM\"";
```

Para obtener más información, consulte la clase javax.security.auth.kerberos.DelegationPermission en la documentación de J2SDK

del sitio Web de Sun.

Comprobación ServicePermission: Las comprobaciones ServicePermission restringen el uso de credenciales para la iniciación y la aceptación del contexto. Un iniciador de contexto debe tener permiso para iniciar un contexto. Del mismo modo, un aceptante de contexto debe tener permiso para aceptar un contexto.

Ejemplo: utilizar la comprobación ServicePermission

El ejemplo siguiente permite al lado del cliente iniciar un contexto con el servicio ftp otorgando permiso al cliente:

```
permission javax.security.auth.kerberos.ServicePermission
     "ftp/host.ibm.com@REALM.IBM.COM", "initiate";
```

El ejemplo siguiente permite al lado del servidor acceder a la clave secreta para el servicio ftp y utilizarla otorgando permiso al servidor:

```
permission javax.security.auth.kerberos.ServicePermission
     "ftp/host.ibm.com@REALM.IBM.COM", "accept";
```

Para obtener más información, consulte la clase javax.security.auth.kerberos.ServicePermission en la documentación de J2SDK

del sitio Web de Sun.

Ejecutar aplicaciones IBM JGSS

La API IBM Java Generic Security Service (JGSS) 1.0 ampara las aplicaciones seguras de las complejidades y peculiaridades de los distintos mecanismos de seguridad subyacentes. JGSS utiliza las funciones que proporcionan el servicio de autenticación y autorización Java (JAAS) y la ampliación de criptografía Java (JCE) de IBM.

Entre las funciones de JGSS destacan:

- Autenticación de identidades
- Integridad y confidencialidad de mensajes
- Interfaz de inicio de sesión de Kerberos JAAS opcional y comprobaciones de autorización

Para obtener más información sobre cómo ejecutar aplicaciones JGSS, consulte los temas siguientes:

Obtener credenciales de Kerberos
Descubra cómo obtener credenciales de Kerberos y crear claves secretas. Aprenda a utilizar JAAS para efectuar inicios de sesión de Kerberos y comprobaciones de autorización y vea una lista de los permisos de JAAS que requiere la máquina virtual Java (JVM).

Archivos de configuración y política
Obtenga información sobre los distintos tipos de archivos de soporte que necesita para ejecutar JGSS, tales como los archivos de configuración, los archivos de política, el archivo de propiedades de seguridad maestro de Java y la antememoria de credenciales.

Depuración
Descubra cómo utilizar la depuración de JGSS para categorizar y visualizar mensajes de depuración de gran utilidad.

Ejemplos de JGSS
Utilice los programas de ejemplo para probar y verificar la configuración de JGSS. La documentación de ejemplo contiene el código fuente Java, las instrucciones para ejecutar los ejemplos, los archivos de configuración y política y mucho más.

Obtener credenciales de Kerberos y crear claves secretas: GSS-API no define ningún método para obtener credenciales. Es por ello que el mecanismo Kerberos de IBM JGSS requiere que el usuario obtenga credenciales de Kerberos mediante uno de los métodos siguientes:

- Herramientas Kinit y Ktab
- Interfaz de inicio de sesión de Kerberos JAAS opcional

Herramientas Kinit y Ktab: El proveedor JGSS que elija determinará qué herramientas utilizará para obtener las credenciales de Kerberos y claves secretas.

Con el proveedor Java JGSS puro: Si emplea el proveedor Java JGSS puro, utilice las herramientas Kinit y Ktab de IBM JGSS para obtener credenciales y claves secretas. Las herramientas Kinit y Ktab utilizan interfaces de línea de mandatos y proporcionan opciones similares a las de otras versiones.

- Puede obtener credenciales de Kerberos mediante la herramienta Kinit. Esta herramienta establece el contacto con el centro de distribución de Kerberos (KDC) y obtiene un ticket de otorgación de tickets (TGT). El TGT permite acceder a otros servicios habilitados para Kerberos, entre ellos los que utilizan GSS-API.
- Un servidor puede obtener una clave secreta mediante la herramienta Ktab. JGSS almacena la clave secreta en el archivo de tabla de claves del servidor. Consulte la documentación para Java de Ktab a fin de obtener más información.

La aplicación también puede emplear la interfaz de inicio de sesión de JAAS a fin de obtener tickets TGT y claves secretas. Para obtener más información, consulte lo siguiente:

- Javadoc de Kinit
- Javadoc de Ktab
- Interfaz de inicio de sesión de JAAS

Con el proveedor iSeries JGSS nativo: Si emplea el proveedor iSeries JGSS nativo, utilice los programas de utilidad de Qshell kinit y klist. Para obtener más información, consulte Programas de utilidad para credenciales de Kerberos y tablas de claves.

Interfaz de inicio de sesión de Kerberos JAAS: IBM JGSS presenta una interfaz de inicio de sesión de Kerberos del servicio de autenticación y autorización Java (JAAS). Puede inhabilitar esta función estableciendo la propiedad Java javax.security.auth.useSubjectCredsOnly en false.

Nota: Aunque el proveedor Java JGSS puro puede emplear la interfaz de inicio de sesión, el proveedor iSeries JGSS nativo no puede.

Para obtener más información sobre JAAS, consulte Servicio de autenticación y autorización Java.

Permisos de JAAS y JVM: Si utiliza un gestor de seguridad, debe comprobar que la aplicación y JGSS tienen los permisos de JVM y JAAS necesarios. Para obtener más información, consulte Utilizar un gestor de seguridad.

Opciones del archivo de configuración de JAAS: La interfaz de inicio de sesión requiere un archivo de configuración de JAAS que especifique com.ibm.security.auth.module.Krb5LoginModule como el módulo de inicio de sesión que se empleará. La tabla siguiente muestra las opciones que admite Krb5LoginModule. Tenga en cuenta que las opciones no son sensibles a las mayúsculas y minúsculas.

Nombre de opción	Valor	Valor por omisión	Descripción
principal	<serie>	Ninguno; se solicita.	Nombre de principal de Kerberos
credsType	initiator | acceptor | both	initiator	Tipo de credencial de JGSS
forwardable	true | false	false	Si debe adquirirse un ticket de otorgación de tickets (TGT) reenviable
proxiable	true | false	false	Si debe adquirirse un TGT que admite proxy
useCcache	<URL>	No utilizar la antememoria de credenciales	Recuperar el TGT de la antememoria de credenciales especificada
useKeytab	<URL>	No utilizar la tabla de claves	Recuperar la clave secreta de la tabla de claves especificada
useDefaultCcache	true | false	No utilizar la antememoria de credenciales por omisión	Recuperar el TGT de la antememoria de credenciales por omisión
useDefaultKeytab	true | false	No utilizar la tabla de claves por omisión	Recuperar la clave secreta de la tabla de claves especificada

Para ver un sencillo ejemplo de cómo utilizar Krb5LoginModule, consulte el archivo de configuración de inicio de sesión de JAAS de ejemplo.

Incompatibilidades de opciones

Algunas opciones de Krb5LoginModule, sin incluir el nombre de principal, son incompatibles entre ellas, lo que significa que no se pueden especificar juntas. La tabla siguiente indica las opciones de módulo de inicio de sesión compatibles e incompatibles.

Los indicadores de la tabla describen la relación entre las dos opciones asociadas:

- X = Incompatible
- N/A = Combinación no aplicable

- Blanco = Compatible

opción de Krb5LoginModule	credsType initiator	credsType acceptor	credsType both	forward	proxy	use Ccache	use Keytab	useDefault Ccache	useDefault Keytab
credsType=initiator		N/A	N/A				X		X
credsType=acceptor	N/A		N/A	X	X	X		X	
credsType=both	N/A	N/A							
forwardable		X				X	X	X	X
proxiable		X				X	X	X	X
useCcache		X		X	X		X	X	X
useKeytab	X			X	X	X		X	X
useDefaultCcache		X		X	X	X	X		X
useDefaultKeytab	X			X	X	X	X	X	

Opción de nombre de principal: Puede especificar un nombre de principal junto con cualquier otra opción. Si no especifica un nombre de principal, Krb5LoginModule puede solicitar al usuario un nombre de principal. Krb5LoginModule solicitará o no esta información al usuario en función de las demás opciones especificadas. Para obtener más información, consulte "Solicitar el nombre de principal y la contraseña".

Formato del nombre de principal de servicio

Debe emplear uno de los formatos siguientes para especificar un nombre de principal de servicio:

- <nombre_servicio> (por ejemplo, superSecureServer)
- <nombre_servicio>@<sistema_principal> (por ejemplo, superSecureServer@myhost)

En el segundo formato, <sistema_principal> es el nombre de sistema principal de la máquina donde reside el servicio. Aunque no está obligado a ello, puede utilizar un nombre de sistema principal totalmente calificado.

Nota: JAAS reconoce determinados caracteres como delimitadores. Si emplea alguno de los caracteres siguientes en una serie de JAAS (como un nombre de principal), escriba el carácter entre comillas:

```
_  (subrayado)
: (dos puntos)/ (barra inclinada)     \  (barra inclinada invertida)
```

Solicitar el nombre de principal y la contraseña: Las opciones que especifique en el archivo de configuración de JAAS determinarán si el inicio de sesión de Krb5LoginModule será o no interactivo.

- Un inicio de sesión no interactivo no solicita ninguna información.
- Un inicio de sesión interactivo solicita el nombre de principal, la contraseña o ambos.

Inicios de sesión no interactivos

El inicio de sesión se efectuará de modo no interactivo si especifica el tipo de credencial initiator (`credsType=initiator`) y lleva a cabo una de las acciones siguientes:

- Especificar la opción useCcache
- Especificar la opción useDefaultCcache en true

El inicio de sesión también se efectuará de modo no interactivo si especifica el tipo de credencial acceptor o both (`credsType=acceptor` o `credsType=both`) y lleva a cabo una de las acciones siguientes:

- Especificar la opción useKeytab
- Especificar la opción useDefaultKeytab en true

Inicios de sesión interactivos

Otras configuraciones hacen que el módulo de inicio de sesión solicite un nombre de principal y una contraseña a fin de obtener un TGT de un KDC de Kerberos. El módulo de inicio de sesión sólo solicita una contraseña si se especifica la opción principal.

Los inicios de sesión interactivos requieren que la aplicación especifique com.ibm.security.auth.callback.Krb5CallbackHandler como manejador de retornos de llamada al crear el contexto de inicio de sesión. El manejador de retornos de llamada es el encargado de solicitar la entrada.

Opción de tipo de credencial*:* Si se requiere que el tipo de credencial sea tanto initiator como acceptor (`credsType=both`), Krb5LoginModule obtiene un TGT y una clave secreta. El módulo de inicio de sesión utiliza el TGT para iniciar contextos y la clave secreta para aceptar contextos. El archivo de configuración de JAAS debe contener suficiente información para permitir al módulo de inicio de sesión adquirir los dos tipos de credenciales.

Para los tipos de credencial acceptor y both, el módulo de inicio de sesión asume un principal de servicio.

Archivos de configuración y política: JGSS y JAAS dependen de varios archivos de configuración y política. Debe editar estos archivos para que se ajusten al entorno y la aplicación que utiliza. Si no emplea JAAS con JGSS, puede omitir los archivos de configuración y política de JAAS.

- "Archivo de configuración de Kerberos"
- "Archivo de configuración de JAAS"
- "Archivo de política de JAAS" en la página 221
- "Archivo de propiedades de seguridad maestro de Java" en la página 221
- "Antememoria de credenciales y tabla de claves de servidor" en la página 222

Nota: en las instrucciones siguientes, ${java.home} indica la vía de acceso de la ubicación de la versión de Java que utiliza en el servidor. Por ejemplo, si utiliza J2SDK, versión 1.4, ${java.home} es /QIBM/ProdData/Java400/jdk14. No olvide sustituir ${java.home} en los valores de propiedades por la vía de acceso real del directorio inicial de Java.

Archivo de configuración de Kerberos: IBM JGSS requiere un archivo de configuración de Kerberos. El nombre por omisión y la ubicación del archivo de configuración de Kerberos dependen del sistema operativo que se utilice. JGSS utiliza el orden siguiente para buscar el archivo de configuración por omisión:

1. El archivo al que hace referencia la propiedad Java java.security.krb5.conf
2. ${java.home}/lib/security/krb5.conf
3. c:\winnt\krb5.ini en las plataformas Microsoft Windows
4. /etc/krb5/krb5.conf en las plataformas Solaris
5. /etc/krb5.conf en otras plataformas Unix

Archivo de configuración de JAAS: El uso de la función de inicio de sesión de JAAS requiere un archivo de configuración de JAAS. Puede especificar el archivo de configuración de JAAS estableciendo una de las propiedades siguientes:

- La propiedad del sistema Java java.security.auth.login.config
- La propiedad de seguridad login.config.url.<entero> en el archivo ${java.home}/lib/security/java.security

Para obtener más información, consulte el sitio Web de Sun Java Authentication and Authorization Service (JAAS)

.

Archivo de política de JAAS: Al utilizar la implementación de política por omisión, JGSS otorga permisos de JAAS a las entidades anotando los permisos en un archivo de política. Puede especificar el archivo de política de JAAS estableciendo una de las propiedades siguientes:

- La propiedad del sistema Java java.security.policy
- La propiedad de seguridad policy.url.<entero> en el archivo ${java.home}/lib/security/java.security

Si utiliza J2SDK, versión 1.4, la especificación de un archivo de política aparte para JAAS es opcional. El proveedor de política por omisión en J2SDK, versión 1.4, da soporte a las entradas de archivo de política que requiere JAAS.

Para obtener más información, consulte el sitio Web del servicio de autenticación y autorización Java (JAAS) de Sun

.

Archivo de propiedades de seguridad maestro de Java: Una máquina virtual Java (JVM) utiliza muchas propiedades de seguridad importantes que se establecen editando el archivo de propiedades de seguridad maestro de Java. Este archivo, denominado java.security, normalmente se encuentra en el directorio ${java.home}/lib/security del servidor iSeries.

La lista siguiente describe varias propiedades de seguridad relevantes para utilizar JGSS. Utilice las descripciones a modo de guía para editar el archivo java.security.

Nota: cuando corresponde, las descripciones incluyen los valores adecuados que son necesarios para ejecutar los ejemplos de JGSS.

security.provider.<entero>: el proveedor JGSS que desea utilizar. Estáticamente también registra las clases de proveedor criptográfico. IBM JGSS emplea la criptografía y otros servicios de seguridad proporcionados por IBM JCE Provider. Especifique los paquetes sun.security.provider.Sun y com.ibm.crypto.provider.IBMJCE exactamente igual que en el ejemplo siguiente:

```
security.provider.1=sun.security.provider.Sun
security.provider.2=com.ibm.crypto.provider.IBMJCE
```

policy.provider: clase de manejador de política del sistema. Por ejemplo:

```
policy.provider=sun.security.provider.PolicyFile
```

policy.url.<entero>: URL de los archivos de política. Para emplear el archivo de política de ejemplo, incluya una entrada como:

```
policy.url.1=file:/home/user/jgss/config/java.policy
```

login.configuration.provider: clase de manejador de configuración de inicio de sesión de JAAS, como por ejemplo:

```
login.configuration.provider=com.ibm.security.auth.login.ConfigFile
```

auth.policy.provider: clase de manejador de política de control de acceso basado en el principal de JAAS, como por ejemplo:

```
auth.policy.provider=com.ibm.security.auth.PolicyFile
```

login.config.url.<entero>: URL para los archivos de configuración de inicio de sesión de JAAS. Para emplear el archivo de configuración de ejemplo, incluya una entrada parecida a la siguiente:

```
login.config.url.1=file:/home/user/jgss/config/jaas.conf
```

auth.policy.url.<entero>: URL para los archivos de política de JAAS. Puede incluir construcciones basadas en el principal y el origen del código en el archivo de política de JAAS. Para emplear el archivo de política de ejemplo, incluya una entrada como:

```
auth.policy.url.1=file:/home/user/jgss/config/jaas.policy
```

Antememoria de credenciales y tabla de claves de servidor: Un principal de usuario mantiene sus credenciales de Kerberos en una antememoria de credenciales. Un principal de servicio mantiene su clave secreta en una tabla de claves. En el momento de la ejecución, IBM JGSS localiza estas antememorias del modo siguiente:

Antememoria de credenciales de usuario

JGSS utiliza el orden siguiente para localizar la antememoria de credenciales de usuario:

1. El archivo al que hace referencia la propiedad Java KRB5CCNAME
2. El archivo al que hace referencia la variable de entorno KRB5CCNAME
3. /tmp/krb5cc_<uid> en sistemas Unix
4. ${user.home}/krb5cc_${user.name}
5. ${user.home}/krb5cc (si no es posible obtener ${user.name})

Tabla de claves de servidor

JGSS utiliza el orden siguiente para localizar el archivo de tabla de claves de servidor:

1. El valor de la propiedad Java KRB5_KTNAME
2. La entrada default_keytab_name de la stanza libdefaults del archivo de configuración de Kerberos
3. ${user.home}/krb5_keytab

Desarrollar aplicaciones IBM JGSS

Para obtener más información sobre cómo desarrollar aplicaciones IBM JGSS, consulte los temas siguientes:

Procedimiento de programación
Aprenda los pasos necesarios para desarrollar una aplicación JGSS, tales como utilizar señales de transporte, crear los objetos JGSS necesarios, establecer y suprimir un contexto y emplear los servicios por mensaje.

Utilizar JAAS con la aplicación JGSS
Lea cómo habilitar la función de inicio de sesión de Kerberos JAAS de JGSS. La información que se facilita incluye los requisitos para utilizar la función de inicio de sesión y un snippet del código de ejemplo.

Depuración
Descubra cómo utilizar la depuración de JGSS para categorizar y visualizar mensajes de depuración de gran utilidad.

Información de consulta de javadoc de JGSS
Consulte la información de javadoc relacionada con las clases y los métodos del paquete de API org.ietf.jgss y las versiones Java de las herramientas de gestión de credenciales de Kerberos (kinit, ktab y klist).

Ejemplos de JGSS
Utilice los programas de ejemplo para descubrir cómo puede emplear JGSS en las aplicaciones. La documentación de ejemplo contiene el código fuente Java, las instrucciones para ejecutar los ejemplos, los archivos de configuración y política y mucho más.

Para desarrollar aplicaciones JGSS, debe conocer la especificación GSS-API de alto nivel y la especificación de enlaces Java. IBM JGSS 1.0 se basa principalmente en estas especificaciones y las cumple. Consulte los siguientes enlaces para obtener más información.

- RFC 2743: Generic Security Service Application Programming Interface Version 2, Update 1

- RFC 2853: Generic Security Service API Version 2: Java Bindings

Procedimiento de programación de aplicaciones IBM JGSS: Las operaciones de una aplicación JGSS siguen el modelo operativo de GSS-API (Generic Security Service Application Programming Interface). Para obtener información sobre los conceptos importantes para las operaciones de JGSS, consulte Conceptos de JGSS.

Señales de transporte de JGSS: Algunas de las operaciones de JGSS importantes generan señales con el formato de matrices de bytes Java. La aplicación es la encargada de reenviar las señales de un igual JGSS al otro. JGSS no restringe en modo alguno el protocolo que utiliza la aplicación para transportar señales. Las aplicaciones pueden transportar señales de JGSS junto con otros datos de aplicación (es decir, datos que no son de JGSS). Sin embargo, las operaciones de JGSS aceptan y utilizan únicamente señales específicas de JGSS.

Secuencia de operaciones en una aplicación JGSS: Las operaciones de JGSS requieren determinadas construcciones de programación que se deben emplear en el orden indicado a continuación. Cada uno de los pasos se aplica tanto al iniciador como al aceptante.

Nota: la información contiene snippets de código de ejemplo que muestran cómo utilizar las API de JGSS de alto nivel y suponen que la aplicación importa el paquete org.ietf.jgss. Aunque muchas de las API de alto nivel se cargan a posteriori, los snippets sólo muestran los formatos más empleados de esos métodos. Por supuesto, utilice los métodos de las API que mejor se adapten a sus necesidades.

1. Crear un GSSManager
   Una instancia de GSSManager actúa como fábrica para crear otras instancias de objetos JGSS.
2. Crear un GSSName
   Un GSSName representa la identidad de un principal JGSS. Algunas operaciones de JGSS pueden localizar y utilizar un principal por omisión al especificar un GSSName nulo.
3. Crear un GSSCredential
   Un GSSCredential contiene las credenciales del principal específicas del mecanismo.
4. Crear un GSSContext
   Un GSSContext se utiliza para el establecimiento del contexto y los servicios por mensaje posteriores.
5. Seleccionar servicios opcionales en el contexto
   La aplicación debe solicitar explícitamente los servicios opcionales, tales como la autenticación mutua.
6. Establecer el contexto
   El iniciador se autentica para el aceptante. Sin embargo, al solicitar la autenticación mutua, el aceptante se autentica a su vez para el iniciador.
7. Utilizar los servicios por mensaje
   El iniciador y el aceptante intercambian mensajes seguros en el contexto establecido.
8. Suprimir el contexto
   La aplicación suprime un contexto que ya no es necesario.

Crear un GSSManager: La clase abstracta GSSManager actúa como una fábrica para crear los siguientes objetos JGSS:

- GSSName
- GSSCredential
- GSSContext

GSSManager también tiene métodos para determinar los mecanismos de seguridad y tipos de nombres soportados y especificar proveedores JGSS. Utilice el método estático de GSSManager getInstance para crear una instancia del GSSManager por omisión:

```
GSSManager manager = GSSManager.getInstance();
```

Crear un GSSName: GSSName representa la identidad de un principal GSS-API. Un GSSName puede contener numerosas representaciones del principal, una para cada mecanismo subyacente soportado. Un GSSName que contiene una sola representación de nombre se denomina nombre de mecanismo (MN).

GSSManager tiene varios métodos cargados a posteriori para crear un GSSName a partir de una serie o una matriz contigua de bytes. Los métodos interpretan la serie o matriz de bytes según un tipo de nombre especificado. Por lo general, se utilizan los métodos de matriz de bytes de GSSName para volver a formar un nombre exportado. El nombre exportado normalmente es un nombre de mecanismo de tipo GSSName.NT_NOMBRE_EXPORT. Algunos de estos métodos permiten especificar un mecanismo de seguridad con el que se creará el nombre.

Ejemplos: utilizar GSSName: El snippet de código básico siguiente muestra cómo utilizar GSSName.

Nota: especifique series de nombre de servicio de Kerberos como <servicio> o <servicio@sistemaprincipal> donde <servicio> es el nombre del servicio y <sistemaprincipal> es el nombre de sistema principal de la máquina en el que se ejecuta el servicio. Aunque no está obligado a ello, puede utilizar un nombre de sistema principal totalmente calificado. Si omite la parte @<sistemaprincipal> de la serie, GSSName utiliza el nombre de sistema principal local.

```
// Crear GSSName para el usuario foo.
GSSName fooName = manager.createName("foo", GSSName.NT_USER_NAME);

// Crear un nombre de mecanismo Kerberos V5 para el usuario foo.
Oid krb5Mech = new Oid("1.2.840.113554.1.2.2");
GSSName fooName = manager.createName("foo", GSSName.NT_USER_NAME, krb5Mech);

// Crear un nombre de mecanismo a partir de un nombre de no mecanismo mediante
// el método canonicalize de GSSName.
GSSName fooName = manager.createName("foo", GSSName.NT_USER_NAME);
GSSName fooKrb5Name = fooName.canonicalize(krb5Mech);
```

Crear un GSSCredential: Un GSSCredential contiene toda la información criptográfica necesaria para crear un contexto en nombre de un principal y puede contener información de credenciales para varios mecanismos.

GSSManager tiene tres métodos de creación de credenciales. Dos de los métodos toman para los parámetros un GSSName, el tiempo de vida de la credencial, uno o varios mecanismos de los que se obtendrán credenciales y el tipo de uso de credenciales. El tercer método toma sólo un tipo de uso y utiliza los valores por omisión para los demás parámetros. Al especificar un mecanismo nulo también se utiliza el mecanismo por omisión. Al especificar una matriz nula de mecanismos, el método devuelve credenciales para el conjunto por omisión de mecanismos.

Nota: como IBM JGSS sólo da soporte al mecanismo Kerberos V5, éste es el mecanismo por omisión.

La aplicación puede crear sólo uno de los tres tipos de credenciales (*initiate, accept* o *initiate and accept*) cada vez.

- Un iniciador de contexto crea credenciales *initiate*.

- Un aceptante crea credenciales *accept*.
- Un aceptante que también se comporta como iniciador crea credenciales *initiate and accept* .

Ejemplos: obtener credenciales

El ejemplo siguiente obtiene las credenciales por omisión para un iniciador:

```
GSSCredentials fooCreds = manager.createCredentials(GSSCredential.INITIATE)
```

El ejemplo siguiente obtiene las credenciales de Kerberos V5 para el iniciador foo con el periodo de validez por omisión:

```
GSSCredential fooCreds = manager.createCredential(fooName, GSSCredential.DEFAULT_LIFETIME,
                                                  krb5Mech,GSSCredential.INITIATE);
```

El ejemplo siguiente obtiene una credencial de aceptante con todos los valores por omisión:

```
GSSCredential serverCreds = manager.createCredential(null, GSSCredential.DEFAULT_LIFETIME,
                                                     (Oid)null, GSSCredential.ACCEPT);
```

Crear GSSContext: IBM JGSS da soporte a dos métodos proporcionados por GSSManager para crear un contexto:

- Un método empleado por el iniciador del contexto
- Un método empleado por el aceptante

Nota: GSSManager proporciona un tercer método para crear un contexto que supone volver a crear contextos exportados anteriormente. Sin embargo, como el mecanismo de Kerberos V5 de IBM JGSS no admite el uso de contextos exportados, IBM JGSS no da soporte a este método.

La aplicación no puede emplear un contexto de iniciador para la aceptación del contexto, ni puede utilizar un contexto de aceptante para la iniciación del contexto. Ambos métodos soportados para crear un contexto requieren una credencial como entrada. Si el valor de la credencial es nulo, JGSS utiliza la credencial por omisión.

Ejemplos: utilizar GSSContext

El ejemplo siguiente crea un contexto con el que el principal (foo) puede iniciar un contexto con el igual (superSecureServer) en el sistema principal (securityCentral). El ejemplo especifica el igual como superSecureServer@securityCentral. El contexto creado es válido durante el periodo por omisión:

```
GSSName serverName = manager.createName("superSecureServer@securityCentral",
                                        GSSName.NT_HOSTBASED_SERVICE, krb5Mech);
GSSContext fooContext = manager.createContext(serverName, krb5Mech, fooCreds,
                                              GSSCredential.DEFAULT_LIFETIME);
```

El ejemplo siguiente crea un contexto para superSecureServer a fin de aceptar los contextos iniciados por cualquier igual:

```
GSSContext serverAcceptorContext = manager.createContext(serverCreds);
```

Tenga en cuenta que la aplicación puede crear y utilizar de forma simultánea ambos tipos de contextos.

Solicitar servicios de seguridad opcionales: La aplicación puede solicitar cualquiera de los diversos servicios de seguridad opcionales. IBM JGSS da soporte a los siguientes servicios opcionales:

- Delegación
- Autenticación mutua
- Detección de reproducción
- Detección fuera de secuencia
- Confidencialidad por mensaje disponible

- Integridad por mensaje disponible

Para solicitar un servicio opcional, la aplicación debe solicitarlo explícitamente empleando el método de solicitud adecuado en el contexto. Sólo un iniciador puede solicitar estos servicios opcionales. El iniciador debe efectuar la petición antes de que comience el establecimiento del contexto.

Para obtener más información sobre los servicios opcionales, consulte la información acerca del soporte de servicios opcionales en el documento de IETF (Internet Engineering Task Force) RFC 2743 Generic Security Services Application Programming Interface Version 2, Update 1

.

Ejemplo: solicitar servicios opcionales

En el ejemplo siguiente, un contexto (fooContext) efectúa solicitudes para habilitar los servicios de autenticación mutua y delegación:

```
fooContext.requestMutualAuth(true);
fooContext.requestCredDeleg(true);
```

Establecer el contexto: Los dos iguales que se comunican deben establecer un contexto de seguridad en el que pueden utilizar servicios por mensaje. El iniciador llama a initSecContext() en su contexto, que devuelve una señal a la aplicación del iniciador. La aplicación del iniciador transporta la señal de contexto a la aplicación del aceptante. El aceptante llama a acceptSecContext() en su contexto especificando la señal de contexto recibida del iniciador. Según el mecanismo subyacente y los servicios opcionales que ha seleccionado el iniciador, acceptSecContext() puede generar una señal que la aplicación del aceptante tiene que reenviar a la aplicación del iniciador. A continuación, la aplicación del iniciador utiliza la señal recibida para llamar a initSecContext() una vez más.

Una aplicación puede efectuar varias llamadas a GSSContext.initSecContext() y GSSContext.acceptSecContext(). Asimismo, una aplicación puede intercambiar varias señales con un igual durante el establecimiento del contexto. Por consiguiente, el método habitual para establecer un contexto utiliza un bucle para llamar a GSSContext.initSecContext() o GSSContext.acceptSecContext() hasta que las aplicaciones establecen el contexto.

Ejemplo: establecer el contexto

El ejemplo siguiente muestra el lado del iniciador (foo) del establecimiento del contexto:

```
byte array[] inToken = null; // La señal de entrada es nula para la primera llamada
int inTokenLen = 0;

do {
     byte[] outToken = fooContext.initSecContext(inToken, 0, inTokenLen);

     if (outToken != null) {
         send(outToken); // señal de transporte al aceptante
     }

     if( !fooContext.isEstablished()) {
         inToken = receive(); // señal de recepción del aceptante
           inTokenLen = inToken.length;
     }
 } while (!fooContext.isEstablished());
```

El ejemplo siguiente muestra el lado del aceptante del establecimiento del contexto:

```
// El código del aceptante para establecer el contexto puede ser:
do {
     byte[] inToken = receive(); // señal de recepción del iniciador
```

```
        byte[] outToken =
            serverAcceptorContext.acceptSecContext(inToken, 0, inToken.length);

        if (outToken != null) {
            send(outToken); // señal de transporte al iniciador
        }
    } while (!serverAcceptorContext.isEstablished());
```

Utilizar los servicios por mensaje: Tras establecer un contexto de seguridad, dos iguales que se comunican pueden intercambiar mensajes seguros en el contexto establecido. Cualquiera de los dos iguales puede originar un mensaje seguro, independientemente de si ha actuado como iniciador o como aceptante al establecer el contexto. Para que el mensaje sea seguro, IBM JGSS calcula un código de integridad de mensaje (MIC) criptográfico a partir del mensaje. De modo opcional, IBM JGSS puede hacer que el mecanismo de Kerberos V5 cifre el mensaje para ayudar a garantizar la privacidad.

Enviar mensajes: IBM JGSS proporciona dos conjuntos de métodos para proteger mensajes: wrap() y getMIC().

Utilizar wrap()

El método wrap lleva a cabo las acciones siguientes:

- Calcula un MIC
- Cifra el mensaje (opcional)
- Devuelve una señal

La aplicación que efectúa la llamada utiliza la clase MessageProp junto con GSSContext para especificar si debe aplicarse cifrado al mensaje.

La señal devuelta contiene tanto el MIC como el texto del mensaje. El texto del mensaje es texto cifrado (en el caso de un mensaje cifrado) o el texto plano original (en el caso de los mensajes no cifrados).

Utilizar getMIC()

El método getMIC lleva a cabo las acciones siguientes pero no puede cifrar el mensaje:

- Calcula un MIC
- Devuelve una señal

La señal devuelta contiene únicamente el MIC calculado y no incluye el mensaje original. Por consiguiente, además de transportar la señal de MIC al igual, es preciso hacer que de algún modo el igual tenga conocimiento del mensaje original para que pueda verificar su MIC.

Ejemplo: utilizar los servicios por mensaje para enviar un mensaje

El ejemplo siguiente muestra cómo un igual (foo) puede envolver un mensaje para la entrada a otro igual (superSecureServer):

```
byte[] message = "Ready to roll!".getBytes();
MessageProp mprop = new MessageProp(true); // foo quiere el mensaje cifrado
byte[] wrappedMessage =
    fooContext.wrap(message, 0, message.length, mprop);
send(wrappedMessage); // transferir el mensaje envuelto a superSecureServer

// Así puede obtener superSecureServer un MIC para la entrega a foo:
byte[] message = "You bet!".getBytes();
MessageProp mprop = null;  // superSecureServer está satisfecho con
                           // la calidad de protección por omisión

byte[] mic =
```

```
        serverAcceptorContext.getMIC(message, 0, message.length, mprop);
    send(mic);
    // enviar el MIC a foo. foo también necesita el mensaje original para verificar el MIC
```

Recibir mensajes: El receptor de un mensaje envuelto utiliza unwrap() para descodificar el mensaje. El método unwrap lleva a cabo las acciones siguientes:

- Verifica el MIC criptográfico incorporado en el mensaje
- Devuelve el mensaje original a partir del cual el emisor ha calculado el MIC

Si el emisor ha cifrado el mensaje, unwrap() descifra el mensaje antes de verificar el MIC y, a continuación, devuelve el mensaje de texto plano original. El receptor de una señal de MIC utiliza verifyMIC() para verificar el MIC a partir de un mensaje determinado.

Las aplicaciones de iguales utilizan su propio protocolo para entregarse señales de mensaje y contexto de JGSS. Las aplicaciones de iguales también deben definir un protocolo para determinar si la señal es un MIC o un mensaje envuelto. Por ejemplo, una parte de este protocolo puede ser tan sencilla (y rígida) como el empleado por las aplicaciones SASL (Simple Authentication and Security Layer). El protocolo SASL especifica que el aceptante del contexto siempre es el primer igual en enviar una señal por mensaje (envuelto) tras el establecimiento del contexto.

Para obtener más información, consulte Simple Authentication and Security Layer (SASL)

.

Ejemplo: utilizar los servicios por mensaje para recibir un mensaje

Los ejemplos siguientes muestran cómo un igual (superSecureServer) desenvuelve la señal envuelta que ha recibido de otro igual (foo):

```
    MessageProp mprop = new MessageProp(false);

    byte[] plaintextFromFoo =
        serverAcceptorContext.unwrap(wrappedTokenFromFoo, 0,
                                     wrappedTokenFromFoo.length, mprop);

    // superSecureServer ahora puede examinar mprop para determinar las propiedades del mensaje
    // (por ejemplo, si se ha cifrado el mensaje) que foo ha aplicado.

    // foo verifica el MIC recibido de superSecureServer:

MessageProp mprop = new MessageProp(false);
    fooContext.verifyMIC(micFromFoo, 0, micFromFoo.length, messageFromFoo, 0,
                         messageFromFoo.length, mprop);

    // foo ahora puede examinar mprop para determinar las propiedades del mensaje que ha
    // aplicado superSecureServer. En concreto, puede verificar que el mensaje no se ha
    // cifrado ya que getMIC nunca debe cifrar un mensaje.
```

Suprimir el contexto: Un igual suprime un contexto cuando el contexto ya no es necesario. En las operaciones de JGSS, cada igual decide de modo unilateral cuándo suprimir un contexto y no es necesario que informe de ello a su igual.

JGSS no define una señal de supresión de contexto. Para suprimir un contexto, el igual llama al método de eliminación (dispose) del objeto GSSContext para liberar los recursos empleados por el contexto. Es posible seguir accediendo a un objeto GSSContext eliminado, salvo que la aplicación establezca el objeto como nulo. Sin embargo, todo intento de utilizar un contexto eliminado (pero al que todavía se puede acceder) generará una excepción.

Utilizar JAAS con la aplicación JGSS: IBM JGSS incluye un recurso de inicio de sesión de JAAS opcional que permite a la aplicación utilizar JAAS para obtener credenciales. Una vez que el recurso de inicio de sesión de JAAS guarda las credenciales de principales y claves secretas en el objeto de sujeto de un contexto de inicio de sesión de JAAS, JGSS puede recuperar las credenciales de ese sujeto.

El comportamiento por omisión de JGSS consiste en recuperar las credenciales y claves secretas del sujeto. Puede inhabilitar esta función estableciendo la propiedad Java javax.security.auth.useSubjectCredsOnly en false.

Nota: Aunque el proveedor Java JGSS puro puede emplear la interfaz de inicio de sesión, el proveedor iSeries JGSS nativo no puede.

Para obtener más información sobre las funciones de JAAS, consulte Obtener credenciales de Kerberos y claves secretas.

Para utilizar la función de inicio de sesión de JAAS, la aplicación debe seguir el modelo de programación de JAAS del modo siguiente:

- Crear un contexto de inicio de sesión de JAAS
- Operar en los límites de una construcción JAAS Subject.doAs

El snippet de código siguiente muestra el concepto de operar en los límites de una construcción JAAS Subject.doAs:

```
static class JGSSOperations implements PrivilegedExceptionAction {
    public JGSSOperations() {}
    public Object run () throws GSSException {
        // El código de la aplicación JGSS va/se ejecuta aquí
    }
}

public static void main(String args[]) throws Exception {
    // Crear un contexto de inicio de sesión que utilizará el
    // manejador de retornos de llamada de Kerberos
    // com.ibm.security.auth.callback.Krb5CallbackHandler

    // Debe haber una configuración de JAAS para "JGSSClient"
    LoginContext loginContext =
        new LoginContext("JGSSClient", new Krb5CallabackHandler());
        loginContext.login();

    // Ejecutar toda la aplicación JGSS en modalidad privilegiada de JAAS
    Subject.doAsPrivileged(loginContext.getSubject(),
                           new JGSSOperations(), null);
}
```

Depuración

Cuando intente identificar problemas de JGSS, utilice la posibilidad de depuración de JGSS para generar mensajes categorizados, de gran utilidad. Puede activar una o varias categorías estableciendo los valores adecuados para la propiedad Java com.ibm.security.jgss.debug. Para activar varias categorías, utilice una coma a fin de separar los nombres de categoría.

Las categorías de depuración son las siguientes:

Categoría	Descripción
help	Listar categorías de depuración
all	Activar la depuración para todas las categorías
off	Desactivar la depuración por completo
app	Depuración de aplicaciones (valor por omisión)

Categoría	Descripción
ctx	Depuración de operaciones de contexto
cred	Operaciones de credenciales (con el nombre)
marsh	Ordenamiento de señales
mic	Operaciones de MIC
prov	Operaciones de proveedor
qop	Operaciones de QOP
unmarsh	Desordenamiento de señales
unwrap	Operaciones de desenvoltura
wrap	Operaciones de envoltura

Clase de depuración de JGSS: Para depurar la aplicación JGSS de modo programático, utilice la clase de depuración de la infraestructura de IBM JGSS. La aplicación puede emplear la clase de depuración para activar y desactivar las categorías de depuración y visualizar información de depuración para las categorías activas.

El constructor de depuración por omisión lee la propiedad Java com.ibm.security.jgss.debug para determinar qué categorías debe activar.

Ejemplo: depuración para la categoría de aplicaciones

El ejemplo siguiente muestra cómo solicitar información de depuración para la categoría de aplicaciones:

```
import com.ibm.security.jgss.debug;

Debug debug = new Debug(); // Obtiene categorías de la propiedad Java

// Muchas tareas necesarias para configurar someBuffer. Probar que la
// categoría está activa antes de efectuar la configuración para la depuración.

if (debug.on(Debug.OPTS_CAT_APPLICATION)) {
    // Llenar someBuffer con datos.
 debug.out(Debug.OPTS_CAT_APPLICATION, someBuffer);
    // someBuffer puede ser una matriz de bytes o una serie.
```

Ejemplos: IBM Java Generic Security Service (JGSS)

Los archivos de ejemplo de IBM Java Generic Security Service (JGSS) incluyen programas cliente y servidor, archivos de configuración, archivos de política e información de consulta de javadoc.

Puede ver versiones HTML de los ejemplos o bajar la información de javadoc y el código fuente de los programas de ejemplo. El hecho de bajar los ejemplos le permite ver la información de consulta de javadoc, examinar el código, editar los archivos de configuración y política y compilar y ejecutar los programas de ejemplo:

- Ver versiones HTML de los ejemplos
- Bajar y visualizar información de javadoc de ejemplos
- Bajar y ejecutar los programas de ejemplo

Descripción de los programas de ejemplo: Los ejemplos de JGSS incluyen cuatro programas:

- Servidor no JAAS
- Cliente no JAAS
- Servidor habilitado para JAAS
- Cliente habilitado para JAAS

Las versiones habilitadas para JAAS son plenamente interoperativas con sus versiones no JAAS correspondientes. Por consiguiente, puede ejecutar un cliente habilitado para JAAS con un servidor no JAAS, así como ejecutar un cliente no JAAS con un servidor habilitado para JAAS.

Nota: al ejecutar un ejemplo, puede especificar una o varias propiedades Java opcionales, tales como los nombres de los archivos de configuración y política, las opciones de depuración de JGSS y el gestor de seguridad. También puede activar y desactivar las funciones de JAAS.

Puede ejecutar los ejemplos en una configuración de un servidor o dos servidores. La configuración de un servidor consta de un cliente que se comunica con un servidor primario. La configuración de dos servidores consta de un servidor primario y otro secundario, donde el servidor primario actúa como iniciador, o cliente, para el servidor secundario.

Al utilizar la configuración de dos servidores, el cliente primero inicia un contexto e intercambia mensajes seguros con el servidor primario. A continuación, el cliente delega sus credenciales al servidor primario. Posteriormente, en nombre del cliente, el servidor primario utiliza estas credenciales para iniciar un contexto e intercambiar mensajes seguros con el servidor secundario. También puede emplear una configuración de dos servidores en la que el servidor primario actúa como cliente en nombre propio. En este caso, el servidor primario utiliza sus propias credenciales para iniciar un contexto e intercambiar mensajes con el servidor secundario.

Puede ejecutar cualquier número de clientes con el servidor primario de forma simultánea. Aunque puede ejecutar un cliente directamente con el servidor secundario, el servidor secundario no puede emplear las credenciales delegadas ni ejecutarse como iniciador con sus propias credenciales.

Ver los ejemplos de IBM JGSS: Los archivos de ejemplo de IBM Java Generic Security Service (JGSS) incluyen programas cliente y servidor, archivos de configuración, archivos de política e información de consulta de javadoc. Emplee los enlaces siguientes para ver las versiones HTML de los ejemplos de JGSS.

Para obtener más información, consulte los siguientes temas:
- "Descripción de los programas de ejemplo" en la página 230
- Bajar y ejecutar los programas de ejemplo

Ver los programas de ejemplo: Visualice las versiones HTML de los programas de ejemplo de JGSS mediante los enlaces siguientes:
- Programa cliente no JAAS de ejemplo
- Programa servidor no JAAS de ejemplo
- Programa cliente habilitado para JAAS de ejemplo
- Programa servidor habilitado para JAAS de ejemplo

Ver los archivos de configuración y política de ejemplo: Visualice las versiones HTML de los archivos de configuración y política de JGSS mediante los enlaces siguientes:
- Archivo de configuración de Kerberos
- Archivo de configuración de JAAS
- Archivo de política de JAAS
- Archivo de política Java

Ejemplos: bajar y visualizar información de javadoc para los ejemplos de IBM JGSS: Para bajar y visualizar la documentación de los programas de ejemplo de IBM JGSS, siga estos pasos:
1. Elija un directorio existente (o cree uno nuevo) donde desee almacenar la información de javadoc.
2. Baje la información de javadoc (jgsssampledoc.zip) al directorio.
3. Extraiga los archivos de jgsssampledoc.zip al directorio.
4. Utilice el navegador para acceder al archivo index.htm.

Declaración de limitación de responsabilidad sobre el código de ejemplo

IBM otorga al usuario una licencia de copyright no exclusiva para utilizar todos los ejemplos de código de programación, a partir de los que puede generar funciones similares adaptadas a sus necesidades específicas.

IBM proporciona la totalidad del código de ejemplo sólo con propósito ilustrativo. Estos ejemplos no se han probado exhaustivamente bajo todas las condiciones. Por tanto, IBM no puede garantizar la fiabilidad, capacidad de servicio o funcionamiento de estos programas.

Todos los programas que contiene esta documentación se suministran ″TAL CUAL″, sin garantías de ninguna clase. Se renuncia explícitamente a las garantías implícitas de no infringibilidad, comercialización y adecuación a un propósito determinado.

Ejemplos: bajar y ejecutar los programas de ejemplo: Antes de modificar o ejecutar los ejemplos, lea la “Descripción de los programas de ejemplo” en la página 230.

Para ejecutar los programas de ejemplo, lleve a cabo las siguientes tareas:

1. Bajar los archivos de ejemplo al servidor iSeries
2. Prepararse para ejecutar los archivos de ejemplo
3. Ejecutar los programas de ejemplo

Para obtener más información sobre cómo ejecutar un ejemplo, consulte Ejemplo: ejecutar el ejemplo no JAAS.

Declaración de limitación de responsabilidad sobre el código de ejemplo

IBM otorga al usuario una licencia de copyright no exclusiva para utilizar todos los ejemplos de código de programación, a partir de los que puede generar funciones similares adaptadas a sus necesidades específicas.

IBM proporciona la totalidad del código de ejemplo sólo con propósito ilustrativo. Estos ejemplos no se han probado exhaustivamente bajo todas las condiciones. Por tanto, IBM no puede garantizar la fiabilidad, capacidad de servicio o funcionamiento de estos programas.

Todos los programas que contiene esta documentación se suministran ″TAL CUAL″, sin garantías de ninguna clase. Se renuncia explícitamente a las garantías implícitas de no infringibilidad, comercialización y adecuación a un propósito determinado.

Ejemplos: bajar los ejemplos de IBM JGSS: Antes de modificar o ejecutar los ejemplos, lea la “Descripción de los programas de ejemplo” en la página 230.

Para bajar los archivos de ejemplo y almacenarlos en el servidor iSeries, siga estos pasos:

1. En el servidor iSeries, elija un directorio existente (o cree uno nuevo) donde desee almacenar los programas de ejemplo, archivos de configuración y archivos de política.
2. Baje los programas de ejemplo (ibmjgsssample.zip).
3. Extraiga los archivos de ibmjgsssample.zip al directorio del servidor.

Al extraer el contenido de ibmjgsssample.jar se llevan a cabo las acciones siguientes:

- Se coloca ibmgjsssample.jar, que contiene los archivos .class de ejemplo, en el directorio seleccionado.
- Se crea un subdirectorio (denominado config) que contiene los archivos de configuración y política.
- Se crea un subdirectorio (denominado src) que contiene los archivos fuente .java de ejemplo.

Información relacionada: Si lo desea, puede leer información acerca de las tareas relacionadas o examinar un ejemplo:

- Prepararse para ejecutar los archivos de ejemplo
- Ejecutar los programas de ejemplo
- Ejemplo: ejecutar el ejemplo no JAAS

Ejemplos: prepararse para ejecutar los programas de ejemplo: Antes de modificar o ejecutar los ejemplos, lea la "Descripción de los programas de ejemplo" en la página 230.

Tras bajar el código fuente, debe llevar a cabo las tareas siguientes antes de poder ejecutar los programas de ejemplo:

- Edite los archivos de configuración y política de modo que se adecuen al entorno. Para obtener más información, consulte los comentarios de cada uno de los archivos de configuración y política.
- Compruebe que el archivo java.security contiene los valores correctos para el servidor iSeries. Para obtener más información, consulte "Archivo de propiedades de seguridad maestro de Java" en la página 221.
- Coloque el archivo de configuración de Kerberos (krb5.conf) modificado en el directorio del servidor iSeries adecuado para la versión de J2SDK que utiliza:
  - Para la versión 1.3 de J2SDK: /QIBM/ProdData/Java400/jdk13/lib/security
  - Para la versión 1.4 de J2SDK: /QIBM/ProdData/Java400/jdk14/lib/security

Información relacionada: Si lo desea, puede leer información acerca de las tareas relacionadas o examinar un ejemplo:

- Bajar los archivos de ejemplo al servidor iSeries
- Ejecutar los programas de ejemplo
- Ejemplo: ejecutar el ejemplo no JAAS

Ejemplos: ejecutar los programas de ejemplo: Antes de modificar o ejecutar los ejemplos, lea la "Descripción de los programas de ejemplo" en la página 230.

Tras bajar y modificar el código fuente, puede ejecutar uno de los ejemplos.

Para ejecutar un ejemplo, primero debe iniciar el programa servidor. El programa servidor debe estar en ejecución y preparado para recibir conexiones antes de iniciar el programa cliente. El servidor estará preparado para recibir conexiones cuando vea el mensaje `listening on port <puerto_servidor>`. Asegúrese de recordar o anotar el <puerto_servidor>, que es el número de puerto que deberá especificar cuando inicie el cliente.

Utilice el mandato siguiente para iniciar un programa de ejemplo:

```
java [-Dpropiedad1=valor1 ... -DpropiedadN=valorN] com.ibm.security.jgss.test.<programa> [opciones]
```

donde

- [-DpropiedadN=valorN] es una o varias propiedades Java opcionales, tales como los nombres de los archivos de configuración y política, las opciones de depuración de JGSS y el gestor de seguridad. Para obtener más información, consulte el ejemplo siguiente y Ejecutar aplicaciones JGSS.
- <programa> es un parámetro obligatorio que especifica el programa de ejemplo que desea ejecutar (`Client`, `Server`, `JAASClient` o `JAASServer`).
- [opciones] es un parámetro opcional para el programa de ejemplo que desea ejecutar. Para ver una lista de las opciones permitidas, utilice el mandato siguiente:

  ```
  java com.ibm.security.jgss.test.<programa> -?
  ```

Nota: desactive las funciones de JAAS en un ejemplo habilitado para JGSS estableciendo la propiedad Java javax.security.auth.useSubjectCredsOnly en false. Naturalmente, el valor por omisión de los ejemplos habilitados para JAAS establece la activación de JAAS, con lo que el valor de la propiedad es true. Los programas cliente y servidor no JAAS establecen la propiedad en false, salvo que se haya establecido explícitamente el valor de la propiedad.

Información relacionada: Si lo desea, puede leer información acerca de las tareas relacionadas o examinar un ejemplo:

- Bajar los archivos de ejemplo al servidor iSeries
- Prepararse para ejecutar los archivos de ejemplo
- Ejemplo: ejecutar el ejemplo no JAAS

Información de consulta de javadoc de IBM JGSS

La información de consulta de javadoc de IBM JGSS incluye las clases y los métodos del paquete de API org.ietf.jgss y las versiones Java de algunas herramientas de gestión de credenciales de Kerberos.

Aunque JGSS contiene varios paquetes de acceso público (por ejemplo, com.ibm.security.jgss y com.ibm.security.jgss.spi), se recomienda utilizar únicamente las API del paquete org.ietf.jgss estándar. El uso exclusivo de este paquete garantiza el cumplimiento de las especificaciones GSS-API por parte de la aplicación, así como una interoperatividad y una portabilidad óptimas.

- org.ietf.jgss
- kinit
- ktab
- klist

Ajustar el rendimiento de un programa Java con IBM Developer Kit para Java

Debe tomar en consideración diversos aspectos relativos al rendimiento de las aplicaciones Java al crear una aplicación Java para el servidor iSeries. He aquí algunos enlaces con información detallada y sugerencias para obtener un rendimiento mejor:

- Mejore el rendimiento del código Java utilizando el mandato Crear programa Java (CRTJVAPGM), el compilador Just-In-Time, o utilizando la antememoria para los cargadores de clases de usuario.
- Cambie los niveles de optimización para alcanzar el mejor rendimiento de compilación estática.
- Establezca cuidadosamente los valores para un rendimiento de recogida de basura óptimo.
- Utilice sólo los métodos nativos para iniciar funciones del sistema que sean de relativamente larga ejecución y que no estén disponibles directamente en Java.
- Utilice la opción -o de `javac` en tiempo de compilación para realizar la incorporación de métodos y mejorar significativamente el rendimiento de las llamadas a métodos.
- Utilice excepciones Java en los casos en que no se produzca el flujo normal a través de la aplicación.

Para localizar los problemas de rendimiento en los programas Java, utilice las herramientas indicadas a continuación junto con el explorador de rendimiento (PEX):

- Puede recoger eventos de rastreo Java mediante la máquina virtual Java de iSeries.
- Para determinar el tiempo que se invierte en cada uno de los métodos Java, utilice rastreos de llamadas Java.
- El perfilado Java localiza el tiempo de CPU relativo que se invierte en cada uno de los métodos Java y todas las funciones del sistema que está utilizando el programa Java.
- Utilice Java Performance Data Collector para proporcionar información de perfil relativa a los programas que se ejecutan en el servidor iSeries.

Cualquier sesión de trabajo puede iniciar y finalizar PEX. Normalmente, los datos se recogen a escala de todo el sistema y están relacionados con todos los trabajos del sistema, incluidos los programas Java. A veces, puede ser necesario iniciar y detener la recogida de rendimiento desde el interior de una aplicación Java. Con ello se reduce el tiempo de recogida y puede reducirse el gran volumen de datos producidos generalmente por un rastreo de retorno o de llamada. PEX no se puede ejecutar desde dentro de una hebra Java. Para iniciar y detener una recogida, es necesario escribir un método nativo que se comunique con un trabajo independiente a través de una cola o memoria compartida. Luego, el segundo trabajo inicia y detiene la recogida en el momento oportuno.

Además de los datos de rendimiento a nivel de aplicación, puede utilizar las herramientas existentes de rendimiento a nivel del sistema iSeries. Estas herramientas proporcionan un informe de estadísticas por cada hebra Java.

Para obtener ejemplos de informes PEX, consulte la publicación Performance Tools for iSeries, SC41-5340

.

Consideraciones sobre el rendimiento de Java

La total comprensión de las siguientes consideraciones puede ayudarle a mejorar el rendimiento de sus aplicaciones Java:

- "Crear programas Java optimizados"
- "Utilizar el compilador Just-In-Time" en la página 236
-

"Utilizar antememorias para cargadores de clases de usuario" en la página 236

Crear programas Java optimizados

Para mejorar considerablemente el rendimiento de arranque del código Java, utilice el mandato de lenguaje de control Crear programa Java (CRTJVAPGM) antes de ejecutar archivos de clase Java, archivos JAR o archivos ZIP. El mandato CRTJVAPGM utiliza los bytecodes para crear un objeto programa Java que contiene instrucciones nativas optimizadas para el servidor iSeries y asocia el objeto programa Java al archivo de clase, JAR o ZIP.

Las ejecuciones subsiguientes serán mucho más rápidas porque el programa Java se ha guardado y permanece asociado al archivo de clase o al archivo JAR. Durante la fase de desarrollo de la aplicación, el rendimiento obtenido al ejecutar los bytecodes de manera interpretada puede ser aceptable, pero en un entorno de producción, es más conveniente utilizar el mandato CRTJVAPGM antes de ejecutar el código Java.

Si no utiliza CRTJVAPGM antes de ejecutar un archivo de clase Java, un archivo JAR o un archivo ZIP, OS/400 utiliza en su lugar el compilador Just-In-Time (con el intérprete de modalidad mixta).

Seleccionar el nivel de optimización

Al crear el objeto programa Java, utilice las siguientes directrices como ayuda para seleccionar el mejor nivel de optimización para la modalidad de ejecución que desea utilizar:

- Cuando desee utilizar el proceso directo, cree el objeto programa Java optimizado en el nivel de optimización 30 o 40.

- Cuando desee ejecutar solamente con el compilador JIT, cree el programa Java optimizado utilizando el parámetro de optimización `*Interpret`. Un programa Java creado mediante el parámetro `*Interpret` es más pequeño que uno creado utilizando el nivel de optimización 40.
- Cuando desee utilizar la modalidad de ejecución por omisión, que es una mezcla de proceso directo y el compilador JIT, utilice los siguientes valores para crear los objetos programa Java:
  - Para las clases que desee ejecutar con el proceso directo, utilice el nivel de optimización 30 o 40
  - Para las clases que desee ejecutar con el compilador JIT, utilice el parámetro de optimización `*Interpret`

Para obtener más información, consulte las siguientes páginas:

Mandato de lenguaje de control Crear programa Java (CRTJVAPGM)

Seleccionar la modalidad que debe utilizarse al ejecutar un programa Java

Utilizar el compilador Just-In-Time

Utilizar el compilador Just-In-Time (JIT) con el Intérprete de modalidad mixta (MMI) da como resultado un rendimiento de arranque casi similar al del código compilado. MMI interpreta el código Java hasta alcanzar el umbral especificado por la propiedad del sistema Java os400.jit.mmi.threshold. Una vez alcanzado el umbral, OS/400 emplea el tiempo y los recursos necesarios para utilizar el compilador JIT en compilar un método sobre los métodos utilizados con mayor frecuencia. Utilizar el compilador JIT da como resultado un código altamente optimizado que mejora el rendimiento de la ejecución en comparación con el código precompilado. Cuando requiera un rendimiento de arranque mejorado con el compilador JIT, puede utilizar CRTJVAPGM para crear un objeto programa Java optimizado.

Si el programa se ejecuta con lentitud, entre el mandato de lenguaje de control Visualizar programa Java (DSPJVAPGM) para ver los atributos de un objeto programa Java. Asegúrese de que el objeto programa Java utiliza la mejor modalidad de ejecución para sus fines. Si desea cambiar la modalidad de ejecución, puede interesarle suprimir el objeto programa Java y crear uno nuevo utilizando parámetros de optimización distintos.

Para obtener más información, consulte lo siguiente:

"Crear programas Java optimizados" en la página 235

Mandato de lenguaje de control Visualizar programa Java (DSPJVAPGM)

Utilizar antememorias para cargadores de clases de usuario

Utilizar la antememoria de la máquina virtual Java (JVM) OS/400 para los cargadores de clases de usuario mejora el rendimiento de arranque para las clases que cargue desde un cargador de clases de usuario. La antememoria almacena los objetos programa Java optimizados, lo que permite a la JVM volver a utilizarlos. Volver a utilizar programas Java almacenados aumenta el rendimiento al evitar que vuelvan a crearse los objetos programa Java en antememoria y al verificar el bytecode.

Utilice las siguientes propiedades para controlar las antememorias para los cargadores de clases de usuario:

os400.define.class.cache.file

El valor de esta propiedad especifica el nombre (con la vía de acceso completa) de un archivo Java ARchive (JAR) válido. Como mínimo, el archivo JAR especificado debe contener un directorio JAR válido (según el creado por el mandato QSH `jar`) y el miembro individual necesario para el mandato `jar` para poder funcionar. No incluya el archivo JAR especificado en

ninguna CLASSPATH Java. El valor por omisión de esta propiedad es /QIBM/ProdData/Java400/QDefineClassCache.jar. Para inhabilitar la puesta en antememoria, especifique esta propiedad sin ningún valor.

os400.define.class.cache.hours
El valor de esta propiedad especifica cuánto tiempo (en horas) desea que un objeto programa Java permanezca en la antememoria. Cuando la JVM no utiliza un objeto programa Java en antememoria durante el período de tiempo especificado, OS/400 elimina el objeto programa Java de la antememoria. El valor por omisión de esta propiedad es de 768 horas (33 días). El valor máximo es de 9999 (unas 59 semanas). Cuando especifique un valor de 0 o un valor que OS/400 no reconozca como un número decimal válido, OS/400 utilizará el valor por omisión.

os400.define.class.cache.maxpgms
El valor de esta propiedad especifica el número máximo de objetos programa Java que puede mantener la antememoria. Cuando se sobrepasa este límite de la antememoria, OS/400 elimina de ella el objeto programa Java más antiguo. OS/400 determina qué programa en antememoria es el más antiguo comparando las horas en que la JVM efectuó la última referencia de los objetos programa Java. El valor por omisión es 5000 y el valor máximo es 40000. Cuando especifique un valor de 0 o un valor que OS/400 no reconozca como un número decimal válido, OS/400 utilizará el valor por omisión.

Utilice DSPJVAPGM en el archivo JAR, que se especifica en la propiedades 400.define.class.cache.file, para determinar el número de objetos programa Java en antememoria.

- El campo **Programas Java** de la pantalla DSPJVAPGM indica el número de objetos programa Java en antememoria.
- El campo **Tamaño de programa Java** indica la cantidad de almacenamiento utilizada por los objetos programa Java en antememoria.
- Otros campos de la pantalla DSPJVAPGM no tienen efecto al utilizar el mandato en un archivo JAR que esté utilizando para la puesta en antememoria.

Rendimiento de la antememoria

Al ejecutar algunas aplicaciones Java se puede poner en antememoria un gran número de objetos programa. Utilice DSPJVAPGM para determinar si el número de programas Java en antememoria se acerca al valor máximo antes de que la aplicación termine de ejecutarse. El rendimiento de la aplicación puede deteriorarse cuando se llena la antememoria, ya que OS/400 podría eliminar de la antememoria algunos programas que la aplicación puede necesitar.

Puede evitar la degradación del rendimiento que se produce cuando se llena la antememoria. Por ejemplo, puede configurar aplicaciones para utilizar antememorias separadas para las aplicaciones que se ejecutan con frecuencia pero cargar programas distintos en la antememoria. Utilizar antememorias separadas puede evitar que la antememoria se llene y evitar así que OS/400 elimine programas Java de la antememoria. Otra solución es aumentar el número que especifique para la propiedad os400.define.class.cache.maxpgms.

Puede utilizar el mandato de lenguaje de control Cambiar programa Java (CHGJVAPGM) en el archivo JAR para cambiar la optimización de las clases en la antememoria. CHGJVAPGM afecta solamente a los programas que están actualmente en la antememoria. Tras realizar cambios en los niveles de optimización, la propiedad os400.defineClass.optLevel especifica cómo optimizar las clases que se añaden a la antememoria.

Por ejemplo, para utilizar el JAR de antememoria enviado con un máximo de 10000 objetos programa Java, donde cada programa Java tiene una vida máxima de 1 año, establezca los siguientes valores para las propiedades de antememoria:

```
os400.define.class.cache.file    /QIBM/ProdData/Java400/QDefineClassCache.jar
os400.define.class.cache.hours   8760
os400.define.class.cache.maxpgms 10000
```

Para obtener más información, consulte lo siguiente:

Propiedades del sistema Java (page 20)

Mandato de lenguaje de control Cambiar programa Java (CHGJVAPGM)

Seleccionar la modalidad que debe utilizarse al ejecutar un programa Java

Al ejecutar un programa Java, puede seleccionar la modalidad que desea utilizar. Todas las modalidades verifican el código y crean un objeto programa Java en el que conservar el formato de programa anterior a la verificación. Puede utilizar cualquiera de las siguientes modalidades:

- Interpretado
- Proceso directo
- Compilación Just-In-Time (JIT)
- Compilación Just-In-Time (JIT) y proceso directo

Modalidad de selección	Detalles
Interpretado	Cada bytecode se interpreta en el momento de la ejecución. Para obtener información acerca de la ejecución del programa Java en modalidad interpretada, consulte el apartado correspondiente al mandato Ejecutar Java (RUNJVA).
Proceso directo	Se generan instrucciones de máquina para un método durante la primera llamada a ese método, y se guardan para utilizarse la próxima vez que se ejecute el programa. También se comparte una copia para todo el sistema. Para obtener información acerca de la ejecución del programa Java mediante el proceso directo, consulte la sección Mandato Ejecutar Java (RUNJVA).

Modalidad de selección	Detalles

Compilación Just-In-Time (JIT)	OS/400 interpreta los métodos Java hasta alcanzar el umbral especificado por la propiedad del sistema Java os400.jit.mmi.threshold. Una vez alcanzado el umbral, OS/400 utiliza el compilador JIT para compilar métodos como instrucciones de máquina nativa.
	
	Para utilizar el compilador Just-In-Time, tiene que establecer el valor del compilador en `jitc`. Puede establecer el valor añadiendo una variable de entorno o estableciendo la propiedad java.compiler del sistema. Seleccione un método de la lista siguiente para establecer el valor del compilador:
	• Desde un indicador de línea de mandatos del servidor iSeries, añada la variable de entorno mediante el mandato Añadir variable de entorno (ADDENVVAR). A continuación, ejecute el programa Java mediante el mandato Ejecutar Java (RUNJVA) o el mandato JAVA. Por ejemplo, utilice:
	`ADDENVVAR ENVVAR (JAVA_COMPILER) VALUE(jitc)`
	`JAVA CLASS(Test)`
	• Establezca la propiedad java.compiler del sistema en la línea de mandatos de iSeries. Por ejemplo, entre `JAVA CLASS(Test) PROP((java.compiler jitc))`
	• Establezca la propiedad java.compiler del sistema en la línea de mandatos del intérprete de Qshell. Por ejemplo, entre `java -Djava.compiler=jitc Test`
	
	Una vez establecido este valor, el compilador JIT optimiza todo el código Java antes de ejecutarlo.

Modalidad de selección	Detalles
Compilación Just-In-Time (JIT) y proceso directo	La forma más común de utilizar el compilador Just-In-Time (JIT) es con la opción `jit_de`. Al ejecutar con esta opción, los programas que ya se han optimizado con el proceso directo se ejecutan en modalidad de proceso directo. Los programas que no se han optimizado para la optimización directa se ejecutan en modalidad JIT. Para utilizar JIT y el proceso directo conjuntamente, es necesario establecer el valor de compilador en `jitc_de`. Puede establecer el valor añadiendo una variable de entorno o estableciendo la propiedad java.compiler del sistema. Seleccione un método de la lista siguiente para establecer el valor del compilador: • Añada la variable de entorno entrando el mandato Añadir variable de entorno (ADDENVVAR) en la línea de mandatos de iSeries. A continuación, ejecute el programa Java mediante el mandato Ejecutar Java (RUNJVA) o el mandato JAVA. Por ejemplo, entre: `ADDENVVAR ENVVAR (JAVA_COMPILER) VALUE(jitc_de) JAVA CLASS(Test)` • Establezca la propiedad java.compiler del sistema en la línea de mandatos de iSeries. Por ejemplo, entre `JAVA CLASS(Test) PROP((java.compiler jitc_de))` • Establezca la propiedad java.compiler del sistema en la línea de mandatos del intérprete de Qshell. Por ejemplo, entre `java -Djava.compiler=jitc_de Test` Una vez establecido este valor, se utiliza el programa Java correspondiente al archivo de clase creado como de proceso directo. Si el programa Java no se ha creado como de proceso directo, JIT optimiza el archivo de clase antes de que se ejecute. Para obtener más información, consulte la sección Comparación entre el compilador Just-In-Time y el proceso directo.

Existen tres maneras de ejecutar un programa Java (CL, QSH y JNI). Cada una de ellas tiene una forma exclusiva de especificar la modalidad. En esta tabla podrá ver lo que se hace:

Modalidad	Mandato CL	Mandato de QShell	La API de invocación de JNI
Intérprete	INTERPRET(*YES)	-Djava.compiler=NONE -interpret	os400.run.mode=interpret
DE	INTERPRET(*NO)	-Djava.compiler=NONE	• os400.run.mode= program_created=pc • os400.create.type= direct
JIT	INTERPRET(*JIT)	-Djava.compiler=jitc	os400.run.mode=jitc
JIT_DE (valor por omisión)	INTERPRET(*OPTIMIZE) OPTIMIZE(*JIT)	-Djava.compiler=jitc_de	os400.run.mode=jitc_de

Intérprete de Java

El intérprete de Java es el componente de la máquina virtual Java que interpreta los archivos de clase Java para una plataforma de hardware determinada. El intérprete de Java decodifica cada bytecode y ejecuta una serie de instrucciones de máquina para ese bytecode.

Compilación estática

El transformador Java es un componente de IBM Operating System/400 (OS/400) que preprocesa archivos de clase para prepararlos para la ejecución con la máquina virtual Java de iSeries. El transformador Java crea un objeto programa optimizado que es persistente y está asociado al archivo de clase. En el caso por omisión, el objeto programa contiene una versión compilada con instrucciones de lenguaje máquina RISC de 64 bits de la clase. El intérprete Java no interpreta el objeto programa optimizado en el momento de la ejecución. En lugar de ello, el objeto programa se ejecuta directamente cuando se carga el archivo de clase.

Por omisión, los programas Java se optimizan mediante JIT. Para utilizar el transformador Java, ejecute el mandato CRTJVAPGM o especifique la utilización del transformador en el mandato RUNJVA o JAVA.

Para iniciar explícitamente el transformador Java, puede utilizar el mandato Crear programa Java (CRTJVAPGM). El mandato CRTJVAPGM optimiza el archivo de clase o el archivo JAR mientras se ejecuta el mandato, por lo que no es necesario realizar ninguna acción mientras se está ejecutando el programa. Esto aumenta la velocidad del programa la primera vez que se ejecuta. La utilización del mandato CRTJVAPGM, en lugar de basarse en la optimización por omisión, garantiza la mejor optimización posible y también mejora la utilización de espacio para los programas Java asociados con el archivo de clase o JAR.

La utilización del mandato CRTJVAPGM en un archivo de clase, JAR o ZIP provoca la optimización de todas las clases del archivo, y el objeto programa Java resultante es persistente. El resultado es que el rendimiento de ejecución mejora. También puede cambiar el nivel de optimización o seleccionar un nivel de optimización diferente del nivel por omisión 10 mediante el mandato CRTJVAPGM o mediante el mandato Cambiar programa Java (CHGJVAPGM). En el nivel de optimización 40, se realiza un enlace entre las clases que están dentro de un archivo JAR y, en algunos casos, las clases se incorporan. El enlace entre clases aumenta la velocidad de llamada. La incorporación elimina por completo la actividad general que supone la llamada a un método. En algunos casos, se pueden incorporar métodos entre clases que están dentro del archivo JAR o del archivo ZIP. Si se especifica `OPTIMIZE(*INTERPRET)` en el mandato CRTJVAPGM, las clases que estén especificadas en el mandato se verifican y preparan para ejecutarse en modalidad interpretada.

En el mandato Ejecutar Java (RUNJVA) también se puede especificar OPTIMIZE(*INTERPRET).Este parámetro indica que las clases que se ejecuten en la máquina virtual Java son interpretadas, con independencia del nivel de optimización del objeto programa asociado. Esto resulta útil a la hora de depurar una clase que haya sido transformada con el nivel de optimización 40. Para forzar la interpretación, utilice INTERPRET(*YES).

Consulte el apartado ″Utilizar antememoria para cargadores de clases de usuario″ en Consideraciones sobre el rendimiento de Java para obtener información acerca de la reutilización de los programas Java creados por cargadores de clases.

Consideraciones sobre el rendimiento de la compilación estática Java: Se puede determinar la velocidad de transformación mediante el nivel de optimización establecido. El nivel de optimización 10 es el que tiene la mayor velocidad de transformación, pero el programa resultante suele ser más lento que uno que tenga establecido un nivel de optimización superior. El nivel de optimización 40 tarda más en realizar la transformación, pero suele ejecutarse con mayor rapidez.

Un pequeño número de programas Java no puede optimizarse al nivel 40. Así, algunos de los programas que no se ejecutan al nivel 40, pueden ejecutarse en cambio al nivel 30. Puede ejecutar programas que no funcionen en el nivel de optimización 40 mediante series de parámetro LICOPT de optimización del código interno bajo licencia. No obstante, el rendimiento al nivel 30 puede bastar para el programa.

Si tiene problemas al ejecutar código Java que parecía funcionar en otra máquina virtual Java, pruebe con el nivel de optimización 30 en lugar de con el nivel 40. Si funciona y el nivel de rendimiento es aceptable, no es necesario hacer nada más. Si necesita un rendimiento mejor, consulte el tema dedicado a las series

del parámetro LICOPT, donde hallará más información sobre la manera de habilitar e inhabilitar las diversas formas de optimización. Por ejemplo, podría intentar crear primero el programa utilizando OPTIMIZE(40) LICOPT(NoPreresolveExtRef). Si la aplicación contiene llamadas ″muertas″ a clases que no están disponibles, este valor de LICOPT permite que el programa se ejecute sin problemas.

Para determinar a qué nivel de optimización se han creado los programas Java, puede utilizar el mandato Visualizar programa Java (DSPJVAPGM). Para cambiar el nivel de optimización del programa Java, utilice el mandato Crear programa Java (CRTJVAPGM).

Compilador Just-In-Time

Un compilador Just-In-Time (JIT) es un compilador específico de plataforma que genera instrucciones de máquina para cada método a medida que son necesarias.

Para obtener más información sobre cómo utilizar el compilador JIT y sobre la diferencia que hay entre el compilador JIT y el proceso directo, consulte las páginas siguientes:

Consideraciones sobre el rendimiento de ejecución Java.

Comparación entre el compilador JIT y el proceso directo.

Nota: El valor por omisión para OS/400 es interpretar (no compilar) métodos Java utilizando el Intérprete de modalidad mixta (MMI). MMI perfila cada método Java a medida que lo interpreta. Tras alcanzar el umbral especificado por la propiedad os400.jit.mmi.threshold, MMI especifica entonces que OS/400 utilice el compilador JIT para compilar el método.

Para obtener más información, consulte las entradas para la propiedad java.compiler y la propiedad os400.jit.mmi.threshold en la lista correspondiente de propiedades del sistema Java:

Propiedades del sistema Java para Java 2 SDK (J2SDK), Standard Edition

Comparación entre el compilador Just-In-Time y el proceso directo: Si está intentando decidir si es mejor utilizar el compilador Just-In-Time o la modalidad de proceso directo para ejecutar el programa Java, en esta tabla hallará información adicional que le ayudará a elegir la mejor opción para su situación.

Compilador Just-In-Time	Proceso directo
Proporciona una compilación automática de cualquier método cuando es necesaria. El compilador JIT puede compilar un método mucho más rápidamente que el proceso directo.	Le permite compilar una clase entera o un archivo JAR utilizando el mandato de lenguaje de control (CL) Crear programa Java (CRTJVAPGM). Si no compila los archivos, el proceso directo compila los archivos automáticamente durante la ejecución.
Le permite evitar utilizar el mandato CL CRTJVAPGM durante el desarrollo de programas. También puede utilizar el compilador JIT con aplicaciones altamente dinámicas que generen o descubran código durante la ejecución.	La mayoría de las aplicaciones de servidor preparadas para el despliegue utilizan el proceso directo con el nivel de optimización 40, ya que es posible que las estén utilizando varios usuarios en un momento dado. Múltiples trabajos de usuario comparten el mismo espacio de código en la memoria, lo que reduce el uso de memoria.
Realiza rápidamente optimizaciones complejas y optimizaciones específicas de Java durante la ejecución.	Habilita las optimizaciones complejas, ya que el proceso directo no realiza optimización durante la ejecución. Sin embargo, el proceso directo no siempre puede realizar optimizaciones específicas de Java (como los métodos de incorporación) porque los objetos programa Java deben ser independientes.

Compilador Just-In-Time	Proceso directo
Ofrece un mejor rendimiento del código al compararlo con el proceso directo. En la mayoría de casos, el rendimiento del código generado por JIT es mejor que el nivel de optimización 40 del proceso directo.	Ofrece la única manera de que el programa Java puede adoptar autorización de propietario.

Recogida de basura de Java

La recogida de basura es el proceso por el cual se libera el almacenamiento que utilizan los objetos a los que ya no hace referencia un programa. Gracias a la recogida de basura, ya no es necesario que los programadores escriban código, susceptible de sufrir errores, para ″liberar″ o ″suprimir″ de manera explícita los objetos. En muchas ocasiones, el resultado que da este código son errores de programa que provocan ″fugas de memoria″. El recogedor de basura detecta automáticamente el objeto o grupo de objetos a los que ya no puede llegar el programa de usuario. Lo detecta porque ya no hay referencias al objeto en ninguna de las estructuras del programa. Una vez recogido un objeto, se puede asignar el espacio para otros usos.

El “Entorno de ejecución Java” en la página 180 incluye un recogedor de basura que libera la memoria que ya no se utiliza. El recogedor de basura se ejecuta automáticamente según convenga.

El recogedor de basura puede iniciarse también de forma explícita bajo el control del programa Java; para ello, debe utilizarse el método java.lang.Runtime.gc().

En Recogida de basura avanzada de IBM Developer Kit para Java hallará información específica de IBM Developer Kit para Java.

Recogida de basura avanzada de IBM Developer Kit para Java

IBM Developer Kit para Java implementa un algoritmo avanzado de recogida de basura. Este algoritmo permite descubrir y recoger objetos no alcanzables sin que se produzcan pausas significativas en el funcionamiento del programa Java. Un recogedor concurrente descubre de manera cooperativa las referencias hechas a objetos en las hebras en ejecución, en lugar de en una sola hebra.

Muchos recogedores de basura son de ″detención total″. Esto significa que, en el punto en que se produce un ciclo de recogida, se detienen todas las hebras, excepto la que efectúa la recogida de basura, mientras el recogedor de basura realiza su trabajo. Cuando esto sucede, los programas Java sufren una pausa y la capacidad multiprocesador que pueda tener la plataforma se malgasta por lo que a Java se refiere mientras el recogedor realiza su trabajo. El algoritmo de iSeries no detiene simultáneamente todas las hebras del programa. Por el contrario, deja que las hebras sigan funcionando mientras el recogedor de basura efectúa su tarea. Esto impide que se produzcan pausas y permite utilizar todos los procesadores mientras tiene lugar la recogida de basura.

La recogida de basura se efectúa de forma automática tomando como base los parámetros especificados al iniciar la máquina virtual Java. También puede iniciarse explícitamente bajo el control del programa Java utilizando el método java.lang.Runtime.gc().

Para obtener una definición básica, consulte el apartado Recogida de basura de Java.

Consideraciones sobre el rendimiento de la recogida de basura de Java

La recogida de basura en la máquina virtual Java de iSeries funciona en modalidad asíncrona continua. El parámetro tamaño inicial de recogida de basura (GCHINL) del mandato Ejecutar Java (RUNJVA) puede afectar al rendimiento de las aplicaciones. Este parámetro especifica el espacio de objetos nuevos que está permitido entre recogidas de basura. Si el valor es bajo, puede provocar una actividad general excesiva

de recogida de basura. Si es alto, la recogida de basura puede sufrir limitaciones y provocar errores por falta de memoria. Sin embargo, para la mayoría de las aplicaciones, los valores por omisión deberían ser correctos.

La recogida de basura determina si un objeto ya no es necesario evaluando si hay o no referencias válidas a dicho objeto.

Consideraciones sobre el rendimiento de la llamada a métodos nativos Java

La llamada a métodos nativos en un servidor iSeries puede no tener un rendimiento tan bueno como en otras plataformas. En el servidor iSeries, se ha optimizado Java moviendo la máquina virtual Java más abajo de la interfaz de máquina (MI). La llamada a métodos nativos requiere una llamada por encima del código MI y puede requerir llamadas costosas de la interfaz Java nativa (JNI) a la máquina virtual Java. Los métodos nativos no deben llevar a cabo rutinas pequeñas, que pueden escribirse fácilmente en Java. Utilícelos para iniciar funciones del sistema que sean de relativamente larga ejecución y que no estén disponibles directamente en Java.

Consideraciones sobre el rendimiento de la incorporación de métodos Java

La incorporación de métodos puede mejorar de manera significativa el rendimiento de las llamadas a métodos. Cualquier método que sea final es un candidato en potencia a la incorporación. La característica de incorporación está disponible en el servidor iSeries a través de la opción -o de `javac` en tiempo de compilación. El tamaño de los archivos de clase y el programa Java transformado se incrementa si se utiliza la opción -o de `javac`. A la hora de utilizar esta opción, debe tomar en consideración tanto el espacio como las características de rendimiento de la aplicación.

Nota: Por lo general, es mejor no utilizar la opción -o de javac, sino dejar la incorporación para fases posteriores.

El transformador Java habilita la incorporación para los niveles de optimización 30 y 40. El nivel de optimización 30 habilita en parte la incorporación de métodos finales dentro de una sola clase. El nivel de optimización 40 habilita la incorporación de métodos finales dentro de un archivo ZIP o JAR. La incorporación de métodos se puede controlar por medio de las series del parámetro LICOPT AllowInlining y NoAllowInlining.El intérprete de iSeries no realiza la incorporación de métodos.

El compilador Just-In-Time (JIT) también realiza la incorporación de la mayoría de métodos finales. Esto se realiza automáticamente siempre que el compilador JIT esté activo y determina que la incorporación será beneficiosa.

Consideraciones sobre el rendimiento de las excepciones Java

La arquitectura de excepciones de iSeries permite disponer de una capacidad de interrupción y reintento versátil. Permite la interacción de lenguajes mixtos. Lanzar excepciones Java en un servidor iSeries puede resultar más costoso que en otras plataformas. Esto no debe afectar al rendimiento global de la aplicación a menos que se utilicen rutinariamente excepciones Java en la vía habitual de la aplicación.

Herramientas de rendimiento de rastreo de llamadas Java

Los rastreos de llamadas a métodos Java facilitan información significativa de rendimiento acerca del tiempo que se invierte en cada uno de los métodos Java. En otras máquinas virtuales Java, se puede utilizar la opción -prof (perfilado) del mandato `java`. Para habilitar el rastreo de las llamadas a métodos en un servidor iSeries, debe especificar el mandato Habilitar recogida de rendimiento (ENBPFRCOL) en la línea de mandatos de Crear programa Java (CRTJVAPGM). Una vez creado el programa Java con esta palabra clave, puede iniciar la recogida de los rastreos de llamadas a métodos mediante una definición del explorador de rendimiento (PEX) que incluya el tipo de rastreo de llamada/retorno.

La salida de rastreo de llamada/retorno generada con el mandato Imprimir informe del explorador de rendimiento (PRTPEXRPT) muestra el tiempo de CPU para cada una de las llamadas de todos los métodos Java de los que se ha realizado un rastreo. En algunos casos, puede que no resulte posible habilitar todos los archivos de clase para el rastreo de llamada/retorno. O tal vez esté llamando a métodos nativos y a funciones del sistema que no estén habilitados para el rastreo. En esta situación, el tiempo de CPU invertido en dichos métodos o funciones del sistema se acumula. Después, se envía notificación al último método Java que se haya llamado y que esté habilitado.

Herramientas de rendimiento de perfilado Java

El perfilado de unidad central de proceso (CPU) a escala de todo el sistema calcula el tiempo relativo de CPU que se invierte en cada uno de los métodos Java y en todas las funciones del sistema que el programa Java utiliza. Emplee una definición del explorador de rendimiento (PEX) que rastree los eventos de ciclo de ejecución de desbordamiento del contador del supervisor de rendimiento (*PMCO). Las muestras suelen especificarse a intervalos de un milisegundo. Para recoger un perfil de rastreo válido, debe ejecutar la aplicación Java hasta que acumule de dos a tres minutos de tiempo de CPU. Esto debería generar más de 100.000 muestras. El mandato Imprimir informe del explorador de rendimiento (PRTPEXRPT) genera un histograma del tiempo de CPU invertido en toda la aplicación. Esto incluye todos los métodos Java y toda actividad a nivel de sistema. La herramienta Performance Data Collector (PDC) también proporciona información de perfil relacionada con los programas que se ejecutan en el servidor iSeries.

Nota: el perfilado de CPU no muestra el uso relativo de CPU de los programas Java interpretados.

Interfaz del perfilador de la máquina virtual Java

La interfaz del perfilador de la máquina virtual Java (JVMPI) es una interfaz experimental para el perfilado de la máquina virtual Java (JVM), que se ha presentado e implementado por primera vez en Java 2 SDK, Standard Edition (J2SDK), versión 1.2. de Sun.

El soporte de JVMPI coloca ganchos en la JVM y en el compilador Just-in-time (JIT) que, cuando se activan, proporcionan información de eventos a un agente de perfilado.

El agente de perfilado se implementa como un programa de servicio del entorno de lenguajes integrados (ILE).

El perfilador envía información de control a la JVM para habilitar e inhabilitar eventos JVMPI. Por ejemplo, el perfilador puede no estar interesado en ganchos de entrada o salida de método, y puede indicar a la JVM que no desea recibir estas notificaciones de eventos. La JVM y JIT tienen eventos JVMPI incorporados que envían notificaciones de eventos al agente de perfilado si el evento está habilitado. El perfilador indica a la JVM qué eventos son de interés, y la JVM envía notificaciones de los eventos al perfilador cuando se producen.

El programa de servicio QSYS/QJVAJVMPI proporciona las funciones de JVMPI.

Para obtener más información, consulte JVMPI

de Sun Microsystems, Inc.

Recoger datos de rendimiento Java

Para recoger datos de rendimiento de Java en un servidor iSeries, siga estos pasos:

1. Cree una definición del explorador de rendimiento (PEX) que especifique:
   - Un nombre definido por el usuario
   - El tipo de recogida de datos
   - El nombre de trabajo
   - Los eventos del sistema sobre los que desea recoger información del sistema

   Nota: si la salida que desea es de tipo `java_g -prof` y sabe cuál es el nombre concreto del trabajo del programa Java, es preferible que la definición de PEX sea *STATS en lugar de *TRACE.

   A continuación se ofrece un ejemplo de una definición *STATS:

   ```
       ADDPEXDFN DFN(YOURDFN) JOB(*ALL/YOURID/QJVACMDSRV) DTAORG(*HIER)
   TEXT('su definición stats')
   ```

   Esta definición *STATS no obtiene todos los eventos Java en ejecución. Sólo se hace el perfilado de los eventos Java que estén en su sesión Java. Esta modalidad de operación puede incrementar el tiempo que se tarda en ejecutar el programa Java.

   A continuación se ofrece un ejemplo de una definición *TRACE:

   ```
       ADDPEXDFN DFN(YOURDFN) TYPE(*TRACE) JOB(*ALL) TRCTYPE(*SLTEVT)
   SLTEVT(*YES) PGMEVT(*JVAENTRY *JVAEXIT)
   ```

   Esta definición *TRACE recoge los eventos Java de entrada y de salida de cualquier programa Java del sistema que cree con ENBPFRCOL(*ENTRYEXIT). Esto hace que el análisis de este tipo de recogida se realice con más lentitud que en el caso de un rastreo *STATS, en función de cuántos eventos de programa Java se tengan y de cuál sea la duración de la recogida de datos de PEX.
2. Habilite *JVAENTRY y *JVAEXIT dentro de la categoría de eventos de programa de la definición de PEX, de manera que PEX reconozca las entradas y salidas Java.

   Nota: si está ejecutando el código Java mediante el compilador Just-in-time (JIT), la entrada y la salida no se habilitan como lo haría si utilizase el mandato CRTJVAPGM para el proceso directo. En lugar de ello, JIT genera código con ganchos de entrada y salida cuando se utiliza la propiedad del sistema os400.enbprfcol.
3. Prepare el programa Java para que notifique los eventos de programa a iSeries Performance Data Collector. Para ello, puede utilizar el mandato Crear programa Java (CRTJVAPGM) en cualquier programa Java sobre el que desee notificar datos de rendimiento. Debe crear el programa Java utilizando el parámetro ENBPFRCOL(*ENTRYEXIT).

   Nota: debe repetir este paso para todos los programas Java sobre los que desee recoger datos de rendimiento. Si no lo hace, PEX no recogerá ningún dato de rendimiento y, al ejecutar la herramienta Java Performance Data Converter (JPDC), no se generará ninguna salida.
4. Inicie la recogida de datos de PEX con el mandato Arrancar explorador de rendimiento (STRPEX).
5. Ejecute el programa que desea analizar. Este programa no debe estar en un entorno de producción. Generará un volumen elevado de datos en poco tiempo. Debe limitar el tiempo de recogida a cinco minutos. Un programa Java que se ejecute durante ese tiempo genera muchos datos PEX del sistema. Si se recogen demasiados datos, se necesitará demasiado tiempo para procesarlos.
6. Finalice la recogida de datos de PEX con el mandato Finalizar explorador de rendimiento (ENDPEX).

Nota: si no es la primera vez que ha finalizado la recogida de datos de PEX, debe especificar *YES en sustituir archivo o, de lo contrario, no se guardarán los datos.

7. Ejecute la herramienta JPDC.
8. Conecte el directorio del sistema de archivos integrado con el visor que prefiera: java_g -prof o Jinsight. Puede copiar este archivo desde el servidor iSeries y utilizarlo como entrada de cualquier herramienta de perfilado que considere oportuna.

La herramienta Performance Data Collector

La herramienta Performance Data Collector (PDC) facilita información de perfil sobre los programas que se ejecutan en el servidor iSeries.

La opción de perfil del estándar industrial de la mayoría de las máquinas virtuales Java depende de la implementación de la función `java_g`. Esta es una versión especial de depuración de la máquina virtual Java, que ofrece la opción `-prof`. Esta opción se especifica en una llamada a un programa Java. Si se especifica, la máquina virtual Java genera un archivo de registro que contiene información sobre cuáles son los componentes del programa que están en funcionamiento mientras dura el programa. La máquina virtual Java genera esta información en tiempo real.

En el servidor iSeries, la función Explorador de rendimiento (PEX) analiza programas y eventos del sistema específicos de registro. Esta información se almacena en una base de datos DB2 y se recupera mediante funciones SQL. La información de PEX es el depósito de información específica de programa que genera datos de perfil Java. Estos datos de perfil son compatibles con la información de perfil de programa de `java_g -prof`. La herramienta Java Performance Data Converter (JPDC) proporciona información de perfil de programa y la salida de programa de `java_g -prof` para una herramienta concreta de IBM conocida como Jinsight.

Si desea obtener información sobre la manera de recoger datos de rendimiento Java, consulte la sección Recoger datos de rendimiento Java.

La herramienta Java Performance Data Converter

La herramienta Java Performance Data Converter (JPDC) proporciona una manera de crear datos de rendimiento Java acerca de los programas Java que se ejecutan en el servidor iSeries. Estos datos de rendimiento son compatibles con la salida de datos de rendimiento de la opción `java_g -prof` de la máquina virtual Java de Sun Microsystems, Inc., y con la salida de IBM Jinsight.

Nota: la herramienta JPDC no genera una salida legible. Para analizar los datos, utilice una herramienta de perfilado Java que acepte `java_g -prof` o los datos de Jinsight.

La herramienta JPDC accede a los datos del explorador de rendimiento (PEX) de iSeries que se almacenan en DB2/400 (mediante JDBC). Convierte los datos al tipo de rendimiento de Jinsight o al tipo de rendimiento general. A continuación, almacena el archivo de salida en una ubicación del sistema de archivos integrado especificada por el usuario.

Nota: para recoger datos de PEX mientras la aplicación Java especificada se ejecuta en un servidor iSeries, debe seguir los procedimientos adecuados de recogida de datos de PEX de iSeries. Debe establecer una definición de PEX que defina la entrada y la salida de un programa o un procedimiento de recogida y almacenamiento. Para obtener detalles acerca de cómo recoger datos de PEX y establecer una definición de PEX, consulte la publicación Performance Tools for iSeries, SC41-5340

.

Si desea obtener información sobre la manera de ejecutar JPDC, consulte la sección Ejecutar Java Performance Data Converter.

Para iniciar el programa JPDC, puede utilizar la interfaz de línea de mandatos de Qshell o el mandato Ejecutar Java (RUNJVA).

Ejecutar Java Performance Data Converter

Para ejecutar Java Performance Data Converter (JPDC) con el fin de realizar la recogida de datos de rendimiento, siga estos pasos.

1. Especifique el primer argumento de entrada, que es `general` para `java_g -prof` o `jinsight` para la salida de Jinsight.
2. Entre el segundo argumento de entrada, que es el nombre de la definición del explorador de rendimiento (PEX) que se ha utilizado para recoger los datos.

   Nota: debe limitar este nombre a cuatro o cinco caracteres debido a la utilización interna de las conexiones del nombre.
3. Entre el tercer argumento de entrada, que es el nombre del archivo que la herramienta JPDC genera. Este archivo generado se escribe en el directorio actual del sistema de archivos integrado. Para especificar el directorio actual del sistema de archivos integrado, utilice el mandato cd (PF4).
4. Especifique el cuarto argumento de entrada, que es el nombre de la entrada de directorio de bases de datos relacionales de sistema principal de iSeries. Para ver cuál es el nombre, utilice el mandato Trabajar con entrada de directorio de bases de datos relacionales (WRKRDBDIRE). Es la única base de datos relacional en la que se indica *LOCAL.

Para que este código funcione, el archivo /QIBM/ProdData/Java400/ext/JPDC.jar debe estar en la vía de acceso de clases Java del servidor iSeries.Cuando el programa acabe de ejecutarse, habrá un archivo de salida de tipo texto en el directorio actual.

Para ejecutar JPDC, utilice la línea de mandatos de iSeries o el entorno Qshell. Consulte el Ejemplo: ejecutar Java Performance Data Converter para obtener detalles.

Mandatos y herramientas de IBM Developer Kit para Java

Al utilizar IBM Developer Kit para Java, puede utilizar herramientas Java con el intérprete de Qshell o bien mandatos CL.

Si ya tiene experiencia en la programación Java, tal vez le resulte más cómodo utilizar las herramientas Java del intérprete de Qshell, ya que son parecidas a las herramientas que utilizaría con Java Development Kit de Sun Microsystems, Inc. En la sección Intérprete de Qshell hallará más información sobre cómo se utiliza el entorno Qshell.

Si es usted programador de iSeries, tal vez le interese utilizar los mandatos CL para Java más habituales del entorno del servidor iSeries. Si desea obtener más información sobre cómo se utilizan los mandatos CL y los mandatos de iSeries Navigator, siga leyendo.

Con IBM Developer Kit para Java puede utilizar cualquiera de los mandatos y herramientas que se indican a continuación:

- El entorno Qshell, que incluye las herramientas de desarrollo Java que se requieren habitualmente para el desarrollo de programas.
- El entorno CL, que contiene los mandatos CL necesarios para optimizar y gestionar programas Java.
- Los mandatos de iSeries Navigator, que también crean y ejecutan programas Java optimizados.

Herramientas Java soportadas por IBM Developer Kit para Java

IBM Developer Kit para Java da soporte a estas herramientas:

- “La herramienta ajar Java” en la página 250
-

La herramienta appletviewer Java (page “La herramienta appletviewer Java” en la página 250)

- “La herramienta extcheck Java” en la página 251
- “La herramienta idlj Java” en la página 251
- “La herramienta jar Java” en la página 251
- “La herramienta jarsigner Java” en la página 251
- “La herramienta javac Java” en la página 252
- “La herramienta javadoc Java” en la página 252
- “La herramienta javah Java” en la página 253
- La herramienta javakey Java
- “La herramienta javap Java” en la página 253
- “La herramienta native2ascii Java” en la página 254
-

“La herramienta orbd Java” en la página 254

- “La herramienta policytool Java” en la página 254
- “La herramienta rmic Java” en la página 255
- “La herramienta rmid Java” en la página 255
- “La herramienta rmiregistry Java” en la página 255
- “La herramienta serialver Java” en la página 255
-

“servertool Java” en la página 255

- “La herramienta tnameserv Java” en la página 256

Con algunas excepciones, las herramientas Java, excepto la herramienta ajar, soportan la sintaxis y las opciones documentadas por Sun Microsystems, Inc. Todas deben ejecutarse mediante el intérprete de Qshell.

Para iniciar el intérprete de Qshell, puede utilizar el mandato Arrancar Qshell (STRQSH o QSH). Cuando el intérprete de Qshell está en ejecución, aparece la pantalla Entrada de mandato QSH. En esta pantalla aparece la salida y los mensajes de los programas y herramientas Java que se ejecutan en Qshell. También puede leerse en ella la entrada de los programas Java. En El mandato java de Qshell hallará información más detallada.

Nota: las funciones de la entrada de mandatos de iSeries no están disponibles directamente desde Qshell. Para obtener una línea de mandatos, pulse F21 (Entrada de mandatos CL).

Herramientas Java

- “La herramienta ajar Java” en la página 250
- La herramienta appletviewer Java (page “La herramienta appletviewer Java” en la página 250)
- “La herramienta extcheck Java” en la página 251

- "La herramienta idlj Java" en la página 251
- "La herramienta jar Java" en la página 251
- "La herramienta jarsigner Java" en la página 251
- "La herramienta javac Java" en la página 252
- "La herramienta javadoc Java" en la página 252

La herramienta ajar Java: La herramienta `ajar` es una interfaz alternativa a la herramienta `jar` que sirve para crear y manipular archivos de archivado Java (JAR). La herramienta `ajar` permite manipular tanto los archivos JAR como los archivos ZIP.

Si necesita una interfaz ZIP o una interfaz UNZIP, utilice la herramienta `ajar` en lugar de la herramienta `jar`.

Al igual que la herramienta `jar`, la herramienta `ajar` elabora una relación del contenido de los archivos JAR, extrae el contenido de los archivos JAR, crea archivos JAR nuevos y da soporte a la mayoría de los formatos ZIP. Además, la herramienta `ajar` da soporte a la adición y supresión de archivos en los archivos JAR existentes.

La herramienta `ajar` está disponible en el intérprete de Qshell. Para obtener más detalles, consulte el apartado ajar - Archivado Java alternativo.

La herramienta appletviewer Java: La herramienta `appletviewer` Java permite ejecutar applets sin necesidad de tener un navegador Web. Es compatible con la herramienta `appletviewer` proporcionada por Sun Microsystems, Inc.

Para ejecutar la herramienta `appletviewer`, debe utilizar Native Abstract Window Toolkit (NAWT) y utilizar la clase `sun.applet.AppletViewer` o ejecutar la herramienta appletviewer en el intérprete de Qshell.

El siguiente es un ejemplo del uso de la clase `sun.applet.AppletViewer` y ejecutar el ejemplo demo TicTacToe. Para obtener información sobre cómo cargar los ejemplos demo, consulte el apartado Cómo extraer archivos de ejemplo.

En la línea de mandatos, entre:

```
cd '/home/MyUserID/demo/applets/TicTacToe'
```

Para JDK 1.3, emita el mandato:

```
JAVA CLASS(sun.applet.AppletViewer) PARM('example1.html')
PROP((os400.class.path.rawt 2)(java.version 1.3))
```

Para JDK 1.4, emita el mandato:

```
JAVA CLASS(sun.applet.AppletViewer) PARM('example1.html')
prop((os400.awt.native true)(java.version 1.4))
```

El siguiente es un ejemplo del uso de la herramienta `appletviewer` en el intérprete Qshell y ejecutar el ejemplo demo TicTacToe. Para obtener información sobre cómo cargar los ejemplos demo, consulte el apartado Cómo extraer archivos de ejemplo.

Los mandatos correspondientes serían:

```
cd /home/MyUserID/demo/applets/TicTacToe
```

Para JDK 1.3, emita el mandato:

```
Appletviewer -J-Dos400.class.path.rawt=2 -J-Djava.version=1.3 example1.html
```

Para JDK 1.4, emita el mandato:

```
Appletviewer -J-Dos400.awt.native=true -J-Djava.version=1.4 example1.html
```

Nota: -J son distintivos de ejecución para Appletviewer. -D son propiedades.

Para obtener más información sobre la herramienta `appletviewer`, le remitimos a la herramienta appletviewer de Sun Microsystems, Inc.

La herramienta extcheck Java: En Java 2 SDK (J2SDK), Standard Edition, versión 1.2 y posteriores, la herramienta `extcheck` detecta los conflictos de versión entre un archivo JAR destino y los archivos JAR de ampliación instalados actualmente. Es compatible con la herramienta `keytool` suministrada por Sun Microsystems, Inc.

La herramienta `extcheck` está disponible cuando se utiliza el intérprete de Qshell.

Si desea más información sobre la herramienta `extcheck`, vea Herramienta extcheck de Sun Microsystems, Inc.

La herramienta idlj Java: La herramienta `idlj` genera enlaces Java a partir de un archivo IDL (Interface Definition Language) determinado. La herramienta `idlj` también se conoce como compilador de IDL a Java. Es compatible con la herramienta `idlj` proporcionada por Sun Microsystems, Inc. Esta herramienta sólo funciona en Java Development Kits 1.3 y 1.4.

Para obtener más información acerca de la herramienta `idlj`, consulte la herramienta idlj de Sun Microsystems, Inc.

La herramienta jar Java: La herramienta `jar` combina varios archivos en un único archivo JAR. Es compatible con la herramienta `jar` proporcionada por Sun Microsystems, Inc.

La herramienta `jar` está disponible en el intérprete de Qshell.

Si desea emplear una interfaz alternativa a la herramienta `jar`, vea la “La herramienta ajar Java” en la página 250, que sirve para crear y manipular archivos JAR.

Para obtener más información acerca de los sistema de archivos de iSeries, consulte los apartados Sistema de archivos integrado o Archivos del sistema de archivos integrado.

Para obtener más información acerca de la herramienta `jar`, consulte la herramienta jar de Sun Microsystems, Inc.

La herramienta jarsigner Java: En Java 2 SDK (J2SDK), Standard Edition, versión 1.2 y posteriores, la herramienta `jarsigner` firma los archivos JAR y verifica las firmas de los archivos JAR firmados. La herramienta `jarsigner` accede al almacén de claves, creado y gestionado por la herramienta `keytool`, cuando tiene que localizar la clave privada para firmar un archivo JAR. En J2SDK, las herramientas `jarsigner` y `keytool` sustituyen a la herramienta `javakey`. Es compatible con la herramienta `jarsigner` proporcionada por Sun Microsystems, Inc.

La herramienta `jarsigner` está disponible en el intérprete de Qshell.

Si desea más información sobre la herramienta `jarsigner`, le remitimos a la herramienta jarsigner de Sun Microsystems, Inc.

La herramienta javac Java: La herramienta `javac` compila programas Java. Es compatible con la herramienta `javac` proporcionada por Sun Microsystems, Inc., con una excepción.

-classpath
No altera temporalmente la vía de acceso de clases por omisión. En lugar de ello, la opción se añade al final de la vía de acceso de clases por omisión del sistema. La opción `-classpath` altera temporalmente la variable de entorno CLASSPATH.

La herramienta `javac` está disponible en el intérprete de Qshell.

Si tiene instalado JDK 1.1.x en el servidor iSeries como valor por omisión, pero necesita ejecutar el mandato `java` de la versión 1.2 o superior, entre este mandato:

```
javac -djava.version=1.2 <mi_dir> MyProgram.java
```

Para obtener más información sobre la herramienta `javac`, consulte la herramienta javac de Sun Microsystems, Inc.

La herramienta javadoc Java: La herramienta `javadoc` genera documentación de API. Es compatible con la herramienta `javadoc` proporcionada por Sun Microsystems, Inc.

La herramienta `javadoc` está disponible en el intérprete de Qshell.

Para obtener más información sobre la herramienta `javadoc`, consulte la herramienta javadoc de Sun Microsystems, Inc.

Cómo extraer archivos de ejemplo: El procedimiento siguiente muestra una manera de extraer los archivos de ejemplo antes de ejecutar la herramienta appletviewer Java. El procedimiento presupone que desea extraer los archivos de ejemplo en el directorio inicial.

1. Entre el mandato Iniciar Qshell (QSH) en la línea de mandatos
2. Si aún no existe, cree un directorio del sistema de archivos integrado (IFS) a nivel inicial para su ID de usuario:
   ```
   mkdir /home/MyUserID
   ```
3. Cree un directorio demo dentro del directorio IFS
   ```
   mkdir /home/MyUserID/demo
   ```
4. Cambie de directorio al directorio demo
   ```
   cd /home/myUserId/demo
   ```
5. Para JDK 1.3, entre lo siguiente en la línea de mandatos para extraer los archivos demo:
   ```
   jar xf /QIBM/ProdData/Java400/jdk13/demo.zip
   ```
   Para JDK 1.4, utilice este mandato alternativo:
   ```
   jar xf
   /QIBM/ProdData/Java400/jdk14/demo.jar
   ```

Herramientas Java

- "La herramienta javah Java" en la página 253
- "La herramienta javap Java" en la página 253
- "La herramienta keytool Java" en la página 254
- "La herramienta native2ascii Java" en la página 254
-

»

"La herramienta orbd Java" en la página 254

«

- “La herramienta policytool Java” en la página 254
- “La herramienta rmic Java” en la página 255
- “La herramienta rmid Java” en la página 255
- “La herramienta rmiregistry Java” en la página 255
- “La herramienta serialver Java” en la página 255
- 
  “servertool Java” en la página 255
  
- “La herramienta tnameserv Java” en la página 256

La herramienta javah Java: La herramienta `javah` facilita la implementación de los métodos nativos Java. Es compatible con la herramienta `javah` proporcionada por Sun Microsystems, Inc., con algunas excepciones.

Nota: si se escriben métodos nativos, significa que la aplicación no es 100% puro Java. También significa que la aplicación no es portable directamente de una plataforma a otra. Los métodos nativos son, por naturaleza, específicos de una plataforma o un sistema. La utilización de métodos nativos puede incrementar el coste de desarrollo y mantenimiento de las aplicaciones.

La herramienta `javah` está disponible en el intérprete de Qshell. Lee un archivo de clase Java y crea un archivo de cabecera escrito en C dentro del directorio de trabajo actual. El archivo de cabecera que se escribe es un archivo continuo de iSeries (STMF). Para poder incluirlo en un programa C del servidor iSeries, primero debe copiarse en un miembro de archivo.

La herramienta `javah` es compatible con la herramienta proporcionada por Sun Microsystems, Inc. Sin embargo, si se especifican estas opciones, el servidor iSeries las pasa por alto.

-td La herramienta `javah` del servidor iSeries no requiere ningún directorio temporal.

-stubs En el servidor iSeries, Java únicamente da soporte al formato JNI (interfaz Java nativa) de los métodos nativos. Los apéndices solo se necesitaron para el formato anterior a JNI de los métodos nativos.

-trace
Guarda relación con la salida de archivo de apéndice .c, que no está soportada por Java en el servidor iSeries.

-v No está soportada.

Nota: se debe especificar siempre la opción -jni. El servidor iSeries no da soporte a las implementaciones de métodos nativos anteriores a JNI.

Para obtener más información sobre la herramienta `javah`, consulte la herramienta javah de Sun Microsystems, Inc.

La herramienta javap Java: La herramienta `javap` desensambla archivos Java compilados e imprime una representación del programa Java. Esto puede resultar útil cuando el código fuente original ha dejado de estar disponible en un sistema.

Es compatible con la herramienta `javap` proporcionada por Sun Microsystems, Inc., con algunas excepciones.

-b Se hace caso omiso de esta opción. La compatibilidad hacia atrás no es necesaria, puesto que, en el servidor iSeries, Java solamente da soporte a Java Development Kit 1.1.4 y versiones posteriores.

-p En el servidor iSeries, -p no es una opción válida. Debe escribirse -private.

-verify
Se hace caso omiso de esta opción. La herramienta `javap` no realiza verificación alguna en el servidor iSeries.

La herramienta `javap` está disponible en el intérprete de Qshell.

Nota: la utilización de la herramienta `javap` para desensamblar clases puede constituir una violación del acuerdo de licencia de dichas clases. Antes de utilizar la herramienta `javap`, consulte el acuerdo de licencia de las clases.

Para obtener más información sobre la herramienta `javap`, consulte la herramienta javap de Sun Microsystems, Inc.

La herramienta keytool Java: En Java 2 SDK (J2SDK), Standard Edition, versión 1.2 o superior, la herramienta `keytool` crea pares de claves públicas y privadas, certificados autofirmados, y gestiona almacenes de claves. En J2SDK, las herramientas `jarsigner` y `keytool` sustituyen a la herramienta `javakey`. Es compatible con la herramienta `keytool` suministrada por Sun Microsystems, Inc.

La herramienta `keytool` está disponible en el intérprete de Qshell.

Para obtener más información acerca de `keytool`, consulte el apartado keytool de Sun Microsystems, Inc.

La herramienta native2ascii Java: La herramienta `native2ascii` convierte un archivo que contenga caracteres nativos (caracteres que no son Latin-1 ni Unicode) en un archivo con caracteres Unicode. Es compatible con la herramienta `native2ascii` proporcionada por Sun Microsystems, Inc.

La herramienta `native2ascii` está disponible en el intérprete de Qshell.

Para obtener más información sobre la herramienta `native2ascii`, consulte la herramienta native2ascii de Sun Microsystems, Inc.

La herramienta orbd Java: La herramienta `orbd` proporciona soporte para clientes para localizar e invocar objetos persistentes de forma transparente en servidores del entorno CORBA. ORBD se utiliza en lugar del Servicio de denominación transitorio (tnameserv), que incluye un Servicio de denominación transitorio y un Servicio de denominación persistente. La herramienta `orbd` incorpora la funcionalidad de un Gestor de servidor, un Servicio de denominación interoperativo y un Servidor de nombres de rutina de carga. Cuando se utiliza conjuntamente con `servertool`, el Gestor de servidor localiza, registra y activa un servidor cuando un cliente desea acceder al servidor.

Para obtener más información sobre la herramienta `orbd`, consulte la herramienta orbd de Sun Microsystems, Inc.

La herramienta policytool Java: En Java 2 SDK, Standard Edition, la herramienta `policytool` crea y cambia los archivos de configuración de política externa que definen la política de seguridad Java de la instalación. Es compatible con la herramienta `policytool` proporcionada por Sun Microsystems, Inc.

`policytool` es una herramienta de interfaz gráfica de usuario (GUI) que está disponible cuando se utiliza el intérprete de Qshell y NAWT (Native Abstract Window Toolkit). Consulte Native Abstract Window Toolkit de IBM Developer Kit para Java para obtener más información.

Para obtener más información acerca de `policytool`, consulte el apartado policytool de Sun Microsystems, Inc.

La herramienta rmic Java: La herramienta `rmic` genera archivos de apéndice y archivos de clase para los objetos Java. Es compatible con la herramienta `rmic` proporcionada por Sun Microsystems, Inc.

La herramienta `rmic` está disponible en el intérprete de Qshell.

Para obtener más información sobre la herramienta `rmic`, consulte la herramienta rmic de Sun Microsystems, Inc.

La herramienta rmid Java: En Java 2 SDK (J2SDK), Standard Edition, la herramienta `rmid` inicia el daemon del sistema de activación para que los objetos se puedan registrar y activar en una máquina virtual Java. Es compatible con la herramienta `rmid` suministrada por Sun Microsystems, Inc.

La herramienta `rmid` está disponible en el intérprete de Qshell.

Para obtener más información sobre la herramienta `rmid`, consulte la herramienta rmid de Sun Microsystems, Inc.

La herramienta rmiregistry Java: La herramienta `rmiregistry` inicia un registro de objetos remotos en un puerto especificado. Es compatible con la herramienta `rmiregistry` proporcionada por Sun Microsystems, Inc.

La herramienta `rmiregistry` está disponible en el intérprete de Qshell.

Para obtener más información sobre la herramienta `rmiregistry`, consulte la herramienta rmiregistry de Sun Microsystems, Inc.

La herramienta serialver Java: La herramienta `serialver` devuelve el número de versión o el identificador exclusivo de serialización correspondiente a una o varias clases. Es compatible con la herramienta `serialver` proporcionada por Sun Microsystems, Inc.

La herramienta `serialver` está disponible en el intérprete de Qshell.

Para obtener más información sobre la herramienta `serialver`, consulte la herramienta serialver de Sun Microsystems, Inc.

servertool Java: `servertool` proporciona una interfaz de línea de mandatos para que los programadores de aplicaciones registren, eliminen el registro, arranquen y concluyan un servidor persistente.

Para obtener más información acerca de `servertool`, consulte el apartado servertool de Sun Microsystems, Inc.

La herramienta tnameserv Java: En Java 2 SDK (J2SDK), Standard Edition, versión 1.3 o superior, la herramienta `tnameserv` (Servicio de denominación transitorio) proporciona acceso al servicio de denominación. Es compatible con la herramienta `tnameserv` proporcionada por Sun Microsystems, Inc.

La herramienta `tnameserv` está disponible en el intérprete de Qshell.

El mandato java de Qshell

El mandato `java` de Qshell ejecuta programas Java. Es compatible con la herramienta `java` proporcionada por Sun Microsystems, Inc., con algunas excepciones.

IBM Developer Kit para Java hace caso omiso de las siguientes opciones del mandato `java` de Qshell:

Opción	Descripción
-cs	Esta opción no está soportada.
-checksource	Esta opción no está soportada.
-debug	Esta opción está soportada por el depurador interno de iSeries.
-noasyncgc	La recogida de basura siempre se está ejecutando con IBM Developer Kit para Java.
-noclassgc	La recogida de basura siempre se está ejecutando con IBM Developer Kit para Java.
-prof	El servidor tiene herramientas de rendimiento propias.
-ss	Esta opción no es aplicable al servidor iSeries.
-oss	Esta opción no es aplicable al servidor iSeries.
-t	El servidor iSeries utiliza una función de rastreo propia.
-verify	Verificar siempre en el servidor iSeries.
-verifyremote	Verificar siempre en el servidor iSeries.
-noverify	Verificar siempre en el servidor iSeries.

En el servidor iSeries, la opción `-classpath` no altera temporalmente la vía de acceso de clases por omisión. En lugar de ello, la opción se añade al final de la vía de acceso de clases por omisión del sistema. La opción `-classpath` altera temporalmente la variable de entorno CLASSPATH.

El mandato `java` de Qshell da soporte a las opciones nuevas del servidor iSeries. Las nuevas opciones soportadas son estas:

Opción	Descripción
-chkpath	Esta opción comprueba si existe acceso público de escritura a los directorios de la variable CLASSPATH.
-opt	Esta opción especifica el nivel de optimización.
-Xrun[:]	Se visualiza un mensaje que indica un programa de servicio y una serie de parámetro opcional para la función JVM_OnLoad durante el inicio de la JVM.
 -agentlib:	Indica un programa de servicio OS/400 que contiene un agente VM. La VM intenta cargar el programa de servicio desde una biblioteca OS/400 incluida en la lista de bibliotecas de OS/400 durante el inicio. 

Opción	Descripción
 -agentpath:	Carga la biblioteca desde la vía de acceso absoluta que sigue a esta opción. No se produce la expansión del nombre de biblioteca y las opciones pasan al agente durante el inicio.

El mandato Ejecutar Java (RUNJVA) en la información de consulta de mandatos CL describe estas nuevas opciones de forma detallada. Los apartados dedicados a los mandatos Crear programa Java (CRTJVAPGM), Suprimir programa Java (DLTJVAPGM) y Visualizar programa Java (DSPJVAPGM) en la información de consulta de mandatos CL contienen información sobre la gestión de programas Java.

El mandato `java` de Qshell está disponible en el intérprete de Qshell.

Para obtener más información sobre el mandato `java` de Qshell, le remitimos a la herramienta java de Sun Microsystems, Inc.

Mandatos CL soportados por Java

IBM Developer Kit para Java da soporte a estos mandatos CL:

- El mandato Analizar programa Java (ANZJVAPGM) analiza un programa Java, lista sus clases y muestra el estado actual de cada clase.
- El mandato Analizar máquina virtual Java (ANZJVM) recupera y establece información en una máquina virtual Java (JVM). Este mandato la ayuda a depurar programas Java devolviendo información acerca de las clases activas.
- El mandato Cambiar programa Java (CHGJVAPGM) cambia los atributos de un programa Java.
- El mandato Crear programa Java (CRTJVAPGM) crea un programa Java en un servidor iSeries a partir de un archivo de clase, un archivo ZIP o un archivo JAR.
- El mandato Suprimir programa Java (DLTJVAPGM) suprime un programa Java de iSeries asociado a un archivo de clase Java, un archivo ZIP o un archivo JAR.
- El mandato Visualizar programa Java (DSPJVAPGM) visualiza información acerca de un programa Java en iSeries.
- El mandato Volcar máquina virtual Java (DMPJVM) vuelca información acerca de la máquina virtual Java para un trabajo especificado en un archivo de impresora en spool.
- Los mandatos JAVA y Ejecutar Java (RUNJVA) ejecutan programas Java de iSeries.

Para obtener más información, consulte las siguientes páginas:

Consideraciones sobre el uso del mandato ANZJVM

Series del parámetro de opción de Código interno bajo licencia

API de programa y mandato CL

Consideraciones sobre el uso del mandato ANZJVM

Debido a la posible duración de la ejecución de ANZJVM, es muy posible que una JVM finalice antes de que ANZJVM pueda finalizar. Si la JVM finaliza, ANZJVM devuelve el mensaje JVAB606 (es decir, la JVM ha finalizado mientras se procesaba ANZJVM) junto con los datos que ha podido obtener.

Tampoco existe límite superior en cuanto al número de clases que una JVM puede manejar. Si existen más clases que las que han podido manejarse, ANZJVM debe devolver los datos que pueden manejarse

junto con un mensaje indicando que existe información adicional de la que no se ha informado. Si los datos requieren truncamiento, ANZJVM devuelve tanta información como es posible.

El parámetro interno está restringido a 3600 segundos (una hora) de duración. El número de clases acerca de las que ANZJVM puede devolver información está limitado por la cantidad de almacenamiento del sistema.

Depurar programas Java que se ejecutan en el servidor

Tiene varias opciones para depurar y resolver problemas de los programas Java que se ejecuten en el servidor. La siguiente información no es una valoración completa de las posibilidades pero enumera diversas opciones.

Una de las maneras más fáciles de depurar programas Java que se ejecuten en el servidor iSeries es utilizar el IBM iSeries System Debugger. IBM iSeries System Debugger proporciona una interfaz gráfica de usuario (GUI) que le permite utilizar con mayor facilidad las posibilidades de depuración del servidor iSeries.

Puede utilizar la pantalla interactiva del servidor para depurar programas Java, aunque el iSeries System Debugger proporciona una GUI de más fácil utilización que le permite realizar las mismas funciones.

Adicionalmente la Máquina virtual Java (JVM) de iSeries da soporte al protocolo JDWP (Java Debug Wire Protocol), que forma parte de la arquitectura Java Platform Debugger. Los depuradores habilitados para JDWP le permiten realizar la depuración remota desde clientes que se ejecutan en sistemas operativos distintos. (El IBM iSeries Debugger también le permite realizar la depuración remota de forma similar, aunque no utiliza JDWP.) Un programa de ese tipo, habilitado para JDWP, es el depurador de Java en la plataforma de herramientas universales del proyecto Eclipse.

Si el rendimiento del programa se degrada al ejecutarse durante mucho más tiempo, es posible que haya codificado una fuga de memoria erróneamente. Puede utilizar Java Watcher como ayuda para depurar el programa y localizar las fugas de memoria realizando el análisis del almacenamiento dinámico de aplicaciones Java y el perfilado de creación de objeto.

Para obtener más detalles sobre las opciones de depuración anteriores, consulte la siguiente información:

IBM iSeries System Debugger

Depurar programas mediante IBM Developer Kit para Java

Arquitectura del depurador de la plataforma Java

Debugger Java Development Tool

del sitio web del proyecto Eclipse

JavaWatcher

Depurar programas Java desde una línea de mandatos de OS/400

Para depurar programas Java desde la línea de mandatos de OS/400, seleccione una de las opciones siguientes:

- Depurar un programa Java
- Depurar programas Java y programas de métodos nativos
- Depurar un programa Java desde otra pantalla
- Depurar clases Java cargadas mediante un cargador de clases personalizadas
- Depurar servlets

Cuando se depura un programa Java, este se ejecuta en realidad dentro de la máquina virtual Java en un trabajo inmediato de proceso por lotes (BCI). El código fuente aparece en la pantalla interactiva, pero el programa no se ejecuta en ella. Se ejecuta en el otro trabajo, que es un trabajo al que se da servicio. Cuando finaliza el programa Java, finaliza también el trabajo al que se da servicio y se visualiza un mensaje que indica que `el trabajo al que se ha dado servicio ha finalizado`.

No es posible depurar programas Java ejecutados con el compilador Just-In-Time (JIT). Si un archivo no tiene un programa Java asociado, el valor por omisión es ejecutar el JIT. Este valor puede inhabilitarse de varias formas para permitir la depuración:

- Especifique la propiedad `java.compiler=NONE` al iniciar la máquina virtual Java.
- Especifique OPTION(*DEBUG) en el mandato Ejecutar Java (RUNJVA).
- Especifique INTERPRET(*YES) en el mandato Ejecutar Java (RUNJVA).
- Utilice CRTJVAPGM OPTIMIZATION(10) para crear un programa Java asociado antes de iniciar la máquina virtual Java.

  Nota: Ninguna de estas soluciones afecta a una máquina virtual Java en ejecución. Si una máquina virtual Java no se ha iniciado con una de estas alternativas, debe detenerse y reiniciarse para que pueda depurarse.

La interfaz entre los dos trabajos se establece al especificar la opción *DEBUG en el mandato Ejecutar Java (RUNJVA).

Para obtener más información acerca del depurador del sistema, consulte la publicación WebSphere Development Studio: ILE C/C++ Programmer's Guide, SC09-2712-04

y la información de ayuda en línea.

Depurar un programa Java

La manera más fácil de depurar programas Java que se ejecuten en el servidor iSeries es utilizar el IBM iSeries System Debugger. El IBM iSeries System Debugger proporciona una interfaz gráfica de usuario que le permite utilizar con mayor facilidad las posibilidades de depuración del servidor iSeries.

Para obtener más información sobre cómo utilizar el iSeries System Debugger para depurar y probar programas que se ejecuten en el servidor iSeries, consulte el apartado IBM iSeries System Debugger.

Si lo desea, puede utilizar la pantalla interactiva del servidor para utilizar la opción `*DEBUG` para ver el código fuente antes de ejecutar el programa. A continuación, se pueden establecer puntos de interrupción o bien emitir mandatos de recorrer principal o recorrer todo en un programa con el fin de analizar los errores mientras se ejecuta el programa.

Para depurar programas Java, siga estos pasos:

1. Compile el programa Java con la opción DEBUG, que es la opción -g de la herramienta `javac`. Consulte la sección Depurar programas Java mediante la opción *DEBUG para obtener más detalles.
2. Inserte el archivo de clase (.class) y el archivo fuente (.java) en el mismo directorio del servidor iSeries.
3. Ejecute el programa Java utilizando el mandato Ejecutar Java (RUNJVA) en la línea de mandatos de iSeries. Especifique OPTION(*DEBUG) en el mandato Ejecutar Java (RUNJVA).

   Nota: sólo puede depurarse una clase. Si se especifica un nombre de archivo JAR para la palabra clave CLASS, OPTION(*DEBUG) no está soportado.
4. Se visualizará el fuente del programa Java.
5. Pulse F6 (Añadir/Borrar punto de interrupción), para establecer puntos de interrupción, o F10 (Recorrer), para recorrer paso a paso el programa. Para obtener más información acerca de cómo establecer puntos de interrupción, consulte la sección Establecer puntos de interrupción. Para obtener detalles acerca de los mandatos de recorrido, consulte la sección Recorrer paso a paso los programas Java para la depuración.

Consejos:

1. Mientras utiliza los puntos de interrupción y los mandatos de recorrer, compruebe el flujo lógico del programa Java y, a continuación, vea las variables y cámbielas según convenga.
2. La utilización de OPTION(*DEBUG) en el mandato RUNJVA inhabilita el compilador Just-In-Time (JIT). Los archivos que no tienen un programa Java asociado se ejecutan en modalidad interpretada.

Depurar programas Java mediante la opción *DEBUG: La opción *DEBUG sirve para ver el código fuente antes de ejecutar el programa. Permite establecer puntos de interrupción dentro del código.

Para utilizar la opción *DEBUG, entre el mandato Ejecutar Java (RUNJVA) seguido del nombre del archivo de clase y OPTION(*DEBUG) en la línea de mandatos. Por ejemplo, la línea de mandatos de iSeries debe ser así:

```
RUNJVA CLASS(nombreclase) OPTION(*DEBUG)
```

Nota: si no tiene autorización para utilizar el mandato Arrancar trabajo de servicio (STRSRVJOB), se hará caso omiso de OPTION(*DEBUG).

Para ver las pantallas de depuración, consulte la sección Pantallas iniciales de depuración de programas Java.

La manera más fácil de depurar programas Java que se ejecuten en el servidor iSeries es utilizar el IBM iSeries System Debugger. El IBM iSeries System Debugger proporciona una interfaz gráfica de usuario que le permite utilizar con mayor facilidad las posibilidades de depuración del servidor iSeries.

Para obtener más información sobre cómo utilizar el iSeries System Debugger para depurar y probar programas que se ejecuten en el servidor iSeries, consulte el apartado IBM iSeries System Debugger.

Pantallas iniciales de depuración de programas Java: Al depurar los programas Java, siga estas pantallas de ejemplo para sus programas. En ellas aparece un programa de ejemplo llamado Hellod.

- Entre `ADDENVVAR ENVVAR(CLASSPATH) VALUE ('/MIDIR')`.
- Entre este mandato: `RUNJVA CLASS(HELLOD) OPTION(*DEBUG)`. Inserte el nombre del programa Java en lugar de HELLOD.

- Espere a que se visualice la pantalla Visualizar fuente de módulo. Es el fuente del programa Java HELLOD.

```
+------------------------------------------------------------------------------+
|                    Visualizar fuente de módulo                               |
|                                                                              |
|  Nombre archivo clase:   HELLOD                                              |
|       1  import java.lang.*;                                                 |
|       2                                                                      |
|       3  public class Hellod extends Object                                  |
|       4  {                                                                   |
|       5  int k;                                                              |
|       6  int l;                                                              |
|       7  int m;                                                              |
|       8  int n;                                                              |
|       9  int o;                                                              |
|      10  int p;                                                              |
|      11  String myString;                                                    |
|      12  Hellod myHellod;                                                    |
|      13  int myArray[];                                                      |
|      14                                                                      |
|      15  public Hellod()                                                     |
|                                                                    Más...    |
|  Depurar . . .                                                               |
|                                                                              |
|  F3=Fin programa  F6=Añadir/Borrar pto interrup  F10=Recorrer F11=Ver variable |
|  F12=Reanudar  F17=Observar var  F18=Trabajar con pto observ  F24=Más teclas |
|                                                                              |
+------------------------------------------------------------------------------+
```

- Pulse F14 (Trabajar con lista de módulos).
- Aparece la pantalla Trabajar con lista de módulos. Puede añadir otras clases y otros programas para depurar entrando la opción 1 (Añadir programa). Para visualizar el fuente de los módulos, utilice la opción 5 (Visualizar fuente de módulo).

```
+------------------------------------------------------------------------------+
|                         Trabajar con lista de módulos                        |
|                                                  Sistema:   AS400            |
|  Teclee opciones, pulse Intro.                                               |
|    1=Añadir programa   4=Eliminar programa   5=Visualizar fuente de módulo   |
|    8=Trabajar con puntos de interrupción de módulo                           |
|                                                                              |
|  Opc     Progr/módulo       Biblioteca       Tipo                            |
|                             *LIBL            *SRVPGM                         |
|          HELLOD                              *CLASS      Seleccionado        |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                    Final     |
|  Mandato                                                                     |
|  ===>                                                                        |
|  F3=Salir   F4=Solicitud   F5=Renovar   F9=Recuperar   F12=Cancelar          |
|  F22=Visualizar nombre de archivo de clase                                   |
|                                                                              |
+------------------------------------------------------------------------------+
```

- Cuando añada una clase para depurar, puede que necesite entrar un nombre de clase calificado por paquete cuya longitud supere la del campo de entrada Programa/módulo. Para entrar un nombre de mayor longitud, siga estos pasos:
  1. Entre la opción 1 (Añadir programa).
  2. Deje en blanco el campo Programa/módulo.

3. Deje el campo Biblioteca como *LIBL.
4. Entre *CLASS en Tipo.
5. Pulse Intro.
6. Se visualiza una pantalla emergente, en la que tiene más espacio para especificar el nombre de archivo de clase calificado por paquete.

Establecer puntos de interrupción: Los puntos de interrupción permiten controlar la ejecución de un programa. Los puntos de interrupción detienen la ejecución de un programa en una sentencia determinada.

Para establecer puntos de interrupción, lleve a cabo los siguientes pasos:

1. Sitúe el cursor en la línea de código en la que desee establecer un punto de interrupción.
2. Pulse F6 (Añadir/Borrar punto de interrupción) para establecer el punto de interrupción.
3. Pulse F12 (Reanudar) para ejecutar el programa.

Nota: justo antes de que se ejecute la línea de código en la que está establecido el punto de interrupción, se visualiza el fuente del programa para indicar que se ha llegado al punto de interrupción.

```
+------------------------------------------------------------------------------+
|                    Visualizar fuente de módulo                               |
|                                                                              |
|Hebra actual:   00000019     Hebra detenida:   00000019                       |
|Nombre archivo clase:   Hellod                                                |
|35  public static void main(String[] args)                                    |
|36      {                                                                     |
|37    int i,j,h,B[],D[][];                                                    |
|38      Hellod A=new Hellod();                                                |
|39      A.myHellod = A;                                                       |
|40      Hellod C[];                                                           |
|41     C = new Hellod[5];                                                     |
|42      for (int counter=0; counter<2; counter++) {                           |
|43        C[counter] = new Hellod();                                          |
|44        C[counter].myHellod = C[counter];                                   |
|45      }                                                                     |
|46      C[2] = A;                                                             |
|47      C[0].myString = null;                                                 |
|48      C[0].myHellod = null;                                                 |
|                                                                              |
|49      A.method1();                                                          |
|  Depurar . . .                                                               |
|                                                                              |
|F3=Fin programa  F6=Añadir/Borrar pto interrup  F10=Recorrer F11=Ver variable |
|F12=Reanudar  F17=Observar var  F18=Trabajar con pto observ  F24=Más teclas   |
|Se ha añadido el punto de interrupción a la línea 41.                         |
+------------------------------------------------------------------------------+
```

Cuando llegue a un punto de interrupción, si desea establecer puntos de interrupción a los que solo se llegue dentro de la hebra actual, utilice el mandato TBREAK.

Para obtener más información acerca los mandatos del depurador del sistema, consulte la publicación WebSphere Development Studio: ILE C/C++ Programmer's Guide, SC09-2712

y la información de ayuda en línea.

Para obtener información acerca de la evaluación de variables cuando un programa deja de ejecutarse en un punto de interrupción, consulte la sección Evaluar variables en programas Java.

Recorrer los programas Java que deben depurarse: Puede recorrer paso a paso el programa mientras lo depura. Puede recorrer la función principal o bien otras funciones del mismo. Los programas Java y los métodos nativos pueden utilizar la función de recorrido (step).

Cuando aparezca el fuente del programa por primera vez, podrá empezar a recorrer. El programa se detendrá antes de ejecutar la primera sentencia. Pulse F10 (Recorrer). Siga pulsando F10 (Recorrer) para recorrer paso a paso el programa. Pulse F22 (Recorrer todo) para recorrer cualquier función a la que llame el programa. También puede empezar a recorrer siempre que se llegue a un punto de interrupción. Para obtener información acerca de cómo establecer puntos de interrupción, consulte la sección Establecer puntos de interrupción.

```
+------------------------------------------------------------------------------+
|                    Visualizar fuente de módulo                               |
|                                                                              |
|Hebra actual:   00000019     Hebra detenida:   00000019                       |
|Nombre archivo clase:   Hellod                                                |
|35  public static void main(String[] args)                                    |
|36      {                                                                     |
|37    int i,j,h,B[],D[][];                                                    |
|38      Hellod A=new Hellod();                                                |
|39      A.myHellod = A;                                                       |
|40      Hellod C[];                                                           |
|41     C = new Hellod[5];                                                     |
|42      for (int counter=0; counter<2; counter++) {                           |
|43        C[counter] = new Hellod();                                          |
|44        C[counter].myHellod = C[counter];                                   |
|45      }                                                                     |
|46      C[2] = A;                                                             |
|47      C[0].myString = null;                                                 |
|48      C[0].myHellod = null;                                                 |
|49      A.method1();                                                          |
|  Depurar . . .                                                               |
|                                                                              |
|F3=Fin programa  F6=Añadir/Borrar pto interrup  F10=Recorrer F11=Ver variable |
|F12=Reanudar  F17=Observar var  F18=Trabajar con pto observ  F24=Más teclas   |
|Recorrer completado en la línea 42 de la hebra 00000019                       |
+------------------------------------------------------------------------------+
```

Para dejar de recorrer y seguir ejecutando el programa, pulse F12 (Reanudar).

Para obtener más información acerca del proceso de recorrer, consulte la publicación WebSphere Development Studio: ILE C/C++ Programmer's Guide, SC09-2712

y la información de ayuda en línea.

Para obtener información acerca de la evaluación de variables cuando un programa se detiene en un punto de recorrido, consulte la sección Evaluar variables en programas Java.

Evaluar variables en programas Java: Existen dos formas de evaluar una variable cuando un programa detiene su ejecución en un punto de interrupción o en una parte del recorrido:

- Entre `EVAL NombreVariable` en la línea de mandatos de depuración.
- Sitúe el cursor en el nombre de la variable dentro del código fuente visualizado y pulse F11 (Visualizar variable).

Para evaluar las variables de un programa Java, utilice el mandato EVAL.

Nota: con el mandato EVAL, también se puede cambiar el contenido de una variable. Para obtener más información acerca de las variaciones del mandato EVAL, consulte la publicación WebSphere Development Studio: ILE C/C++ Programmer's Guide, SC09-2712

y la información de ayuda en línea.

Cuando examine las variables de un programa Java, tenga presente lo siguiente:

- Si evalúa una variable que es una instancia de una clase Java, la primera línea de la pantalla muestra el tipo de objeto de que se trata. También muestra el identificador del objeto. A continuación de la primera línea de la pantalla, se visualiza el contenido de cada uno de los campos del objeto. Si la variable es nula, la primera línea de la pantalla indica que lo es. El contenido de cada uno de los campos (de un objeto nulo) se muestra por medio de asteriscos.
- Si evalúa una variable que es un objeto de tipo serie Java, se visualiza el contenido de la serie. Si la serie es nula, se visualiza null.
- No se pueden cambiar las variables que son de tipo serie.
- Si evalúa una variable que es una matriz, se visualiza 'ARR' seguido del identificador de la matriz. Para evaluar los elementos de la matriz, puede utilizar un subíndice del nombre de variable. Si la matriz es nula, se visualiza null.
- No se pueden cambiar las variables que son de tipo matriz. Se puede cambiar un elemento de una matriz si no se trata de una matriz de series o de objetos.
- En el caso de las variables de tipo matriz, se puede especificar arrayname.length para ver cuántos elementos hay en la matriz.
- Si desea ver el contenido de una variable que es un campo de una clase, puede especificar classvariable.fieldname.
- Si intenta evaluar una variable antes de que se haya inicializado, pueden ocurrir dos cosas. O bien aparece un mensaje según el cual `la variable no está disponible para visualizarse` o bien se muestra el contenido de la variable no inicializada, que podría ser un valor extraño.

Depurar programas Java y programas de métodos nativos

Pueden depurarse programas Java y programas de métodos nativos a la vez. Mientras se depura el fuente en la pantalla interactiva, se puede depurar un método nativo programado en C que esté dentro de un programa de servicio (*SRVPGM). El *SRVPGM debe compilarse y crearse con datos de depuración.

La manera más fácil de depurar programas Java y programas de método nativo (o programas de servicio) es utilizar el IBM iSeries System Debugger. El IBM iSeries System Debugger proporciona un entorno de depuración gráfico de usuario en el servidor iSeries. Para obtener más información sobre cómo utilizar el iSeries System Debugger para depurar y probar programas que se ejecuten en el servidor iSeries, consulte el apartado IBM iSeries System Debugger.

Para utilizar la pantalla interactiva del servidor para depurar programas Java y programas de método nativo a la vez, complete los pasos siguientes:

1. Pulse F14 (Trabajar con lista de módulos) cuando aparezca el fuente del programa Java para visualizar la pantalla Trabajar con lista de módulos (WRKMODLST).
2. Seleccione la opción 1 (Añadir programa) para añadir el programa de servicio.
3. Seleccione la opción 5 (Visualizar fuente del módulo) para visualizar el objeto *MODULE que desea depurar y el fuente.
4. Pulse F6 (Añadir/Borrar punto de interrupción), para establecer puntos de interrupción en el programa de servicio. Para obtener más información acerca de cómo establecer puntos de interrupción, consulte la sección Establecer puntos de interrupción.

5. Pulse F12 (Reanudar) para ejecutar el programa.

Nota: cuando se llega al punto de interrupción en el programa de servicio, se detiene la ejecución del programa y se visualiza el fuente del programa de servicio.

Depurar un programa Java desde otra pantalla

La manera más fácil de depurar programas Java que se ejecuten en el servidor iSeries es utilizar el IBM iSeries System Debugger. El IBM iSeries System Debugger proporciona una interfaz gráfica de usuario que le permite utilizar con mayor facilidad las posibilidades de depuración del servidor iSeries.

Para obtener más información sobre cómo utilizar el iSeries System Debugger para depurar y probar programas que se ejecuten en el servidor iSeries, consulte el apartado IBM iSeries System Debugger.

Al depurar un programa Java utilizando la pantalla interactiva del servidor, se visualiza el fuente del programa cada vez que éste se encuentra con un punto de interrupción. Esto puede interferir con la salida de pantalla del programa Java. Para evitarlo, depure el programa Java desde otra pantalla. La salida del programa Java se visualiza donde se ejecuta el mandato Java, y el fuente del programa se visualiza en la otra pantalla.

También es posible depurar de esta forma un programa Java que ya está en ejecución, siempre que éste no utilice el compilador Just-In-Time (JIT).

Para depurar Java desde otra pantalla, haga lo siguiente:

1. El programa Java debe estar retenido mientras se inicia la preparación de la depuración. Para retener el programa Java, puede hacer que el programa:
   - Espere a que se produzca una entrada desde el teclado.
   - Espere durante un intervalo de tiempo.
   - Entre en un bucle para comprobar una variable, lo que requiere que usted haya establecido un valor para sacar el programa Java del bucle.
2. Una vez retenido el programa Java, vaya a otra pantalla y siga estos pasos:
   a. Entre el mandato Trabajar con trabajos activos (WRKACTJOB) en la línea de mandatos.
   b. Busque el trabajo inmediato de proceso por lotes (BCI) en el que se está ejecutando el programa Java. Busque QJVACMDSRV en el listado Subsistema/trabajo. Busque su ID de usuario en el listado Usuario. Busque BCI bajo Tipo.
   c. Entre la opción 5 para trabajar con el trabajo.
   d. En la parte superior de la pantalla Trabajar con trabajo, figura el número, el usuario y el trabajo. Entre `STRSRVJOB Número/Usuario/Trabajo`.
   e. Entre `STRDBG CLASS(nombreclase)`. Nombreclase es el nombre de la clase Java que desea depurar. Puede tratarse del nombre de clase que ha especificado en el mandato Java o de otra clase.
   f. El fuente de dicha clase aparece en la pantalla Visualizar fuente de módulo.
   g. Establezca puntos de interrupción, pulsando F6 (Añadir/Borrar punto de interrupción), allí donde desee detenerse dentro de la clase Java. Pulse F14 para añadir más clases, programas o programas de servicio que depurar. Para obtener más información acerca de cómo establecer puntos de interrupción, consulte la sección Establecer puntos de interrupción.
   h. Pulse F12 (Reanudar) para seguir ejecutando el programa.
3. Deje de retener el programa Java original. Cuando se llegue a un punto de interrupción, aparecerá la pantalla Visualizar fuente de módulo en la pantalla en la que se hayan entrado los mandatos Arrancar programa de servicio (STRSRVJOB) y Arrancar depuración (STRDBG). Cuando finalice el programa Java, aparecerá el mensaje `El trabajo al que se ha dado servicio ha finalizado`.

4. Entre el mandato Finalizar depuración (ENDDBG).
5. Entre el mandato Finalizar trabajo de servicio (ENDSRVJOB).

Nota: asegúrese de inhabilitar el compilador Just-In-Time (JIT) al iniciar la máquina virtual Java en el trabajo original. Puede hacerlo con la propiedad `java.compiler=NONE`. Si el JIT está en funcionamiento durante la depuración, pueden producirse resultados inesperados.

En Variable de entorno QIBM_CHILD_JOB_SNDINQMSG hallará más información sobre esta variable, que controla si el trabajo BCI queda en espera antes de llamar a la máquina virtual Java.

Variable de entorno QIBM_CHILD_JOB_SNDINQMSG: La variable de entorno QIBM_CHILD_JOB_SNDINQMSG es la que controla si el trabajo inmediato de proceso por lotes (BCI), en el que se ejecuta la máquina virtual Java, espera antes de iniciar la máquina virtual Java.

Si establece la variable de entorno en 1 al ejecutar el mandato Ejecutar Java (RUNJVA), se envía un mensaje a la cola de mensajes del usuario. El mensaje se envía antes de que se inicie la máquina virtual Java en el trabajo BCI. Es similar al siguiente:

```
El proceso (hijo) engendrado 023173/JOB/QJVACMDSRV está detenido (G C)
```

Para ver este mensaje, entre SYSREQ y seleccione la opción 4.

El trabajo BCI espera hasta usted entre una respuesta al mensaje. Si la respuesta es (G), se inicia la máquina virtual Java.

Antes de responder al mensaje, puede establecer puntos de interrupción en el programa *SRVPGM o *PGM al que llamará el trabajo BCI.

Nota: no puede establecer puntos de interrupción en una clase Java, porque en este momento todavía no se ha iniciado la máquina virtual Java.

Depurar clases Java cargadas mediante un cargador de clases personalizadas

La manera más fácil de depurar programas Java que se ejecuten en el servidor iSeries es utilizar el IBM iSeries System Debugger. El IBM iSeries System Debugger proporciona una interfaz gráfica de usuario que le permite utilizar con mayor facilidad las posibilidades de depuración del servidor iSeries.

Para obtener más información sobre cómo utilizar el iSeries System Debugger para depurar y probar programas que se ejecuten en el servidor iSeries, consulte el apartado IBM iSeries System Debugger.

Para utilizar la pantalla interactiva del servidor para depurar una clase cargada mediante un cargador de clases personalizadas, lleve a cabo los siguientes pasos:

1. Establezca la variable de entorno DEBUGSOURCEPATH en el directorio que contiene el código fuente o, en el caso de una clase calificada por paquete, en el directorio inicial de los nombres de paquete.

   Por ejemplo, si el cargador de clases personalizadas carga clases ubicadas bajo el directorio /MYDIR, haga lo siguiente:

   ```
   ADDENVVAR ENVVAR(DEBUGSOURCEPATH) VALUE('/MYDIR')
   ```

2. Añada la clase a la vista de depuración desde la pantalla Visualizar fuente de módulo.

   Si la clase ya se ha cargado en la máquina virtual Java (JVM), añada simplemente la *CLASS como es habitual y visualice el código fuente para depurarlo.

Por ejemplo, para ver el código fuente de pkg1/test14.class, especifique lo siguiente:

```
Opc    Programa/módulo     Biblioteca     Tipo
1       pkg1.test14_       *LIBL          *CLASS
```

Si la clase no se ha cargado en la JVM, realice los mismo pasos para añadir la *CLASS como se ha indicado anteriormente. A continuación, se visualizará el mensaje **Archivo de clase Java no disponible**. En este punto, puede reanudar el proceso del programa. La JVM se detiene automáticamente cuando se especifica cualquier método de la clase que coincide con el nombre dado. El código fuente de la clase se visualiza y puede depurarse.

Depurar servlets

La depuración de servlets es un caso especial de las clases de depuración cargadas mediante un cargador de clases personalizadas. Los servlets se ejecutan en la ejecución Java de IBM HTTP Server. Tiene varias opciones para depurar servlets.

La manera más fácil de depurar programas y servlets Java que se ejecuten en el servidor iSeries es utilizar el IBM iSeries System Debugger. El IBM iSeries System Debugger proporciona una interfaz gráfica de usuario que le permite utilizar con mayor facilidad las posibilidades de depuración del servidor iSeries.

Para obtener más información sobre cómo utilizar el iSeries System Debugger para depurar y probar programas y servlets Java que se ejecuten en el servidor iSeries, consulte el apartado IBM iSeries System Debugger.

Otra forma de depurar servlets es seguir las instrucciones para clases cargadas mediante un cargador de clases personalizadas.

También puede utilizar la pantalla interactiva del servidor para depurar un servlet completando los pasos siguientes:

1. Utilice el mandato `javac -g` del intérprete de Qshell para compilar el servlet.
2. Copie el código fuente (archivo .java) y el código compilado (archivo .class) en /QIBM/ProdData/Java400.
3. Ejecute el mandato Crear programa Java (CRTJVAPGM) en el archivo .class utilizando el nivel de optimización 10, OPTIMIZE(10).
4. Inicie el servidor.
5. Ejecute el mandato Arrancar trabajo de servicio (STRSRVJOB) en el trabajo donde se ejecuta el servlet.
6. Entre STRDBG CLASS(myServlet), siendo myServlet el nombre del servlet. Debe visualizarse el fuente.
7. Establezca un punto de interrupción en el servlet y pulse F12.
8. Ejecute el servlet. Cuando el servlet alcance el punto de interrupción, puede continuar depurando.

Arquitectura del depurador de la plataforma Java

La arquitectura del depurador de la plataforma Java (JPDA) consta de tres componentes:

- “Interfaz de depuración de la máquina virtual Java” en la página 268
- “Java Debug Wire Protocol” en la página 268
- “Interfaz de depuración Java” en la página 269

Los tres componentes de JPDA permiten a cualquier componente frontal de un depurador que utilice JDWP realizar operaciones de depuración. El componente frontal del depurador puede ejecutarse remotamente o como aplicación iSeries.

Para obtener más información sobre las características de depuración que están disponibles, consulte el apartado Depuración a máxima velocidad.

Interfaz de depuración de la máquina virtual Java: En Java 2 SDK (J2SDK), Standard Edition, versión 1.2 o superior, la interfaz de depuración de la máquina virtual Java (JVMDI) forma parte de las interfaces de programas de aplicación (API) de la plataforma Sun Microsystems, Inc. JVMDI permite a cualquier usuario escribir un depurador Java para un servidor iSeries en código C de iSeries. El depurador no necesita conocer la estructura interna de la máquina virtual Java, dado que utiliza interfaces JVMDI. JVMDI es la interfaz de nivel más bajo de JPDA, que se encuentra más cerca de la máquina virtual Java.

El depurador se ejecuta en el mismo trabajo con capacidad multihebra que la máquina virtual Java. El depurador utiliza las API de invocación de la interfaz Java nativa (JNI) para crear una máquina virtual Java. A continuación, coloca un gancho al principio de un método main de clase de usuario y llama al método main. Cuando el método main empieza, se encuentra con el gancho y se inicia la depuración. Están disponibles recursos habituales de depuración como los de establecer puntos de interrupción, emitir mandatos de recorrer, visualizar variables y cambiar variables.

El depurador maneja la comunicación entre el trabajo en el que se ejecuta la máquina virtual Java y un trabajo que maneja la interfaz de usuario. Esta interfaz de usuario está en el servidor iSeries o en otro sistema.

Hay un programa de servicio denominado QJVAJVMDI, que reside en la biblioteca QSYS y da soporte a las funciones de JVMDI.

Java Debug Wire Protocol: JDWP (Java Debug Wire Protocol) es un protocolo de comunicaciones predefinido entre un proceso de depurador y JVMDI. JDWP puede utilizarse desde un sistema remoto o a través de un socket local. Se encuentra a una capa de distancia de JVMDI, pero es una interfaz más compleja.

Iniciar JDWP en QShell: Para iniciar JDWP y ejecutar la clase Java SomeClass, especifique el siguiente mandato en QShell:

```
    java -interpret -Xrunjdwp:transport=dt_socket,
address=8000,server=y,suspend=n SomeClass
```

En este ejemplo, JDWP está a la escucha de las conexiones de depuradores remotos en el puerto TCP/IP 8000, pero puede utilizar cualquier número de puerto que desee; dt_socket es el nombre del SRVPGM que maneja el transporte de JDWP, y no cambia.

Para conocer las opciones adicionales que puede utilizar con -Xrunjdwp, consulte Sun VM Invocation Options

de Sun Microsystems, Inc.

Iniciar JDWP desde una línea de mandatos CL: Para utilizar la opción -Xrun con el mandato CL, puede definirse la propiedad os400.xrun.option de forma que sea la misma serie que se utilizaría en la línea de mandatos de QShell. Para iniciar JDWP y ejecutar la clase Java SomeClass, especifique el siguiente mandato:

```
JAVA CLASS(SomeClass) INTERPRET(*YES)
PROP((os400.xrun.option 'jdwp:transport=dt_socket,address=8000,
                                        server=y,suspend=n'))
```

No se recomienda utilizar JVMDI para el código de Ejecución directa. Deberá ejecutar la aplicación con el intérprete, o utilizar el compilador Just-In-Time (JIT) con la depuración a máxima velocidad.

Interfaz de depuración Java: JDI (interfaz de depuración Java) es una interfaz de alto nivel del lenguaje Java suministrada para el desarrollo de herramientas. JDI oculta la complejidad de JVMDI y JDWP detrás de algunas definiciones de clase Java. JDI está incluida en el archivo rt.jar y, por tanto, el componente frontal del depurador existe en cualquier plataforma que tenga instalado Java.

Si desea escribir depuradores para Java debe utilizar JDI, ya que es la interfaz más sencilla y el código es independiente de plataforma.

Para obtener más información acerca de JDPA, consulte Java Platform Debugger Architecture Overview

de Sun Microsystems, Inc.

Depuración a máxima velocidad: La máquina virtual Java (JVM) de iSeries ahora da soporte a la ″depuración a máxima velocidad″. Antes de la v5r3, habilitar la depuración significaba tener que inhabilitar el compilador Just-In-Time (JIT). El rendimiento de las aplicaciones sufría debido a que debía ejecutarse muchos métodos con el intérprete lento. Esta importante degradación del rendimiento era especialmente difícil para las aplicaciones que podían estar ejecutándose durante varios días antes de llegar el punto en que el usuario deseaba empezar la depuración.

La depuración a máxima velocidad le permite ejecutar la aplicación con todas las ventajas de rendimiento del código compilado de JIT, sin perder la capacidad de realizar algunas de las actividades de depuración más comunes como, por ejemplo, establecer puntos de interrupción, recorrer el código y visualizar variables locales.

Dado que la depuración a máxima velocidad permite que los métodos se compilen con JIT, existen un par de limitaciones para la depuración:

- Las operaciones de recorrer en las sentencias de retorno no funcionan si el llamante es código compilado.
- Los puntos de observación solamente se activan en los métodos no compilados que modifican el campo observado.

Nota: Esta característica solamente está soportada para los depuradores que utilizan el protocolo JDWP (Java Debug Wire Protocol) para realizar operaciones de depuración. Actualmente, el depurador del sistema no da soporte a la depuración a máxima velocidad.

Localizar fugas de memoria

Si el rendimiento del programa se degrada al ejecutarse durante mucho más tiempo, es posible que haya codificado una fuga de memoria erróneamente. Puede utilizar Java Watcher como ayuda para depurar el programa y localizar las fugas de memoria realizando el análisis del almacenamiento dinámico de aplicaciones Java y el perfilado de creación de objeto.

Para obtener más detalles, consulte JavaWatcher.

También puede utilizar el mandato de lenguaje de control Analizar máquina virtual Java (ANZJVM) para buscar fugas de objetos. ANZJVM busca fugas de objeto tomando dos copias del almacenamiento dinámico de recogida de basura separadas por un intervalo de tiempo especificado. Para buscar fugas de objeto, observará el número de instancias de cada clase que figuran en el almacenamiento dinámico. Las clases que tienen un número de instancias inusitadamente alto deben considerarse como posibles fugas.

También debe observar el cambio en el número de instancias de cada clase entre las dos copias del almacenamiento dinámico de recogida de basura. Si el número de instancias de una clase aumenta continuamente, dicha clase debe considerarse como posible fuga. Cuanto mayor sea el intervalo de tiempo entre las dos copias, más posibilidades hay de que realmente existan fugas en los objetos. Ejecutando ANZJVM una serie de veces con un intervalo de tiempo mayor, debe ser capaz de diagnosticar las fugas con un mayor grado de certeza.

Ejemplos de código para IBM Developer Kit para Java

A continuación se ofrece una lista de ejemplos de código para IBM Developer Kit para Java.

Declaración de limitación de responsabilidad sobre el código de ejemplo

IBM otorga al usuario una licencia de copyright no exclusiva para utilizar todos los ejemplos de código de programación, a partir de los que puede generar funciones similares adaptadas a sus necesidades específicas.

IBM proporciona la totalidad del código de ejemplo sólo con propósito ilustrativo. Estos ejemplos no se han probado exhaustivamente bajo todas las condiciones. Por tanto, IBM no puede garantizar la fiabilidad, capacidad de servicio o funcionamiento de estos programas.

Todos los programas que contiene esta documentación se suministran TAL CUAL, sin garantías de ninguna clase. Se renuncia explícitamente a las garantías implícitas de no infringibilidad, comercialización y adecuación a un propósito determinado.

Internacionalización

- DateFormat
- NumberFormat
- ResourceBundle

JDBC

- Propiedad Access
- Blob
- Interfaz CallableStatement
- Cambiar valores con una sentencia mediante el cursor de otra sentencia
- Clob
- Crear un UDBDataSource y enlazarlo con JNDI
- Crear UDBDataSource y obtener un ID de usuario y una contraseña
- Crear un UDBDataSourceBind y establecer propiedades de DataSource
- Interfaz DatabaseMetaData

- Crear un UDBDataSource y enlazarlo con JNDI
- Enlace de datos (Datalink)
- Tipos distintivos
- Intercalar sentencias SQL
- Finalizar una transacción
- ID de usuario y contraseña no válidos
- JDBC
- Varias conexiones que funcionan en una transacción
- Obtener un contexto inicial antes de enlazar UDBDataSource
- ParameterMetaData
- Eliminar valores de una tabla mediante el cursor de otra sentencia
- Interfaz ResultSet
- Sensibilidad de ResultSet
- ResultSets sensibles e insensibles
- Configurar una agrupación de conexiones con UDBDataSource y UDBConnectionPoolDataSource
- SQLException
- Suspender y reanudar una transacción
- ResultSets suspendidos
- Probar el rendimiento de una agrupación de conexiones
- Probar el rendimiento de dos DataSources
- Actualizar BLOB
- Actualizar CLOB
- Utilizar una conexión con varias transacciones
- Utilizar BLOB
- Utilizar CLOB
- "Utilizar propiedades DB2CachedRowSet y DataSources" en la página 109
- "Utilizar propiedades DB2CachedRowSet y los URL de JDBC" en la página 110
- Utilizar JTA para manejar una transacción
- Utilizar ResultSets de metadatos que tienen más de una columna
- Utilizar JDBC nativo y JDBC de IBM Toolbox para Java de forma concurrente
- Utilizar PreparedStatement para obtener un ResultSet
- "Utilizar el método execute(Connection) para utilizar una conexión de base de datos existente" en la página 111
- "Utilizar el método execute(int) para agrupar peticiones de base de datos" en la página 111
- "Utilizar el método populate" en la página 109
- "Utilizar el método setConnection(Connection) para utilizar una conexión de base de datos existente" en la página 110
- Utilizar el método executeUpdate de un objeto Statement

Servicio de autenticación y autorización Java
- Ejemplo HelloWorld de JAAS
- Ejemplo SampleThreadSubjectLogin de JAAS

Java Generic Security Service
- Programa cliente no JAAS de ejemplo
- Programa servidor no JAAS de ejemplo

- Programa cliente habilitado para JAAS de ejemplo
- Programa servidor habilitado para JAAS de ejemplo

Extensión de sockets seguros Java
- Cliente y servidor SSL que utilizan un objeto SSLContext

Java con otros lenguajes de programación
- Llamar a un programa CL
- Llamar a un mandato CL
- Llamar a otro programa Java
- Llamar a Java desde C
- Llamar a Java desde RPG
- Corrientes de entrada y de salida
- La API de invocación
- Método nativo OS/400 PASE para Java
- Sockets
- Utilizar la interfaz nativa Java para métodos nativos

Herramientas de rendimiento
- Java Performance Data Converter

SQLJ
- Intercalar sentencias SQL en la aplicación Java

SSL
- Fábricas de sockets
- Fábricas de sockets de servidor
- SSL
- Servidor de capa de sockets segura

Ejemplo: internacionalización de las fechas con la clase java.util.DateFormat

Este ejemplo muestra cómo pueden utilizarse los entornos nacionales para formatear las fechas.

Ejemplo 1: enseña a utilizar la clase java.util.DateFormat para internacionalizar las fechas

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
//************************
// Archivo: DateExample.java
//************************

import java.text.*;
import java.util.*;
import java.util.Date;

public class DateExample {

   public static void main(String args[]) {

     // Se obtiene la fecha
     Date now = new Date();
```

```
    // Se obtienen los formateadores de fecha de los entornos nacionales por
    // omisión, alemán y francés
    DateFormat theDate = DateFormat.getDateInstance(DateFormat.LONG);
    DateFormat germanDate = DateFormat.getDateInstance(DateFormat.LONG, Locale.GERMANY);
    DateFormat frenchDate = DateFormat.getDateInstance(DateFormat.LONG, Locale.FRANCE);

    // Se formatean y se imprimen las fechas
    System.out.println("Fecha en el entorno nacional por omisión: " + theDate.format(now));
    System.out.println("Fecha en el entorno nacional alemán: " + germanDate.format(now));
    System.out.println("Fecha en el entorno nacional francés: " + frenchDate.format(now));
  }
}
```

Para obtener más información, consulte Crear un programa Java internacionalizado.

Ejemplo: internacionalización de las presentaciones numéricas con la clase java.util.NumberFormat

Este ejemplo muestra cómo pueden utilizarse los entornos nacionales para formatear los números.

Ejemplo 1: enseña a utilizar la clase java.util.NumberFormat para internacionalizar la salida numérica.

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
//**************************
// Archivo: NumberExample.java
//**************************

 import java.lang.*;
 import java.text.*;
 import java.util.*;

 public class NumberExample {

     public static void main(String args[]) throws NumberFormatException {

       // El número que debe formatearse
       double number = 12345.678;

       // Se obtienen los formateadores de los entornos nacionales
       // por omisión, español y japonés
       NumberFormat defaultFormat = NumberFormat.getInstance();
       NumberFormat spanishFormat = NumberFormat.getInstance(new
 Locale("es", "ES"));
       NumberFormat japaneseFormat = NumberFormat.getInstance(Locale.JAPAN);

       // Se imprimen los números con el formato por omisión, español y japonés
       // (Nota: NumberFormat no es necesario para el formato por omisión)
       System.out.println("El número formateado para el entorno nacional por omisión; " +
                          defaultFormat.format(number));
       System.out.println("El número formateado para el entorno nacional español; " +
                          spanishFormat.format(number));
       System.out.println("El número formateado para el entorno nacional japonés; " +
                          japaneseFormat.format(number));
     }
 }
```

Para obtener más información, consulte Crear un programa Java internacionalizado.

Ejemplo: internacionalización de los datos específicos de entorno nacional con la clase java.util.ResourceBundle

Este ejemplo muestra cómo pueden utilizarse los entornos nacionales junto con paquetes de recursos para internacionalizar las series del programa.

Para que el programa ResourceBundleExample funcione como se pretende, se necesitan los archivos de propiedades siguientes:

Contenido de RBExample.properties
Hello.text=Hello

Contenido de RBExample_de.properties
Hello.text=Guten Tag

Contenido de RBExample_fr_FR.properties
Hello.text=Bonjour

Ejemplo 1: enseña a utilizar la clase java.util.ResourceBundle para internacionalizar los datos específicos de entorno nacional

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
//*********************************
// Archivo: ResourceBundleExample.java
//*********************************

import java.util.*;

public class ResourceBundleExample {
   public static void main(String args[]) throws MissingResourceException {

     String resourceName = "RBExample";
     ResourceBundle rb;

     // Entorno nacional por omisión
     rb = ResourceBundle.getBundle(resourceName);
     System.out.println("Por omisión: " + rb.getString("Hello" + ".text"));

     // Se solicita un paquete de recursos con un entorno nacional
     // especificado de manera explícita
     rb = ResourceBundle.getBundle(resourceName, Locale.GERMANY);
     System.out.println("Alemán: " + rb.getString("Hello" + ".text"));

     // No existe ningún archivo de propiedades para China en este
     // ejemplo... se utiliza el valor por omisión
     rb = ResourceBundle.getBundle(resourceName, Locale.CHINA);
     System.out.println("Chino: " + rb.getString("Hello" + ".text"));

     // He aquí otra manera de hacerlo...
     Locale.setDefault(Locale.FRANCE);
     rb = ResourceBundle.getBundle(resourceName);
     System.out.println("Francés: " + rb.getString("Hello" + ".text"));

     // No existe ningún archivo de propiedades para China en este
     // ejemplo... se utiliza el valor por omisión, que ahora es
     // fr_FR.
     rb = ResourceBundle.getBundle(resourceName, Locale.CHINA);
     System.out.println("Chino: " + rb.getString("Hello" + ".text"));
   }
}
```

Para obtener más información, consulte Crear un programa Java internacionalizado.

Ejemplo: propiedad Access

Este es un ejemplo de cómo utilizar la propiedad Access.

Ejemplo: propiedad Access

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
// Nota: En este programa se presupone la existencia del directorio cujosql.
import java.sql.*;
import javax.sql.*;
import javax.naming.*;

public class AccessPropertyTest {
    public String url = "jdbc:db2:*local";
    public Connection connection = null;

    public static void main(java.lang.String[] args)
    throws Exception
    {
        AccessPropertyTest test = new AccessPropertyTest();

        test.setup();

        test.run();
        test.cleanup();
    }


/**
Configurar el DataSource utilizado en la prueba.
**/
    public void setup()
    throws Exception
    {
   Class.forName("com.ibm.db2.jdbc.app.DB2Driver");

        connection = DriverManager.getConnection(url);
        Statement s = connection.createStatement();
           try {
            s.executeUpdate("DROP TABLE CUJOSQL.TEMP");
        } catch (SQLException e) { // Ignorarlo - no existe
        }

           try {
            String sql = "CREATE PROCEDURE CUJOSQL.TEMP "
                  + " LANGUAGE SQL SPECIFIC CUJOSQL.TEMP "
                  + " MYPROC: BEGIN"
                  + "   RETURN 11;"
                  + " END MYPROC";
                s.executeUpdate(sql);
            } catch (SQLException e) {
            // Ignorarlo - existe.
        }
        s.executeUpdate("create table cujosql.temp (col1 char(10))");
        s.executeUpdate("insert into cujosql.temp values ('compare')");
    s.close();
    }


    public void resetConnection(String property)
    throws SQLException
    {
        if (connection != null)
            connection.close();
```

```
    connection = DriverManager.getConnection(url + ";access=" + property);
}


public boolean canQuery() {
    Statement s = null;
       try {
        s = connection.createStatement();
        ResultSet rs = s.executeQuery("SELECT * FROM cujosql.temp");
        if (rs == null)
        return false;

        rs.next();

        if (rs.getString(1).equals("compare   "))
        return true;

        return false;

        } catch (SQLException e) {
        // System.out.println("Excepción: SQLState(" +
        //                    e.getSQLState() + ") " + e + " (" + e.getErrorCode() + ")");
        return false;
    } finally {
          if (s != null) {
       try {
s.close();
       } catch (Exception e) {
                // Ignorarlo.
            }
        }
    }
}


public boolean canUpdate() {
    Statement s = null;
       try {
        s = connection.createStatement();
        int count = s.executeUpdate("INSERT INTO CUJOSQL.TEMP VALUES('x')");
        if (count != 1)
        return false;

        return true;

        } catch (SQLException e) {
        //System.out.println("Excepción: SQLState(" +
        //                   e.getSQLState() + ") " + e + " (" + e.getErrorCode() + ")");
        return false;
    } finally {
          if (s != null) {
       try {
s.close();
       } catch (Exception e) {
                // Ignorarlo.
            }
        }
    }
}


public boolean canCall() {
    CallableStatement s = null;
       try {
        s = connection.prepareCall("? = CALL CUJOSQL.TEMP()");
        s.registerOutParameter(1, Types.INTEGER);
        s.execute();
```

```
            if (s.getInt(1) != 11)
            return false;

            return true;

            } catch (SQLException e) {
            //System.out.println("Excepción: SQLState(" +
            //                   e.getSQLState() + ") " + e + " (" + e.getErrorCode() + ")");
            return false;
        } finally {
                if (s != null) {
           try {
    s.close();
           } catch (Exception e) {
                    // Ignorarlo.
                }
            }
        }
    }


    public void run()
    throws SQLException
    {
        System.out.println("Set the connection access property to read only");
        resetConnection("read only");

        System.out.println("Can run queries  -->" + canQuery());
        System.out.println("Can run updates  -->" + canUpdate());
        System.out.println("Can run sp calls -->" + canCall());

        System.out.println("Set the connection access property to read call");
        resetConnection("read call");

        System.out.println("Can run queries  -->" + canQuery());
        System.out.println("Can run updates  -->" + canUpdate());
        System.out.println("Can run sp calls -->" + canCall());

        System.out.println("Set the connection access property to all");
        resetConnection("all");

        System.out.println("Can run queries  -->" + canQuery());
        System.out.println("Can run updates  -->" + canUpdate());
        System.out.println("Can run sp calls -->" + canCall());

    }


    public void cleanup() {
           try {
            connection.close();
           } catch (Exception e) {
                    // Ignorarlo.
        }
    }
}
```

Ejemplo: BLOB

Este es un ejemplo de cómo puede colocarse un BLOB en la base de datos o recuperarse de la misma.

Ejemplo: BLOB

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
//////////////////////////////////////
// PutGetBlobs es una aplicación de ejemplo
// que muestra cómo trabajar con la API de JDBC
// para colocar objetos BLOB en columnas de base
// de datos y obtenerlos desde ellas.
//
// Los resultados de la ejecución de este programa
// provocan la inserción de dos valores de BLOB
// en una tabla nueva. Ambos son idénticos
// y contienen 500k de datos de byte
// aleatorios.
//////////////////////////////////////
import java.sql.*;
import java.util.Random;

public class PutGetBlobs {
   public static void main(String[] args)
   throws SQLException
   {
       // Registrar el controlador JDBC nativo.
           try {
   Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
                   } catch (Exception e) {
          System.exit(1);  // Error de configuración.
      }

      // Establecer una conexión y una sentencia con las que trabajar.
      Connection c = DriverManager.getConnection("jdbc:db2:*local");
   Statement s = c.createStatement();

      // Borrar ejecuciones anteriores de esta aplicación.
           try {
          s.executeUpdate("DROP TABLE CUJOSQL.BLOBTABLE");
           } catch (SQLException e) {
          // Ignorarlo y suponer que la tabla no existía.
      }

      // Crear una tabla con una columna BLOB. El tamaño de columna BLOB
      // por omisión es de 1 MB.
      s.executeUpdate("CREATE TABLE CUJOSQL.BLOBTABLE (COL1 BLOB)");

      // Crear un objeto PreparedStatement que permita colocar
      // un nuevo objeto Blob en la base de datos.
      PreparedStatement ps = c.prepareStatement("INSERT INTO CUJOSQL.BLOBTABLE VALUES(?)");

      // Crear un valor BLOB grande...
      Random random = new Random ();
      byte [] inByteArray = new byte[500000];
      random.nextBytes (inByteArray);

      // Establecer el parámetro PreparedStatement. Nota: no es portable
      // a todos los controladores JDBC. Los controladores JDBC no
      // tienen soporte al utilizar setBytes para columnas BLOB. Se
      // utiliza para permitir la generación de nuevos BLOB. También permite
      // a los controladores JDBC 1.0 trabajar con columnas que contienen datos BLOB.
      ps.setBytes(1, inByteArray);

      // Procesar la sentencia, insertando el BLOB en la base de datos.
   ps.executeUpdate();

      // Procesar una consulta y obtener el BLOB que se acaba de insertar
      // de la base de datos como objeto Blob.
      ResultSet rs = s.executeQuery("SELECT * FROM CUJOSQL.BLOBTABLE");
      rs.next();
      Blob blob = rs.getBlob(1);
```

```
      // Colocar de nuevo ese Blob en la base de datos mediante
      // la PreparedStatement.
      ps.setBlob(1, blob);
      ps.execute();

      c.close(); // El cierre de la conexión también cierra stmt y rs.
   }
}
```

Ejemplo: interfaz CallableStatement para IBM Developer Kit para Java

Este es un ejemplo de cómo utilizar la interfaz CallableStatement.

Ejemplo: interfaz CallableStatement

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
// Conectarse al servidor iSeries.
Connection c = DriverManager.getConnection("jdbc:db2://mySystem");

// Se crea el objeto CallableStatement.
// Este precompila la llamada especificada a un procedimiento almacenado.
// Los signos de interrogación señalan el lugar en que deben establecerse
// los parámetros de entrada y el lugar en que deben recuperarse los
// parámetros de salida.
// Los dos primeros parámetros son de entrada y el tercero es de salida.
CallableStatement cs = c.prepareCall("CALL MYLIBRARY.ADD (?, ?, ?)");

// Se establecen los parámetros de entrada.
cs.setInt (1, 123);
cs.setInt (2, 234);

// Se registra el tipo del parámetro de salida.
cs.registerOutParameter (3, Types.INTEGER);

// Se ejecuta el procedimiento almacenado.
cs.execute ();

// Se obtiene el valor del parámetro de salida.
int sum = cs.getInt (3);

// Se cierra CallableStatement y la conexión.
cs.close();
c.close();
```

Para obtener más información, consulte la sección CallableStatements.

Ejemplo: eliminar valores de una tabla mediante el cursor de otra sentencia

Este es un ejemplo de cómo eliminar valores de una tabla mediante el cursor de otra sentencia.

Ejemplo: eliminar valores de una tabla mediante el cursor de otra sentencia

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
import java.sql.*;

public class UsingPositionedDelete {
    public Connection connection = null;
    public static void main(java.lang.String[] args) {
        UsingPositionedDelete test = new UsingPositionedDelete();
```

```
        test.setup();
        test.displayTable();

        test.run();
        test.displayTable();

        test.cleanup();
    }



/**
Manejar todo el trabajo de configuración necesario.
**/
   public void setup() {
            try {
            // Registrar el controlador JDBC.
   Class.forName("com.ibm.db2.jdbc.app.DB2Driver");

            connection = DriverManager.getConnection("jdbc:db2:*local");

            Statement s = connection.createStatement();
           try {
                s.executeUpdate("DROP TABLE CUJOSQL.WHERECUREX");
            } catch (SQLException e) {
                // Aquí, pasar por alto los problemas.
            }

            s.executeUpdate("CREATE TABLE CUJOSQL.WHERECUREX ( " +
                            "COL_IND INT, COL_VALUE CHAR(20)) ");

            for (int i = 1; i <= 10; i++) {
                s.executeUpdate("INSERT INTO CUJOSQL.WHERECUREX VALUES(" + i + ", 'FIRST')");
            }

   s.close();

                    } catch (Exception e) {
            System.out.println("Excepción capturada: " + e.getMessage());
         e.printStackTrace();
        }
    }


/**
En esta sección, debe añadirse todo el código destinado a realizar
la prueba. Si sólo es necesaria una conexión con la base de datos,
puede utilizarse la variable global 'connection'.
**/
   public void run() {
            try {
            Statement stmt1 = connection.createStatement();

            // Actualizar cada valor utilizando next().
            stmt1.setCursorName("CUJO");
            ResultSet rs = stmt1.executeQuery ("SELECT * FROM CUJOSQL.WHERECUREX " +
                                               "FOR UPDATE OF COL_VALUE");

            System.out.println("El nombre de cursor es " + rs.getCursorName());

            PreparedStatement stmt2 = connection.prepareStatement
                                      ("DELETE FROM " + " CUJOSQL.WHERECUREX WHERE CURRENT OF " +
                                      rs.getCursorName ());


            // Explorar en bucle el ResultSet y actualizar todas las demás entradas.
            while (rs.next()) {
```

```
                if (rs.next())
                    stmt2.execute ();
            }


            // Borrar los recursos una vez utilizados.
            rs.close ();
            stmt2.close ();


                    } catch (Exception e) {
            System.out.println("Excepción capturada: ");
          e.printStackTrace();
         }
    }



/**
En esta sección, colocar todo el trabajo de borrado para la prueba.
**/
    public void cleanup() {
            try {
            // Cerrar la conexión global abierta en setup().
            connection.close();

                    } catch (Exception e) {
            System.out.println("Excepción capturada: ");
          e.printStackTrace();
         }
    }



/**
Visualizar el contenido de la tabla.
**/
    public void displayTable()
    {
            try {
            Statement s = connection.createStatement();
            ResultSet rs = s.executeQuery ("SELECT * FROM CUJOSQL.WHERECUREX");

            while (rs.next()) {
                System.out.println("Index " + rs.getInt(1) + " valor " + rs.getString(2));
            }

            rs.close ();
   s.close();
            System.out.println("----------------------------------------");
                    } catch (Exception e) {
            System.out.println("Excepción capturada: ");
          e.printStackTrace();
         }
    }
}
```

Ejemplo: CLOB

Este es un ejemplo de cómo puede colocarse un CLOB en la base de datos o recuperarse de la misma.

Ejemplo: CLOB

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
//////////////////////////////////////
// PutGetClobs es una aplicación de ejemplo
// que muestra cómo trabajar con la API de JDBC
// para colocar objetos CLOB en columnas de base
// de datos y obtenerlos desde ellas.
//
// Los resultados de la ejecución de este programa
// provocan la inserción de dos valores de CLOB
// en una tabla nueva. Ambos son idénticos
// y contienen alrededor de 500k de datos
// de texto de repetición.
//////////////////////////////////////
import java.sql.*;

public class PutGetClobs {
   public static void main(String[] args)
   throws SQLException
   {
       // Registrar el controlador JDBC nativo.
            try {
   Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
                   } catch (Exception e) {
          System.exit(1);  // Error de configuración.
      }

      // Establecer una conexión y una sentencia con las que trabajar.
      Connection c = DriverManager.getConnection("jdbc:db2:*local");
   Statement s = c.createStatement();

      // Borrar ejecuciones anteriores de esta aplicación.
            try {
          s.executeUpdate("DROP TABLE CUJOSQL.CLOBTABLE");
            } catch (SQLException e) {
          // Ignorarlo y suponer que la tabla no existía.
      }

      // Crear una tabla con una columna CLOB. El tamaño de columna CLOB
      // por omisión es de 1 MB.
      s.executeUpdate("CREATE TABLE CUJOSQL.CLOBTABLE (COL1 CLOB)");

      // Crear un objeto PreparedStatement que permita colocar
      // un nuevo objeto Clob en la base de datos.
      PreparedStatement ps = c.prepareStatement("INSERT INTO CUJOSQL.CLOBTABLE VALUES(?)");

      // Crear un valor CLOB grande...
      StringBuffer buffer = new StringBuffer(500000);
      while (buffer.length() < 500000) {
          buffer.append("All work and no play makes Cujo a dull boy.");
      }
      String clobValue = buffer.toString();

      // Establecer el parámetro PreparedStatement.  No es portable
      // a todos los controladores JDBC. Los controladores JDBC no
      // tienen soporte para setBytes para columnas CLOB. Se realiza para
      //  permitir la generación de nuevos CLOB. También permite
      // a los controladores JDBC 1.0 trabajar con columnas que contienen
      // datos Clob.
      ps.setString(1, clobValue);

      // Procesar la sentencia, insertando el clob en la base de datos.
   ps.executeUpdate();

      // Procesar una consulta y obtener el CLOB que se acaba de insertar
      // de la base de datos como objeto Clob.
      ResultSet rs = s.executeQuery("SELECT * FROM CUJOSQL.CLOBTABLE");
      rs.next();
Clob clob = rs.getClob (1);
```

```
      // Colocar de nuevo ese Clob en la base de datos mediante
      // la PreparedStatement.
      ps.setClob(1, clob);
      ps.execute();

      c.close(); // El cierre de la conexión también cierra stmt y rs.
   }
}
```

Ejemplo: crear un UDBDataSource y enlazarlo con JNDI

Este es un ejemplo de cómo crear un UDBDataSource y enlazarlo con JNDI.

Ejemplo: crear un UDBDataSource y enlazarlo con JNDI

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
// Importar los paquetes necesarios. En el momento del despliegue,
// debe importarse la clase específica del controlador JDBC
// que implementa DataSource.
import java.sql.*;
import javax.naming.*;
import com.ibm.db2.jdbc.app.UDBDataSource;

public class UDBDataSourceBind
{
    public static void main(java.lang.String[] args)
    throws Exception
    {
        // Crear un nuevo objeto UDBDataSource y proporcionarle
        // una descripción.
        UDBDataSource ds = new UDBDataSource();
        ds.setDescription("Un UDBDataSource simple");

        // Recuperar un contexto JNDI. El contexto funciona
        // como raíz donde se enlazan o se encuentran
        // los objetos en JNDI.
        Context ctx = new InitialContext();

        // Enlazar el objeto UDBDataSource recién creado
        // con el servicio de directorios JNDI, dándole un nombre
        // que pueda utilizarse para buscar de nuevo este objeto
        // con posterioridad.
        ctx.rebind("SimpleDS", ds);
    }
}
```

Ejemplo: crear UDBDataSource y obtener un ID de usuario y una contraseña

Este es un ejemplo de cómo crear un UDBDataSource y utilizar el método getConnection para obtener un ID de usuario y una contraseña durante la ejecución.

Ejemplo: crear un UDBDataSource y obtener un ID de usuario y una contraseña

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
// Importar los paquetes necesarios. No es necesario ningún
// código específico de controlador en aplicaciones
// de ejecución.
import java.sql.*;
```

```
import javax.sql.*;
import javax.naming.*;

public class UDBDataSourceUse2
{
    public static void main(java.lang.String[] args)
    throws Exception
    {
        // Recuperar un contexto JNDI. El contexto funciona
        // como raíz donde se enlazan o se encuentran
        // los objetos en JNDI.
        Context ctx = new InitialContext();

        // Recuperar el objeto UDBDataSource enlazado mediante el
        // nombre con el que estaba enlazado anteriormente. Durante
        // la ejecución, sólo se utiliza la interfaz DataSource,
        // y por tanto no es necesario convertir el objeto a la clase
        // de implementación de UDBdataSource. (No es necesario saber cuál
        // es la clase de implementación. Sólo es necesario el nombre
        // lógico de JNDI).
        DataSource ds = (DataSource) ctx.lookup("SimpleDS");


        // Una vez obtenido el DataSource, puede utilizarse para establecer
        // una conexión. El perfil de usuario cujo y la contraseña newtiger
        // se utilizan para crear la conexión en lugar del ID de usuario
        // y la contraseña por omisión para el DataSource.
        Connection connection = ds.getConnection("cujo", "newtiger");

        // La conexión puede utilizarse para crear objetos Statement y
        // actualizar la base de datos o procesar consultas de la forma siguiente.
        Statement statement = connection.createStatement();
        ResultSet rs = statement.executeQuery("select * from qsys2.sysprocs");
   while (rs.next()) {
            System.out.println(rs.getString(1) + "." + rs.getString(2));
        }

        // La conexión se cierra antes de que finalice la aplicación.
        connection.close();
    }
}
```

Ejemplo: crear un UDBDataSourceBind y establecer las propiedades de DataSource

Este es un ejemplo de cómo crear un UDBDataSource y establecer el ID de usuario y la contraseña como propiedades de DataSource.

Ejemplo: crear un UDBDataSourceBind y establecer propiedades de DataSource

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
// Importar los paquetes necesarios. En el momento del despliegue,
// debe importarse la clase específica del controlador JDBC
// que implementa DataSource.
import java.sql.*;
import javax.naming.*;
import com.ibm.db2.jdbc.app.UDBDataSource;

public class UDBDataSourceBind2
{
    public static void main(java.lang.String[] args)
    throws Exception
    {
        // Crear un nuevo objeto UDBDataSource y proporcionarle
```

```
        // una descripción.
        UDBDataSource ds = new UDBDataSource();
        ds.setDescription("Un UDBDataSource simple"  +
                          "con cujo como perfil por" +
                          "omisión al que conectarse.");

        // Proporcionar un ID de usuario y una contraseña que
        // deben utilizarse para las propiedades de conexión.
        ds.setUser("cujo");
        ds.setPassword("newtiger");

        // Recuperar un contexto JNDI. El contexto funciona
        // como raíz donde se enlazan o se encuentran
        // los objetos en JNDI.
        Context ctx = new InitialContext();

        // Enlazar el objeto UDBDataSource recién creado
        // con el servicio de directorios JNDI, dándole un nombre
        // que pueda utilizarse para buscar de nuevo este objeto
        // con posterioridad.
        ctx.rebind("SimpleDS2", ds);
    }
}
```

Ejemplo: interfaz DatabaseMetaData para IBM Developer Kit para Java

Este ejemplo muestra cómo devolver una lista de tablas.

Ejemplo 1: devolver una lista de tablas.

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
// Conectarse al servidor iSeries.
Connection c = DriverManager.getConnection("jdbc:db2:mySystem");

// Se obtienen los metadatos de base de datos de la conexión.
DatabaseMetaData dbMeta = c.getMetaData();

// Se obtiene una lista de tablas que cumplen estos criterios.
String catalog = "myCatalog";
String schema = "mySchema";
String table  = "myTable%"; // % indica el patrón de búsqueda
String types[] = {"TABLE", "VIEW", "SYSTEM TABLE"}:
ResultSet rs = dbMeta.getTables(catalog, schema, table, types);

// ... se itera en ResultSet para obtener los valores.

   // Se cierra la conexión.
c.close():
```

Para obtener más información, consulte la sección Interfaz DatabaseMetaData para IBM Developer Kit para Java.

Ejemplo: Datalink

Este es un ejemplo de cómo utilizar datalinks en las aplicaciones.

Ejemplo: Datalink

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
//////////////////////////////////////
// PutGetDatalinks es una aplicación de ejemplo
// que muestra cómo utilizar la API de JDBC
// para manejar columnas de base de datos de tipo datalink.
//////////////////////////////////////
import java.sql.*;
import java.net.URL;
import java.net.MalformedURLException;

public class PutGetDatalinks {
   public static void main(String[] args)
    throws SQLException
    {
        // Registrar el controlador JDBC nativo.
            try {
   Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
                    } catch (Exception e) {
            System.exit(1);  // Error de configuración.
        }

      // Establecer una conexión y una sentencia con las que trabajar.
        Connection c = DriverManager.getConnection("jdbc:db2:*local");
   Statement s = c.createStatement();

        // Borrar ejecuciones anteriores de esta aplicación.
            try {
            s.executeUpdate("DROP TABLE CUJOSQL.DLTABLE");
            } catch (SQLException e) {
            // Ignorarlo y suponer que la tabla no existía.
        }

        // Crear una tabla con una columna datalink.
        s.executeUpdate("CREATE TABLE CUJOSQL.DLTABLE (COL1 DATALINK)");

        // Crear un objeto PreparedStatement que permita añadir
        // un nuevo objeto datalink en la base de datos. Dado que la conversión
        // a un datalink no puede realizarse directamente en la base de datos,
        // puede codificar la sentencia SQL para realizar la conversión explícita.
        PreparedStatement ps = c.prepareStatement("INSERT INTO CUJOSQL.DLTABLE
                                                   VALUES(DLVALUE( CAST(? AS VARCHAR(100))))");

        // Establecer el datalink. Este URL le indica un artículo relativo a
        // las nuevas características de JDBC 3.0.
        ps.setString (1, "http://www-106.ibm.com/developerworks/java/library/j-jdbcnew/index.html");

        // Procesar la sentencia, insertando el CLOB en la base de datos.
   ps.executeUpdate();

        // Procesar una consulta y obtener el CLOB que se acaba de insertar
        // de la base de datos como objeto Clob.
        ResultSet rs = s.executeQuery("SELECT * FROM CUJOSQL.DLTABLE");
        rs.next();
        String datalink = rs.getString(1);


        // Colocar ese valor de datalink en la base de datos mediante
        // la PreparedStatement. Nota: esta función requiere el soporte de
        // JDBC 3.0.
        /*
            try {
            URL url = new URL(datalink);
            ps.setURL(1, url);
            ps.execute();
        } catch (MalformedURLException mue) {
            // Manejar aquí este problema.
        }
```

```
        rs = s.executeQuery("SELECT * FROM CUJOSQL.DLTABLE");
        rs.next();
        URL url = rs.getURL(1);
        System.out.println("el valor de URL es " + url);
        */

        c.close(); // El cierre de la conexión también cierra stmt y rs.
    }
}
```

Ejemplo: tipos Distinct

Este es un ejemplo de utilización de tipos distinct.

Ejemplo: tipos Distinct

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
////////////////////////////////////////
// Este programa de ejemplo muestra ejemplos de
// diversas tareas comunes que pueden realizarse
// con tipos distinct.
////////////////////////////////////////
import java.sql.*;

public class Distinct {
   public static void main(String[] args)
   throws SQLException
    {
       // Registrar el controlador JDBC nativo.
           try {
   Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
                   } catch (Exception e) {
           System.exit(1);  // Error de configuración.
        }

        Connection c = DriverManager.getConnection("jdbc:db2:*local");
   Statement s = c.createStatement();

        // Borrar ejecuciones antiguas.
           try {
           s.executeUpdate("DROP TABLE CUJOSQL.SERIALNOS");
           } catch (SQLException e) {
           // Ignorarlo y suponer que la tabla no existía.
        }

           try {
           s.executeUpdate("DROP DISTINCT TYPE CUJOSQL.SSN");
           } catch (SQLException e) {
           // Ignorarlo y suponer que la tabla no existía.
        }

        // Crear el tipo, crear la tabla e insertar un valor.
        s.executeUpdate("CREATE DISTINCT TYPE CUJOSQL.SSN AS CHAR(9)");
        s.executeUpdate("CREATE TABLE CUJOSQL.SERIALNOS (COL1 CUJOSQL.SSN)");

        PreparedStatement ps = c.prepareStatement("INSERT INTO CUJOSQL.SERIALNOS VALUES(?)");
        ps.setString(1, "399924563");
                ps.executeUpdate();
        ps.close();

        // Puede obtener detalles acerca de los tipos disponibles con nuevos
        // metadatos en JDBC 2.0
        DatabaseMetaData dmd = c.getMetaData();

        int types[] = new int[1];
```

```
        types[0] = java.sql.Types.DISTINCT;

        ResultSet rs = dmd.getUDTs(null, "CUJOSQL", "SSN", types);
        rs.next();
        System.out.println("Nombre tipo " + rs.getString(3) +
                           " tiene el tipo " + rs.getString(4));


        // Acceder a los datos que ha insertado.
        rs = s.executeQuery("SELECT COL1 FROM CUJOSQL.SERIALNOS");
        rs.next();
        System.out.println("SSN es " + rs.getString(1));

        c.close(); // El cierre de la conexión también cierra stmt y rs.
    }
}
```

Ejemplo: intercalar sentencias SQL en la aplicación Java

La siguiente aplicación SQLJ de ejemplo, App.sqlj, utiliza SQL estático para recuperar y actualizar datos de la tabla EMPLOYEE de la base de datos de ejemplo DB2.

Ejemplo: intercalar sentencias SQL en la aplicación Java:

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
import java.sql.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;

#sql iterator App_Cursor1 (String empno, String firstnme) ; // 1
#sql iterator App_Cursor2 (String) ;

class App
{

  /**********************
   **  Controlador de registro **
   **********************/

 static
 {
       try
   {
     Class.forName("com.ibm.db2.jdbc.app.DB2Driver").newInstance();
   }
   catch (Exception e)
   {
         e.printStackTrace();
   }
 }

  /********************
   **      Main      **
   ********************/

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

```
    try
{
  App_Cursor1 cursor1;
  App_Cursor2 cursor2;

  String str1 = null;
  String str2 = null;
  long   count1;

  // El URL es jdbc:db2:nombredb
  String url = "jdbc:db2:sample";

  DefaultContext ctx = DefaultContext.getDefaultContext();
  if (ctx == null)
  {
    try
    {
      // Se conecta con id/contraseña por omisión.
      Connection con = DriverManager.getConnection(url);
      con.setAutoCommit(false);
      ctx = new DefaultContext(con);
    }
    catch (SQLException e)
    {
      System.out.println("Error: no se ha podido obtener un contexto por omisión");
      System.err.println(e) ;
         System.exit(1);
    }
    DefaultContext.setDefaultContext(ctx);
  }

  // Se recuperan datos de la base de datos.
  System.out.println("Recuperar algunos datos de la base de datos.");
  #sql cursor1 = {SELECT empno, firstnme FROM employee}; // 2

  // Se visualiza el conjunto de resultados.
  // cursor1.next() devuelve false cuando no hay más filas.
  System.out.println("Resultados recibidos:");
  while (cursor1.next()) // 3 (page 290)
  {
    str1 = cursor1.empno(); // 4 (page 290)
    str2 = cursor1.firstnme();

    System.out.print (" empno= " + str1);
    System.out.print (" firstname= " + str2);
    System.out.println("");
  }
  cursor1.close(); // 9 (page 291)

  // Se recupera el número de empleado de la base de datos.
  #sql { SELECT count(*) into :count1 FROM employee }; // 5
  if (1 == count1)
    System.out.println ("Hay una fila en la tabla de empleados");
  else
    System.out.println ("Hay " + count1
                        + " filas en la tabla de empleados");
```

```
      // Se actualiza la base de datos.
      System.out.println("Actualizar la base de datos.");
      #sql { UPDATE employee SET firstnme = 'SHILI' WHERE empno = '000010' };

      // Se recuperan los datos actualizados de la base de datos.
      System.out.println("Recuperar los datos actualizados de la base de datos.");
      str1 = "000010";
      #sql cursor2 = {SELECT firstnme FROM employee WHERE empno = :str1}; // 6

      // Se visualiza el conjunto de resultados.
      // cursor2.next() devuelve false cuando no hay más filas.
      System.out.println("Resultados recibidos:");
      while (true)
      {
        #sql { FETCH :cursor2 INTO :str2 }; // 7 (page 291)
        if (cursor2.endFetch()) break; // 8 (page 291)

        System.out.print (" empno= " + str1);
        System.out.print (" firstname= " + str2);
        System.out.println("");
      }
      cursor2.close(); // 9 (page 291)

      // Se retrotrae la actualización.
      System.out.println("Retrotraer la actualización.");
      #sql { ROLLBACK work };
      System.out.println("Retrotracción terminada.");
    }
    catch( Exception e )
    {
          e.printStackTrace();
    }
  }
}
```

1. **Declarar iteradores.** Esta sección declara dos tipos de iteradores:

   App_Cursor1
   Declara nombres y tipos de datos de columna, y devuelve los valores de las columnas de acuerdo con el nombre de columna (enlace por nombre con columnas).

   App_Cursor2
   Declara tipos de datos de columna, y devuelve los valores de las columnas por posición de columna (enlace posicional con columnas).
2. **Inicializar el iterador.** El objeto iterador cursor1 se inicializa utilizando el resultado de una consulta. La consulta almacena el resultado en cursor1.
3. **Adelantar el iterador a la próxima fila.** El método cursor1.next() devuelve un Boolean `false` si no existen más filas para recuperar.
4. **Mover los datos.** El método de acceso por nombre empno() devuelve el valor de la columna denominada empno en la fila actual. El método de acceso por nombre firstnme() devuelve el valor de la columna denominada firstnme en la fila actual.
5. **Aplicar SELECT a los datos en una variable del lenguaje principal.** La sentencia SELECT pasa el número de filas de la tabla a la variable de lenguaje principal count1.
6. **Inicializar el iterador.** El objeto iterador cursor2 se inicializa utilizando el resultado de una consulta. La consulta almacena el resultado en cursor2.

7. **Recuperar los datos.** La sentencia FETCH devuelve el valor actual de la primera columna declarada en el cursor ByPos desde la tabla de resultados a la variable de lenguaje principal str2.
8. **Comprobar el éxito de una sentencia FETCH..INTO.** El método endFetch() devuelve Boolean true si el iterador no está situado en una fila, es decir, si el último intento de extraer una fila ha fallado. El método endFetch() devuelve false si el último intento de extraer una fila ha sido satisfactorio. DB2 intenta extraer una fila cuando se llama al método next(). Una sentencia FETCH...INTO llama implícitamente al método next().
9. **Cerrar los iteradores.** El método close() libera los recursos retenidos por los iteradores. Los iteradores se deben cerrar explícitamente para asegurar que se liberan los recursos del sistema de forma oportuna.

Para obtener información previa relacionada con este ejemplo, consulte la sección Intercalar sentencias SQL en la aplicación Java.

Ejemplo: finalizar una transacción

Este es un ejemplo de cómo finalizar una transacción en la aplicación.

Ejemplo: finalizar una transacción

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.transaction.*;
import javax.transaction.xa.*;
import com.ibm.db2.jdbc.app.*;

public class JTATxEnd {

    public static void main(java.lang.String[] args) {
        JTATxEnd test = new JTATxEnd();

        test.setup();
        test.run();
    }


    /**
     * Manejar la ejecución de limpieza anterior para que esta prueba pueda volver a empezar.
     */
   public void setup() {

        Connection c = null;
        Statement s = null;
            try {
   Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            c = DriverManager.getConnection("jdbc:db2:*local");
            s = c.createStatement();

           try {
                s.executeUpdate("DROP TABLE CUJOSQL.JTATABLE");
            } catch (SQLException e) {
                // Ignorar... no existe
```

```
        }

        s.executeUpdate("CREATE TABLE CUJOSQL.JTATABLE (COL1 CHAR (50))");
        s.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('Fun with JTA')");
        s.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('JTA is fun.)");

s.close();
     } finally {
             if (c != null) {
                 c.close();
        }
     }
 }


 /**
  * Esta prueba utiliza el soporte JTA para manejar transacciones.
  */
public void run() {
     Connection c = null;

        try {
        Context ctx = new InitialContext();

        // Presuponer que el origen de datos se apoya en un UDBXADataSource.
        UDBXADataSource ds = (UDBXADataSource) ctx.lookup("XADataSource");

        // Desde el DataSource, obtener un objeto XAConnection que
        // contiene un XAResource y un objeto Connection.
        XAConnection  xaConn = ds.getXAConnection();
        XAResource    xaRes  = xaConn.getXAResource();
        Connection    c      = xaConn.getConnection();

        // Para transacciones XA, es necesario un identificador de transacción.
        // No se incluye una implementación de la interfaz XID con
        // el controlador JDBC. Consulte "Transacciones con JTA" en la página 77
        // para obtener una descripción de esta interfaz para crear una clase para ella.
        Xid xid = new XidImpl();

        // La conexión de XAResource puede utilizarse como
        // cualquier otra conexión JDBC.
        Statement stmt = c.createStatement();

        // Debe informarse al recurso XA antes de iniciar cualquier
        // trabajo transaccional.
        xaRes.start(xid, XAResource.TMNOFLAGS);

        // Crear un ResultSet durante el proceso de JDBC y extraer una fila.
        ResultSet rs = stmt.executeUpdate("SELECT * FROM CUJOSQL.JTATABLE");
        rs.next();

        // Cuando se llama al método end, se cierran todos los cursores de ResultSet.
        // El intento de acceder a ResultSet después de este punto provoca
        // el lanzamiento de una excepción.
        xaRes.end(xid, XAResource.TMNOFLAGS);
```

```
            try {
                 String value = rs.getString(1);
                 System.out.println("Algo ha fallado si recibe este mensaje.");
             } catch (SQLException e) {
                 System.out.println("Se ha lanzado la excepción esperada.");
             }

             // Comprometer la transacción para asegurarse de que se liberan
             // todos los bloqueos.
             int rc = xaRes.prepare(xid);
             xaRes.commit(xid, false);


                     } catch (Exception e) {
             System.out.println("Algo ha fallado.");
          e.printStackTrace();
         } finally {
            try {
                 if (c != null)
                     c.close();
             } catch (SQLException e) {
                 System.out.println("Nota:  Excepción de limpieza.");
          e.printStackTrace();
            }
         }
     }
}
```

Ejemplo: ID de usuario y contraseña no válidos

Este es un ejemplo de utilización de la propiedad Connection en la modalidad de denominación SQL.

Ejemplo: ID de usuario y contraseña no válidos

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
//////////////////////////////////////////////////////////////////////////////
//
// Ejemplo de InvalidConnect.
//
// Este programa utiliza la propiedad Connection en modalidad de denominación SQL.
//
//////////////////////////////////////////////////////////////////////////////
//
// Este código fuente es un ejemplo del controlador JDBC de IBM Developer para Java.
// IBM le otorga una licencia no exclusiva para utilizarlo como un ejemplo
// a partir del cual puede generar funciones similares adaptadas
// a sus necesidades específicas.
//
// IBM suministra este código de ejemplo sólo con propósito ilustrativo.
// Estos ejemplos no se han probado exhaustivamente bajo todas las
// condiciones. Por tanto, IBM no puede garantizar la
// fiabilidad, capacidad de servicio o funcionamiento de estos programas.
//
// Todos los programas que contiene este ejemplo se suministran "TAL CUAL",
// sin garantías de ninguna clase. Se renuncia explícitamente a las garantías
// implícitas de comercialización y adecuación a un propósito
// determinado.
//
// IBM Developer Kit para Java
```

```
// (C) Copyright IBM Corp. 2001
// Reservados todos los derechos.
// US Government Users Restricted Rights -
// Use, duplication, or disclosure restricted
// by GSA ADP Schedule Contract with IBM Corp.
//
////////////////////////////////////////////////////////////////////////////
import java.sql.*;
import java.util.*;

public class InvalidConnect {

    public static void main(java.lang.String[] args)
    {
   // Se registra el controlador.
            try {
   Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
        } catch (ClassNotFoundException cnf) {
            System.out.println("ERROR: el controlador JDBC no se ha cargado.");
        System.exit(0);
        }

        // Intento de obtener una conexión sin especificar ningún usuario o
        // contraseña. El intento funciona y la conexión utiliza el
        // mismo perfil de usuario bajo el que se ejecuta el trabajo.
            try {
            Connection c1 = DriverManager.getConnection("jdbc:db2:*local");
            c1.close();
            } catch (SQLException e) {
            System.out.println("Esta prueba no debe incluirse en esta vía de acceso de excepciones.");
         e.printStackTrace();
            System.exit(1);
        }

            try {
            Connection c2 = DriverManager.getConnection("jdbc:db2:*local",
                                                        "notvalid", "notvalid");
            } catch (SQLException e) {
            System.out.println("Este es un error esperado.");
            System.out.println("El mensaje es " + e.getMessage());
            System.out.println("SQLSTATE es " + e.getSQLState());
        }

    }
}
```

Ejemplo: JDBC

Este es un ejemplo de utilización del programa BasicJDBC.

Ejemplo: BasicJDBC

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
////////////////////////////////////////////////////////////////////////////
//
// Ejemplo de BasicJDBC. Este programa utiliza el controlador JDBC nativo para el
// Developer Kit para Java para crear una tabla simple y procesar una consulta
// que visualice los datos de dicha tabla.
//
// Sintaxis de mandato:
//    BasicJDBC
//
////////////////////////////////////////////////////////////////////////////
//
// Este código fuente es un ejemplo del controlador JDBC de IBM Developer para Java.
```

```
// IBM le otorga una licencia no exclusiva para utilizarlo como un ejemplo
// a partir del cual puede generar funciones similares adaptadas
// a sus necesidades específicas.
//
// IBM suministra este código de ejemplo sólo con propósito ilustrativo.
// Estos ejemplos no se han probado exhaustivamente bajo todas las
// condiciones. Por tanto, IBM no puede garantizar la
// fiabilidad, capacidad de servicio o funcionamiento de estos programas.
//
// Todos los programas que contiene este ejemplo se suministran "TAL CUAL",
// sin garantías de ninguna clase. Se renuncia explícitamente a las garantías
// implícitas de comercialización y adecuación a un propósito
// determinado.
//
// IBM Developer Kit para Java
// (C) Copyright IBM Corp. 2001
// Reservados todos los derechos.
// US Government Users Restricted Rights -
// Use, duplication, or disclosure restricted
// by GSA ADP Schedule Contract with IBM Corp.
//
////////////////////////////////////////////////////////////////////////////

// Incluir las clases Java que deben utilizarse. En esta aplicación,
// se utilizan muchas clases del paquete java.sql y también se utiliza
// la clase java.util.Properties como parte del proceso de obtención
// de una conexión con la base de datos.
import java.sql.*;
import java.util.Properties;

// Crear una clase pública para encapsular el programa.
public class BasicJDBC {

    // La conexión es una variable privada del objeto.
    private Connection connection = null;

       // Cualquier clase que deba ser un "punto de entrada" para ejecutar
       // un programa debe tener un método main. El método main
       // es el punto donde se inicia el proceso cuando se llama al programa.
    public static void main(java.lang.String[] args) {

        // Crear un objeto de tipo BasicJDBC. Esto
        // es fundamental para la programación orientada a objetos. Una vez que
        // se ha creado un objeto, se llama a diversos métodos en
        // ese objeto para realizar el trabajo.
        // En este caso, al llamar al constructor del objeto
        // se crea una conexión de base de datos que los otros
        // métodos utilizan para realizar el trabajo en la base de datos.
        BasicJDBC test = new BasicJDBC();

        // Llamar al método rebuildTable. Este método garantiza que
        // la tabla utilizada en este programa existe y tiene el aspecto
        // adecuado. El valor de retorno es un valor booleano que indica
        // si la reconstrucción de la tabla se ha completado
        // satisfactoriamente. Si no es así, se visualiza un mensaje
        // y se sale del programa.
        if (!test.rebuildTable()) {
            System.out.println("Failure occurred while setting up " +
                               " for running the test.");
            System.out.println("Test will not continue.");
        System.exit(0);
        }

        // A continuación, se llama al método de ejecución de consulta.
        // Este método procesa una sentencia SQL select en la tabla
        // creada en el método rebuildTable. La salida de
        // esa consulta es la salida de la salida estándar de visualización.
```

```
        test.runQuery();

        // Finalmente, se llama al método cleanup. Este método
        // garantiza que la conexión de base de datos en la que el objeto
        // ha estado a la espera se ha cerrado.
        test.cleanup();
    }


    /**
    Este es el constructor de la prueba básica JDBC. Crea una conexión de
    base de datos que se almacena en una variable de instancia que se utilizará en
    posteriores llamadas de método.
    **/
    public BasicJDBC() {

        // Una forma de crear una conexión de base de datos es pasar un URL
        // y un objeto Properties Java a DriverManager. El siguiente
        // código construye un objeto Properties que contiene el ID de usuario y
        // la contraseña. Estas partes de información se utilizan para conectarse
        // a la base de datos.
            Properties properties = new Properties ();
        properties.put("user", "cujo");
        properties.put("user", "newtiger");

            // Utilizar un bloque try/catch para capturar todas las excepciones que
            // puedan surgir del código siguiente.
            try {
            // DriverManager debe tener conocimiento de que existe un controlador
            // JDBC disponible para manejar una petición de conexión de usuario.
            // La siguiente línea provoca que el controlador JDBC nativo se cargue
            // y registre con DriverManager.
    Class.forName("com.ibm.db2.jdbc.app.DB2Driver");

            // Crear el objeto Connection de base de datos que este programa utiliza
            // en todas las demás llamadas de método que se realizan. El código
            // siguiente especifica que debe establecerse una conexión con la base
            // de datos local y que dicha conexión debe ajustarse a las propiedades
            // configuradas anteriormente (es decir, debe utilizar el ID de usuario
            // y la contraseña especificados).
            connection = DriverManager.getConnection("jdbc:db2:*local", properties);

                    } catch (Exception e) {
            // Si falla alguna de las líneas del bloque try/catch, el control
            // se transfiere a la siguiente línea de código. Una aplicación
            // robusta intenta manejar el problema o proporcionar más detalles
            // al usuario. En este programa se visualiza mensaje de error
            // de la excepción y la aplicación permite el retorno del programa.
            System.out.println("Caught exception: " + e.getMessage());
        }
    }


    /**
    Garantiza que la tabla qgpl.basicjdbc tiene el aspecto deseado al principio de
    la prueba.

    @returns boolean    Devuelve true si la tabla se ha reconstruido
                        satisfactoriamente. Devuelve false si se ha
                        producido alguna anomalía.
    **/
    public boolean rebuildTable() {
        // Reiniciar todas las funciones de un bloque try/catch para que se realice
        // un intento de manejar los errores que puedan producirse dentro de este
        // método.
            try {
```

```
        // Se utilizan objetos Statement para procesar sentencias SQL en la
        // base de datos. El objeto Connection se utiliza para crear un
        // objeto Statement.
        Statement s = connection.createStatement();

       try {
            // Construir la tabla de prueba desde cero. Procesar y actualizar la
            // sentencia que intenta suprimir la tabla si existe actualmente.
            s.executeUpdate("eliminar tabla qgpl.basicjdbc");
        } catch (SQLException e) {
            // No realizar ninguna operación si se ha producido una excepción.
            // Se presupone que el problema es que la tabla que se ha eliminado
            // no existe y que puede crearse a continuación.
        }

        // Utilizar el objeto de sentencia para crear la tabla.
        s.executeUpdate("crear tabla qgpl.basicjdbc(id int, name char(15))");

        // Utilizar el objeto de sentencia para llenar la tabla con algunos
        // datos.
        s.executeUpdate("insertar en valores de qgpl.basicjdbc (1, 'Frank Johnson')");
        s.executeUpdate("insertar en valores de qgpl.basicjdbc (2, 'Neil Schwartz')");
        s.executeUpdate("insertar en valores de qgpl.basicjdbc (3, 'Ben Rodman')");
        s.executeUpdate("insertar en valores de qgpl.basicjdbc (4, 'Dan Gloore')");

        // Cerrar la sentencia SQL para indicar a la base de datos que ya no es
        // necesaria.
s.close();

        // Si todo el método se ha procesado satisfactoriamente, devolver true.
        // En este punto, la tabla se ha creado o renovado correctamente.
        return true;

    } catch (SQLException sqle) {
        // Si ha fallado alguna de las sentencias SQL (que no sea la eliminación
        // de la tabla manejada en el bloque try/catch interno), se visualiza
        // el mensaje de error y se devuelve false al llamador, indicando que
        // la tabla no puede completarse.
        System.out.println("Error in rebuildTable: " + sqle.getMessage());
        return false;
    }
}



/**
Ejecuta una consulta a la tabla de muestra y los resultados se visualizan en
la salida estándar.
**/
public void runQuery() {
    // Reiniciar todas las funciones de un bloque try/catch para que se realice
    // un intento de manejar los errores que puedan producirse dentro de este
    // método.
        try {
        // Crear un objeto Statement.
        Statement s = connection.createStatement();

        // Utilizar el objeto de sentencia para ejecutar una consulta SQL. Las
        // consultas devuelven objetos ResultSet que se utilizan para observar
        // los datos que proporciona la consulta.
        ResultSet rs = s.executeQuery("select * from qgpl.basicjdbc");

        // Visualizar el principio de la 'tabla' e inicializar el contador del
        // número de filas devueltas.
        System.out.println("--------------------");
        int i = 0;
```

```
            // El método next de ResultSet se utiliza para procesar las filas de un
            // ResultSet. Debe llamarse una vez al método next antes de que
            // los primeros datos estén disponibles para visualización. Si next
            // devuelve true, existe otra fila de datos que puede utilizarse.
   while (rs.next()) {

                // Obtener ambas columnas de la tabla para cada fila y escribir una
                // fila en la tabla de pantalla con los datos. A continuación,
                // aumentar la cuenta de filas que se han procesado.
                System.out.println("| " + rs.getInt(1) + " | " + rs.getString(2) + "|");
                i++;
            }

            // Colocar un límite al final de la tabla y visualizar el número de filas
            // como salida.
            System.out.println("--------------------");
            System.out.println("There were " + i + " rows returned.");
            System.out.println("Output is complete.");

            } catch (SQLException e) {
            // Visualizar más información acerca de las excepciones SQL
            // generadas como salida.
            System.out.println("SQLException exception: ");
            System.out.println("Message:....." + e.getMessage());
            System.out.println("SQLState:...." + e.getSQLState());
            System.out.println("Vendor Code:." + e.getErrorCode());
         e.printStackTrace();
        }
    }


    /**
    El siguiente método garantiza que se liberen los recursos JDBC que aún
    están asignados.
    **/
    public void cleanup() {
            try {
            if (connection != null)
                connection.close();
                    } catch (Exception e) {
            System.out.println("Excepción capturada: ");
         e.printStackTrace();
        }
    }
}
```

Ejemplo: varias conexiones que funcionan en una transacción

Este es un ejemplo de utilización de varias conexiones que trabajan en una sola transacción.

Ejemplo: Varias conexiones que trabajan en una transacción

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.transaction.*;
import javax.transaction.xa.*;
import com.ibm.db2.jdbc.app.*;
public class JTAMultiConn {
    public static void main(java.lang.String[] args) {
      JTAMultiConn test = new JTAMultiConn();
      test.setup();
      test.run();
   }
```

```
/**
* Manejar la ejecución de limpieza anterior para que esta prueba pueda volver a empezar.
*/
   public void setup() {
        Connection c = null;
        Statement s = null;
           try {
   Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
         c = DriverManager.getConnection("jdbc:db2:*local");
            s = c.createStatement();
           try {
            s.executeUpdate("DROP TABLE CUJOSQL.JTATABLE");
         }
         catch (SQLException e) {
         // Ignorar... no existe
         }
         s.executeUpdate("CREATE TABLE CUJOSQL.JTATABLE (COL1 CHAR
                         (50))");
   s.close();
      }
      finally {
                if (c != null) {
                    c.close();
         }
      }
   }
/**
* Esta prueba utiliza el soporte JTA para manejar transacciones.
*/
   public void run() {
      Connection c1 = null;
      Connection c2 = null;
      Connection c3 = null;
           try {
         Context ctx = new InitialContext();
         // Presuponer que el origen de datos se apoya en un UDBXADataSource.
         UDBXADataSource ds = (UDBXADataSource)
                              ctx.lookup("XADataSource");
         // Desde el DataSource, obtener algunos objetos XAConnection que
         // contienen un XAResource y un objeto Connection.
         XAConnection xaConn1 = ds.getXAConnection();
         XAConnection xaConn2 = ds.getXAConnection();
         XAConnection xaConn3 = ds.getXAConnection();
         XAResource xaRes1 = xaConn1.getXAResource();
         XAResource xaRes2 = xaConn2.getXAResource();
         XAResource xaRes3 = xaConn3.getXAResource();
         c1 = xaConn1.getConnection();
         c2 = xaConn2.getConnection();
         c3 = xaConn3.getConnection();
         Statement stmt1 = c1.createStatement();
         Statement stmt2 = c2.createStatement();
         Statement stmt3 = c3.createStatement();
         // Para transacciones XA, es necesario un identificador de transacción.
         // El soporte para crear XIDs se deja de nuevo al
         // programa de aplicación.
         Xid xid = JDXATest.xidFactory();
         // Realizar algún trabajo transaccional bajo cada una de las tres
         // conexiones que se han creado.
         xaRes1.start(xid, XAResource.TMNOFLAGS);
         int count1 = stmt1.executeUpdate("INSERT INTO " + tableName + "VALUES('Value 1-A')");
         xaRes1.end(xid, XAResource.TMNOFLAGS);

         xaRes2.start(xid, XAResource.TMJOIN);
         int count2 = stmt2.executeUpdate("INSERT INTO " + tableName + "VALUES('Value 1-B')");
         xaRes2.end(xid, XAResource.TMNOFLAGS);

         xaRes3.start(xid, XAResource.TMJOIN);
```

```
        int count3 = stmt3.executeUpdate("INSERT INTO " + tableName + "VALUES('Value 1-C')");
        xaRes3.end(xid, XAResource.TMSUCCESS);
        // Al terminar, comprometer la transacción como una sola unidad.
        // Es necesario prepare() y commit() o commit() de 1 fase para
        // cada base de datos independiente (XAResource) que ha participado en la
        // transacción. Dado que los recursos a los que se ha accedido
        // (xaRes1, xaRes2 y xaRes3)
        // hacen todos referencia a la misma base de datos, sólo es necesaria una
        // prepare o commit.
        int rc = xaRes.prepare(xid);
        xaRes.commit(xid, false);
      }
    catch (Exception e) {
        System.out.println("Algo ha fallado.");
        e.printStackTrace();
      }
      finally {
           try {
            if (c1 != null) {
               c1.close();
             }
        }
        catch (SQLException e) {
           System.out.println("Nota: Excepción de limpieza " +
                              e.getMessage());
        }
          try {
           if (c2 != null) {
              c2.close();
             }
        }
        catch (SQLException e) {
           System.out.println("Nota: Excepción de limpieza " +
                              e.getMessage());
        }
          try {
           if (c3 != null) {
              c3.close();
             }
        }
        catch (SQLException e) {
           System.out.println("Nota: Excepción de limpieza " +
                              e.getMessage());
        }
      }
   }
}
```

Ejemplo: obtener un contexto inicial antes de enlazar UDBDataSource

En el ejemplo siguiente se obtiene un contexto inicial antes de enlazar el UDBDataSource. A continuación, se utiliza el método lookup en ese contexto para devolver un objeto de tipo DataSource para que lo utilice la aplicación.

Ejemplo: obtener un contexto inicial antes de enlazar UDBDataSource

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
// Importar los paquetes necesarios. No es necesario ningún
// código específico de controlador en aplicaciones
// de ejecución.
import java.sql.*;
import javax.sql.*;
import javax.naming.*;
```

```
public class UDBDataSourceUse
{
    public static void main(java.lang.String[] args)
    throws Exception
    {
        // Recuperar un contexto JNDI. El contexto funciona
        // como raíz donde se enlazan o se encuentran
        // los objetos en JNDI.
        Context ctx = new InitialContext();

        // Recuperar el objeto UDBDataSource enlazado mediante el
        // nombre con el que estaba enlazado anteriormente. Durante
        // la ejecución, sólo se utiliza la interfaz DataSource,
        // y por tanto no es necesario convertir el objeto a la clase
        // de implementación de UDBdataSource. (No es necesario saber cuál
        // es la clase de implementación. Sólo es necesario el nombre
        // lógico de JNDI).
        DataSource ds = (DataSource) ctx.lookup("SimpleDS");


        // Una vez obtenido el DataSource, puede utilizarse para establecer
        // una conexión. Este objeto Connection es el mismo tipo
        // de objeto que se devuelve si se utiliza el método DriverManager
        // para establecer la conexión. Por tanto, a partir de este
        // punto todo es exactamente igual que en cualquier otra
        // aplicación JDBC.
        Connection connection = ds.getConnection();

        // La conexión puede utilizarse para crear objetos Statement y
        // actualizar la base de datos o procesar consultas de la forma siguiente.
        Statement statement = connection.createStatement();
        ResultSet rs = statement.executeQuery("select * from qsys2.sysprocs");
   while (rs.next()) {
            System.out.println(rs.getString(1) + "." + rs.getString(2));
        }

        // La conexión se cierra antes de que finalice la aplicación.
        connection.close();
    }
}
```

Ejemplo: ParameterMetaData

Este es un ejemplo de utilización de la interfaz ParameterMetaData para recuperar información acerca de los parámetros.

Ejemplo: ParameterMetaData

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
//////////////////////////////////////////////////////////////////////////////
//
// Ejemplo de ParameterMetaData. Este programa muestra
// el nuevo soporte de JDBC 3.0 para obtener información
// acerca de los parámetros de una PreparedStatement.
//
// Sintaxis de mandatos:
//    java PMD
//
//////////////////////////////////////////////////////////////////////////////
//
// Este código fuente es un ejemplo del controlador JDBC de IBM Developer para Java.
// IBM le otorga una licencia no exclusiva para utilizarlo como un ejemplo
// a partir del cual puede generar funciones similares adaptadas
// a sus necesidades específicas.
```

```
//
// IBM suministra este código de ejemplo sólo con propósito ilustrativo.
// Estos ejemplos no se han probado exhaustivamente bajo todas las
// condiciones. Por tanto, IBM no puede garantizar la
// fiabilidad, capacidad de servicio o funcionamiento de estos programas.
//
// Todos los programas que contiene este ejemplo se suministran "TAL CUAL",
// sin garantías de ninguna clase. Se renuncia explícitamente a las garantías
// implícitas de comercialización y adecuación a un propósito
// determinado.
//
// IBM Developer Kit para Java
// (C) Copyright IBM Corp. 2001
// Reservados todos los derechos.
// US Government Users Restricted Rights -
// Use, duplication, or disclosure restricted
// by GSA ADP Schedule Contract with IBM Corp.
//
////////////////////////////////////////////////////////////////////////////

import java.sql.*;

public class PMD {

    // Punto de entrada del programa.
    public static void main(java.lang.String[] args)
    throws Exception
    {
        // Obtener configuración.
   Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
        Connection c = DriverManager.getConnection("jdbc:db2:*local");
        PreparedStatement ps = c.prepareStatement("INSERT INTO CUJOSQL.MYTABLE VALUES(?, ?, ?)");
        ParameterMetaData pmd = ps.getParameterMetaData();

        for (int i = 1; i < pmd.getParameterCount(); i++) {
            System.out.println("Número parámetro " + i);
            System.out.println("  El nombre de clase es " + pmd.getParameterClassName(i));
            // Nota: la modalidad hace referencia a entrada, salida y entrada/salida
            System.out.println("  La modalidad es " + pmd.getParameterClassName(i));
            System.out.println("  El tipo es " + pmd.getParameterType(i));
            System.out.println("  El nombre de tipo es " + pmd.getParameterTypeName(i));
            System.out.println("  La precisión es " + pmd.getPrecision(i));
            System.out.println("  La escala es " + pmd.getScale(i));
            System.out.println("  Nullable? es " + pmd.isNullable(i));
            System.out.println("  Signed? es " + pmd.isSigned(i));
        }
    }
}
```

Ejemplo: cambiar valores con una sentencia mediante el cursor de otra sentencia

Este es un ejemplo de cómo cambiar valores con una sentencia mediante el cursor de otra sentencia.

Ejemplo: cambiar valores con una sentencia mediante el cursor de otra sentencia

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
import java.sql.*;

public class UsingPositionedUpdate {
    public Connection connection = null;
    public static void main(java.lang.String[] args) {

        UsingPositionedUpdate test = new UsingPositionedUpdate();
```

```
        test.setup();
        test.displayTable();

        test.run();
        test.displayTable();

        test.cleanup();
    }



/**
Manejar todo el trabajo de configuración necesario.
**/
    public void setup() {
            try {
            // Registrar el controlador JDBC.
   Class.forName("com.ibm.db2.jdbc.app.DB2Driver");

            connection = DriverManager.getConnection("jdbc:db2:*local");

            Statement s = connection.createStatement();
           try {
                s.executeUpdate("DROP TABLE CUJOSQL.WHERECUREX");
            } catch (SQLException e) {
                // Aquí, pasar por alto los problemas.
            }

            s.executeUpdate("CREATE TABLE CUJOSQL.WHERECUREX ( " +
                            "COL_IND INT, COL_VALUE CHAR(20)) ");

            for (int i = 1; i <= 10; i++) {
                s.executeUpdate("INSERT INTO CUJOSQL.WHERECUREX VALUES(" + i + ", 'FIRST')");
            }

   s.close();

                    } catch (Exception e) {
            System.out.println("Excepción capturada: " + e.getMessage());
         e.printStackTrace();
        }
    }


/**
En esta sección, debe añadirse todo el código destinado a realizar
la prueba. Si sólo es necesaria una conexión con la base de datos,
puede utilizarse la variable global 'connection'.
**/
    public void run() {
            try {
            Statement stmt1 = connection.createStatement();

            // Actualizar cada valor utilizando next().
            stmt1.setCursorName("CUJO");
            ResultSet rs = stmt1.executeQuery ("SELECT * FROM CUJOSQL.WHERECUREX " +
                                               "FOR UPDATE OF COL_VALUE");

            System.out.println("El nombre de cursor es " + rs.getCursorName());

            PreparedStatement stmt2 = connection.prepareStatement ("UPDATE "
                                                                   + " CUJOSQL.WHERECUREX
                                                                       SET COL_VALUE = 'CHANGED'
                                                                       WHERE CURRENT OF "
                                                                   + rs.getCursorName ());
```

```
            // Explorar en bucle el ResultSet y actualizar todas las demás entradas.
            while (rs.next()) {
                if (rs.next())
                    stmt2.execute ();
            }


            // Borrar los recursos una vez utilizados.
            rs.close ();
            stmt2.close ();


                    } catch (Exception e) {
            System.out.println("Excepción capturada: ");
         e.printStackTrace();
        }
    }



/**
En esta sección, colocar todo el trabajo de borrado para la prueba.
**/
    public void cleanup() {
            try {
            // Cerrar la conexión global abierta en setup().
            connection.close();

                    } catch (Exception e) {
            System.out.println("Excepción capturada: ");
         e.printStackTrace();
        }
    }



/**
Visualizar el contenido de la tabla.
**/
    public void displayTable()
    {
            try {
            Statement s = connection.createStatement();
            ResultSet rs = s.executeQuery ("SELECT * FROM CUJOSQL.WHERECUREX");

            while (rs.next()) {
                System.out.println("Index " + rs.getInt(1) + " valor " + rs.getString(2));
            }

            rs.close ();
   s.close();
            System.out.println("----------------------------------------");
                    } catch (Exception e) {
            System.out.println("Excepción capturada: ");
         e.printStackTrace();
        }
    }
}
```

Ejemplo: interfaz ResultSet para IBM Developer Kit para Java

Este es un ejemplo de cómo utilizar la interfaz ResultSet.

Ejemplo 1: interfaz ResultSet

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
import java.sql.*;

/**
ResultSetExample.java

Este programa muestra la utilización de ResultSetMetaData y
ResultSet para visualizar todos los datos de una tabla aunque
el programa que obtiene los datos no sabe cuál es el aspecto
que tendrá la tabla (el usuario pasa los valores correspondientes
a la tabla y a la biblioteca).
**/
public class ResultSetExample {

    public static void main(java.lang.String[] args)
    {
        if (args.length != 2) {
            System.out.println("Utilización:  java ResultSetExample <biblioteca> <tabla>");
            System.out.println(" siendo <biblioteca> la biblioteca que contiene la <tabla>");
        System.exit(0);
        }

        Connection con = null;
        Statement s = null;
        ResultSet rs = null;
        ResultSetMetaData rsmd = null;

            try {
            // Se obtiene una conexión a base de datos y se prepara una sentencia.
   Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            con = DriverManager.getConnection("jdbc:db2:*local");

            s = con.createStatement();

            rs = s.executeQuery("SELECT * FROM " + args[0] + "." + args[1]);
            rsmd = rs.getMetaData();

            int colCount = rsmd.getColumnCount();
            int rowCount = 0;
   while (rs.next()) {
                rowCount++;
                System.out.println("Datos para la fila " + rowCount);
                for (int i = 1; i <= colCount; i++)
                    System.out.println("   Fila " + i + ": " + rs.getString(i));
            }

                    } catch (Exception e) {
            // Se manejan los errores.
            System.out.println("Tenemos un error... ");
         e.printStackTrace();
        } finally {
            // Hay que asegurarse de que siempre se haga
            // el borrado. Si la conexión se cierra, la
            // sentencia que hay debajo de ella también se cerrará.
            if (con != null) {
           try {
                    con.close();
            } catch (SQLException e) {
                    System.out.println("Error grave: no se puede cerrar el objeto de conexión");
                }
            }
        }
    }
}
```

Ejemplo: sensibilidad de ResultSet

El ejemplo siguiente muestra cómo un cambio puede afectar a una cláusula where de una sentencia SQL en función de la sensibilidad del ResultSet.

Ejemplo: sensibilidad de ResultSet

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
import java.sql.*;

public class Sensitive2 {

    public Connection connection = null;

    public static void main(java.lang.String[] args) {
        Sensitive2 test = new Sensitive2();

        test.setup();
        test.run("sensitive");
        test.cleanup();

        test.setup();
        test.run("insensitive");
        test.cleanup();
    }


   public void setup() {

            try {
            System.out.println("Se utiliza controlador JDBC nativo");
   Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            connection = DriverManager.getConnection("jdbc:db2:*local");

            Statement s = connection.createStatement();
           try {
                s.executeUpdate("drop table cujosql.sensitive");
            } catch (SQLException e) {
                // Se pasa por alto.
            }

            s.executeUpdate("create table cujosql.sensitive(col1 int)");
            s.executeUpdate("insert into cujosql.sensitive values(1)");
            s.executeUpdate("insert into cujosql.sensitive values(2)");
            s.executeUpdate("insert into cujosql.sensitive values(3)");
            s.executeUpdate("insert into cujosql.sensitive values(4)");
            s.executeUpdate("insert into cujosql.sensitive values(5)");


           try {
                s.executeUpdate("drop table cujosql.sensitive2");
            } catch (SQLException e) {
                // Se pasa por alto.
            }

            s.executeUpdate("create table cujosql.sensitive2(col2 int)");
            s.executeUpdate("insert into cujosql.sensitive2 values(1)");
            s.executeUpdate("insert into cujosql.sensitive2 values(2)");
            s.executeUpdate("insert into cujosql.sensitive2 values(3)");
            s.executeUpdate("insert into cujosql.sensitive2 values(4)");
            s.executeUpdate("insert into cujosql.sensitive2 values(5)");

   s.close();

                   } catch (Exception e) {
```

```
        System.out.println("Excepción capturada: " + e.getMessage());
        if (e instanceof SQLException) {
            SQLException another = ((SQLException) e).getNextException();
            System.out.println("Otra:  " + another.getMessage());
        }
    }
}


public void run(String sensitivity) {
        try {

    Statement s = null;
        if (sensitivity.equalsIgnoreCase("insensitive")) {
            System.out.println("creando cursor TYPE_SCROLL_INSENSITIVE");
            s = connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                ResultSet.CONCUR_READ_ONLY);
} else {
            System.out.println("creando cursor TYPE_SCROLL_SENSITIVE");
            s = connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                ResultSet.CONCUR_READ_ONLY);
        }

        ResultSet rs = s.executeQuery("select col1, col2 From cujosql.sensitive,
                                       cujosql.sensitive2 where col1 = col2");

        rs.next();
        System.out.println("el valor es " + rs.getInt(1));
        rs.next();
        System.out.println("el valor es " + rs.getInt(1));
        rs.next();
        System.out.println("el valor es " + rs.getInt(1));
        rs.next();
        System.out.println("el valor es " + rs.getInt(1));

        System.out.println("se han extraído las cuatro filas...");

        // Otra sentencia crea un valor que no se ajusta a la cláusula where.
        Statement s2 =
          connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATEABLE);
        ResultSet rs2 = s2.executeQuery("select * from cujosql.sensitive where col1 = 5 FOR UPDATE");
        rs2.next();
        rs2.updateInt(1, -1);
        rs2.updateRow();
        s2.close();

        if (rs.next()) {
            System.out.println("Aún existe una fila: " + rs.getInt(1));
} else {
            System.out.println("No hay más filas.");
        }

        } catch (SQLException e) {
        System.out.println("Excepción SQLException: ");
        System.out.println("Mensaje:....." + e.getMessage());
        System.out.println("SQLState:...." + e.getSQLState());
        System.out.println("Código proveedor:." + e.getErrorCode());
        System.out.println("-----------------------------------------------------");
     e.printStackTrace();
    }
    catch (Exception ex) {
        System.out.println("Se ha lanzado una excepción que no es una SQLException: ");
        ex.printStackTrace();
    }
}
```

```
    public void cleanup() {
            try {
            connection.close();
                    } catch (Exception e) {
            System.out.println("Excepción capturada: ");
         e.printStackTrace();
        }
    }
}
```

Ejemplo: ResultSets sensibles e insensibles

El ejemplo siguiente muestra la diferencia entre los ResultSets sensibles e insensibles cuando se insertan filas en una tabla.

Ejemplo: ResultSets sensibles e insensibles

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
import java.sql.*;

public class Sensitive {

    public Connection connection = null;

    public static void main(java.lang.String[] args) {
        Sensitive test = new Sensitive();

        test.setup();
        test.run("sensitive");
        test.cleanup();

        test.setup();
        test.run("insensitive");
        test.cleanup();
    }


   public void setup() {

            try {
   Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            connection = DriverManager.getConnection("jdbc:db2:*local");

            Statement s = connection.createStatement();
           try {
                s.executeUpdate("drop table cujosql.sensitive");
            } catch (SQLException e) {
                // Se pasa por alto.
            }

            s.executeUpdate("create table cujosql.sensitive(col1 int)");
            s.executeUpdate("insert into cujosql.sensitive values(1)");
            s.executeUpdate("insert into cujosql.sensitive values(2)");
            s.executeUpdate("insert into cujosql.sensitive values(3)");
            s.executeUpdate("insert into cujosql.sensitive values(4)");
            s.executeUpdate("insert into cujosql.sensitive values(5)");
   s.close();

                    } catch (Exception e) {
            System.out.println("Excepción capturada: " + e.getMessage());
            if (e instanceof SQLException) {
                SQLException another = ((SQLException) e).getNextException();
                System.out.println("Otra:  " + another.getMessage());
```

```
        }
    }
}


public void run(String sensitivity) {
        try {
    Statement s = null;
        if (sensitivity.equalsIgnoreCase("insensitive")) {
            System.out.println("creando cursor TYPE_SCROLL_INSENSITIVE");
            s = connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                ResultSet.CONCUR_READ_ONLY);
} else {
            System.out.println("creando cursor TYPE_SCROLL_SENSITIVE");
            s = connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                ResultSet.CONCUR_READ_ONLY);
        }

        ResultSet rs = s.executeQuery("select * From cujosql.sensitive");

        // Extraer los cinco valores que se encuentran allí.
        rs.next();
        System.out.println("el valor es " + rs.getInt(1));
        rs.next();
        System.out.println("el valor es " + rs.getInt(1));
        rs.next();
        System.out.println("el valor es " + rs.getInt(1));
        rs.next();
        System.out.println("el valor es " + rs.getInt(1));
        rs.next();
        System.out.println("el valor es " + rs.getInt(1));
        System.out.println("se han extraído las cinco filas...");

        // Nota:  Si extrae la última fila, el ResultSet cree
        //        que las filas cerradas y las subsiguientes nuevas
        //        que se han añadido no se reconocen.

        // Permitir que otra sentencia inserte un valor nuevo.
        Statement s2 = connection.createStatement();
        s2.executeUpdate("insert into cujosql.sensitive values(6)");
        s2.close();

        // El hecho de que una fila se reconozca se basa en el valor
        // de sensibilidad.
        if (rs.next()) {
            System.out.println("Ahora existe una fila: " + rs.getInt(1));
} else {
            System.out.println("No hay más filas.");
        }


        } catch (SQLException e) {
        System.out.println("Excepción SQLException: ");
        System.out.println("Mensaje:....." + e.getMessage());
        System.out.println("SQLState:...." + e.getSQLState());
        System.out.println("Código proveedor:." + e.getErrorCode());
        System.out.println("-----------------------------------------------------");
     e.printStackTrace();
    }
    catch (Exception ex) {
        System.out.println("Se ha lanzado una excepción que no es una SQLException: ");
        ex.printStackTrace();
    }
}
```

```
    public void cleanup() {
            try {
            connection.close();
                    } catch (Exception e) {
            System.out.println("Excepción capturada: ");
         e.printStackTrace();
        }
    }
}
```

Ejemplo: configurar una agrupación de conexiones con UDBDataSource y UDBConnectionPoolDataSource

Este es un ejemplo de cómo utilizar una agrupación de conexiones con UDBDataSource y UDBConnectionPoolDataSource.

Ejemplo: configurar una agrupación de conexiones con UDBDataSource y UDBConnectionPoolDataSource

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
import java.sql.*;
import javax.naming.*;
import com.ibm.db2.jdbc.app.UDBDataSource;
import com.ibm.db2.jdbc.app.UDBConnectionPoolDataSource;

public class ConnectionPoolingSetup
{
    public static void main(java.lang.String[] args)
    throws Exception
    {
        // Crear una implementación ConnectionPoolDataSource
        UDBConnectionPoolDataSource cpds = new UDBConnectionPoolDataSource();
        cpds.setDescription("Objeto DataSource de agrupación de conexiones");

        // Establecer un contexto JNDI y enlazar el origen de datos de agrupación
        // de conexiones
        Context ctx = new InitialContext();
        ctx.rebind("ConnectionSupport", cpds);

        // Crear un origen de datos estándar que haga referencia al mismo.
        UDBDataSource ds = new UDBDataSource();
        ds.setDescription("DataSource que soporta la agrupación");
        ds.setDataSourceName("ConnectionSupport");
        ctx.rebind("PoolingDataSource", ds);
  }
}
```

Ejemplo: SQLException

Este es un ejemplo de cómo capturar una SQLException y volcar toda la información que proporciona.

Ejemplo: SQLException

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
import java.sql.*;

public class ExceptionExample {

    public static Connection connection = null;

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

```
            try {
   Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            connection = DriverManager.getConnection("jdbc:db2:*local");

            Statement s = connection.createStatement();
            int count = s.executeUpdate("insertar en valores de cujofake.cujofake (1, 2,3)");

            System.out.println("No se esperaba que la tabla existiera.");


            } catch (SQLException e) {
            System.out.println("Excepción SQLException: ");
            System.out.println("Mensaje:....." + e.getMessage());
            System.out.println("SQLState:...." + e.getSQLState());
            System.out.println("Código proveedor:." + e.getErrorCode());
            System.out.println("-----------------------------------------------------");
         e.printStackTrace();
        } catch (Exception ex) {
            System.out.println("Se ha lanzado una excepción que no es una SQLException: ");
            ex.printStackTrace();
        } finally {
           try {
                if (connection != null) {
                    connection.close();
                }
            } catch (SQLException e) {
                System.out.println("Excepción capturada al intentar cerrar...");
            }
        }
    }
}
```

Ejemplo: suspender y reanudar una transacción

Este es un ejemplo de una transacción que se suspende y luego se reanuda.

Ejemplo: suspender y reanudar una transacción

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.transaction.*;
import javax.transaction.xa.*;
import com.ibm.db2.jdbc.app.*;

public class JTATxSuspend {

    public static void main(java.lang.String[] args) {
        JTATxSuspend test = new JTATxSuspend();

        test.setup();
        test.run();
    }


    /**
     * Manejar la ejecución de limpieza anterior para que esta prueba pueda volver a empezar.
     */
```

```
public void setup() {

     Connection c = null;
     Statement s = null;
        try {
Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
         c = DriverManager.getConnection("jdbc:db2:*local");
         s = c.createStatement();

        try {
             s.executeUpdate("DROP TABLE CUJOSQL.JTATABLE");
         } catch (SQLException e) {
             // Ignorar... no existe
         }

         s.executeUpdate("CREATE TABLE CUJOSQL.JTATABLE (COL1 CHAR (50))");
         s.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('Fun with JTA')");
         s.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('JTA is fun.)");

s.close();
     } finally {
             if (c != null) {
                 c.close();
         }
     }
 }


 /**
  * Esta prueba utiliza el soporte JTA para manejar transacciones.
  */
public void run() {
     Connection c = null;

        try {
         Context ctx = new InitialContext();

         // Presuponer que el origen de datos se apoya en un UDBXADataSource.
         UDBXADataSource ds = (UDBXADataSource) ctx.lookup("XADataSource");

         // Desde el DataSource, obtener un objeto XAConnection que
         // contiene un XAResource y un objeto Connection.
         XAConnection  xaConn = ds.getXAConnection();
         XAResource    xaRes  = xaConn.getXAResource();
         Connection    c      = xaConn.getConnection();

         // Para transacciones XA, es necesario un identificador de transacción.
         // No se incluye una implementación de la interfaz XID con
         // el controlador JDBC. Consulte "Transacciones con JTA" en la página 77
         // para obtener una descripción de esta interfaz para crear una clase para ella.
         Xid xid = new XidImpl();

         // La conexión de XAResource puede utilizarse como
         // cualquier otra conexión JDBC.
         Statement stmt = c.createStatement();
```

```
   // Debe informarse al recurso XA antes de iniciar cualquier
   // trabajo transaccional.
   xaRes.start(xid, XAResource.TMNOFLAGS);

   // Crear un ResultSet durante el proceso de JDBC y extraer una fila.
   ResultSet rs = stmt.executeUpdate("SELECT * FROM CUJOSQL.JTATABLE");
   rs.next();

   // Se llama al método end con la opción suspend. Los
   // ResultSets asociados con la transacción actual quedan 'suspendidos'.
   // En este estado, no se eliminan ni son accesibles.
   xaRes.end(xid, XAResource.TMSUSPEND);


   // Pueden realizarse otras tareas con la transacción.
   // Como ejemplo, puede crear una sentencia y procesar una consulta.
   // Este trabajo, y cualquier otro trabajo transaccional que la
   // transacción puede realizar, está separado del trabajo realizado
   // anteriormente bajo XID.
   Statement nonXAStmt = conn.createStatement();
   ResultSet nonXARS = nonXAStmt.executeQuery("SELECT * FROM CUJOSQL.JTATABLE");
   while (nonXARS.next()) {
       // Procesar aquí...
   }
   nonXARS.close();
   nonXAStmt.close();


   // Si se intenta utilizar recursos de transacciones
   // suspendidas, se produce una excepción.
  try {
       rs.getString(1);
       System.out.println("El valor de la primera fila es " + rs.getString(1));
   } catch (SQLException e) {
       System.out.println("Esta es una excepción esperada - " +
                          "se ha utilizado un ResultSet suspendido.");
   }


   // Reanudar la transacción suspendida y terminar el trabajo en ella.
   // El ResultSet es exactamente igual que antes de la suspensión.
   xaRes.start(newXid, XAResource.TMRESUME);
   rs.next();
   System.out.println("El valor de la segunda fila es " + rs.getString(1));


   // Cuando la transacción termine, finalizarla
   // y comprometer el trabajo que contenga.
   xaRes.end(xid, XAResource.TMNOFLAGS);
   int rc = xaRes.prepare(xid);
   xaRes.commit(xid, false);


           } catch (Exception e) {
   System.out.println("Algo ha fallado.");
e.printStackTrace();
```

```
        } finally {
           try {
                if (c != null)
                    c.close();
            } catch (SQLException e) {
                System.out.println("Nota:  Excepción de limpieza.");
         e.printStackTrace();
           }
        }
    }
}
```

Ejemplo: ResultSets suspendidos

Este es un ejemplo de cómo se reprocesa un objeto Statement bajo otra transacción para realizar el trabajo.

Ejemplo: ResultSets suspendidos

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.transaction.*;
import javax.transaction.xa.*;
import com.ibm.db2.jdbc.app.*;

public class JTATxEffect {

    public static void main(java.lang.String[] args) {
        JTATxEffect test = new JTATxEffect();

        test.setup();
        test.run();
    }


    /**
     * Manejar la ejecución de limpieza anterior para que esta prueba pueda volver a empezar.
     */
   public void setup() {

        Connection c = null;
        Statement s = null;
            try {
   Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            c = DriverManager.getConnection("jdbc:db2:*local");
            s = c.createStatement();

           try {
                s.executeUpdate("DROP TABLE CUJOSQL.JTATABLE");
            } catch (SQLException e) {
                // Ignorar... no existe
            }
```

```
        s.executeUpdate("CREATE TABLE CUJOSQL.JTATABLE (COL1 CHAR (50))");
        s.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('Fun with JTA')");
        s.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('JTA is fun.)");

s.close();
     } finally {
             if (c != null) {
                 c.close();
         }
     }
 }


 /**
  * Esta prueba utiliza el soporte JTA para manejar transacciones.
  */
public void run() {
     Connection c = null;

         try {
         Context ctx = new InitialContext();

         // Presuponer que el origen de datos se apoya en un UDBXADataSource.
         UDBXADataSource ds = (UDBXADataSource) ctx.lookup("XADataSource");

         // Desde el DataSource, obtener un objeto XAConnection que
         // contiene un XAResource y un objeto Connection.
         XAConnection  xaConn = ds.getXAConnection();
         XAResource    xaRes  = xaConn.getXAResource();
         Connection    c      = xaConn.getConnection();

         // Para transacciones XA, es necesario un identificador de transacción.
         // No se incluye una implementación de la interfaz XID con
         // el controlador JDBC. Consulte "Transacciones con JTA" en la página 77
         // para obtener una descripción de esta interfaz para crear una
         // clase para ella.
         Xid xid = new XidImpl();

         // La conexión de XAResource puede utilizarse como
         // cualquier otra conexión JDBC.
         Statement stmt = c.createStatement();

         // Debe informarse al recurso XA antes de iniciar cualquier
         // trabajo transaccional.
         xaRes.start(xid, XAResource.TMNOFLAGS);

         // Crear un ResultSet durante el proceso de JDBC y extraer una fila.
         ResultSet rs = stmt.executeUpdate("SELECT * FROM CUJOSQL.JTATABLE");
             rs.next();

         // Se llama al método end con la opción suspend. Los
         // ResultSets asociados con la transacción actual quedan 'suspendidos'.
         // En este estado, no se eliminan ni son accesibles.
         xaRes.end(xid, XAResource.TMSUSPEND);
```

```
            // Mientras tanto, pueden realizarse otras tareas fuera de la
            // transacción.
            // Los ResultSets bajo la transacción pueden cerrarse si
            // se reutiliza el objeto Statement utilizado para crearlos.
            ResultSet nonXARS = stmt.executeQuery("SELECT * FROM CUJOSQL.JTATABLE");
            while (nonXARS.next()) {
                // Procesar aquí...
            }


            // Intento de volver a la transacción suspendida. El ResultSet
            // de la transacción suspendida ha desaparecido debido a que
            // la sentencia se ha procesado de nuevo.
            xaRes.start(newXid, XAResource.TMRESUME);
           try {
                rs.next();
            } catch (SQLException ex) {
                System.out.println("Esta es una excepción esperada. " +
                                   "El ResultSet se ha cerrado debido a otro proceso.");
            }


            // Cuando la transacción termine, finalizarla
            // y comprometer el trabajo que contenga.
            xaRes.end(xid, XAResource.TMNOFLAGS);
            int rc = xaRes.prepare(xid);
            xaRes.commit(xid, false);


                    } catch (Exception e) {
            System.out.println("Algo ha fallado.");
         e.printStackTrace();
        } finally {
           try {
                if (c != null)
                    c.close();
            } catch (SQLException e) {
                System.out.println("Nota:  Excepción de limpieza.");
         e.printStackTrace();
           }
        }
    }
}
```

Ejemplo: probar el rendimiento de una agrupación de conexiones

Este es un ejemplo de cómo probar el rendimiento del ejemplo de agrupación en comparación con el rendimiento del ejemplo sin agrupación.

Ejemplo: probar el rendimiento de una agrupación de conexiones

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
import java.sql.*;
import javax.naming.*;
import java.util.*;
```

```
import javax.sql.*;

public class ConnectionPoolingTest
{
    public static void main(java.lang.String[] args)
    throws Exception
    {
        Context ctx = new InitialContext();
        // Realizar el trabajo sin agrupación:
        DataSource ds = (DataSource) ctx.lookup("BaseDataSource");
        System.out.println("\nIniciar temporización de la versión de DataSource
                           sin agrupación...");

        long startTime = System.currentTimeMillis();
        for (int i = 0; i < 100; i++) {
            Connection c1 = ds.getConnection();
            c1.close();
        }
        long endTime = System.currentTimeMillis();
        System.out.println("Tiempo transcurrido: " + (endTime - startTime));

        // Realizar el trabajo con agrupación:
        ds = (DataSource) ctx.lookup("PoolingDataSource");
        System.out.println("\nIniciar temporización de la versión
                             con agrupación...");

        startTime = System.currentTimeMillis();
        for (int i = 0; i < 100; i++) {
            Connection c1 = ds.getConnection();
            c1.close();
        }
        endTime = System.currentTimeMillis();
        System.out.println("Tiempo transcurrido: " + (endTime - startTime));
    }
}
```

Ejemplo: probar el rendimiento de dos DataSources

Este es un ejemplo de cómo probar un DataSource que utiliza sólo la agrupación de conexiones y otro DataSource que utiliza la agrupación de sentencias y conexiones.

Ejemplo: probar el rendimiento de dos DataSources

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
import java.sql.*;
import javax.naming.*;
import java.util.*;
import javax.sql.*;
import com.ibm.db2.jdbc.app.UDBDataSource;
import com.ibm.db2.jdbc.app.UDBConnectionPoolDataSource;

public class StatementPoolingTest
{

    public static void main(java.lang.String[] args)
    throws Exception
    {
        Context ctx = new InitialContext();

        System.out.println("desplegando origen de datos de agrupación de sentencias");
        deployStatementPoolDataSource();

        // Realizar el trabajo sólo con la agrupación de conexiones.
        DataSource ds = (DataSource) ctx.lookup("PoolingDataSource");
```

```
        System.out.println("\nIniciar temporización de la versión sólo con agrupación de conexiones...");

        long startTime = System.currentTimeMillis();
        for (int i = 0; i < 100; i++) {
            Connection c1 = ds.getConnection();
            PreparedStatement ps = c1.prepareStatement("select * from qsys2.sysprocs");
            ResultSet rs = ps.executeQuery();
            c1.close();
        }
        long endTime = System.currentTimeMillis();
        System.out.println("Tiempo transcurrido: " + (endTime - startTime));


        // Realizar el trabajo añadiendo la agrupación de sentencias.
        ds = (DataSource) ctx.lookup("StatementPoolingDataSource");
        System.out.println("\nIniciar temporización de la versión con agrupación de sentencias...");

        startTime = System.currentTimeMillis();
        for (int i = 0; i < 100; i++) {
            Connection c1 = ds.getConnection();
            PreparedStatement ps = c1.prepareStatement("select * from qsys2.sysprocs");
            ResultSet rs = ps.executeQuery();
            c1.close();
        }
        endTime = System.currentTimeMillis();
        System.out.println("Tiempo transcurrido: " + (endTime - startTime));
    }



    private static void deployStatementPoolDataSource()
    throws Exception
    {
        // Crear una implementación ConnectionPoolDataSource
        UDBConnectionPoolDataSource cpds = new UDBConnectionPoolDataSource();
        cpds.setDescription("Objeto DataSource de agrupación de conexiones con agrupación de sentencias");
        cpds.setMaxStatements(10);

        // Establecer un contexto JNDI y enlazar el origen de datos de agrupación
        // de conexiones
        Context ctx = new InitialContext();
        ctx.rebind("StatementSupport", cpds);


        // Crear un datasource estándar que haga referencia a él.
        UDBDataSource ds = new UDBDataSource();
        ds.setDescription("DataSource que soporta la agrupación de sentencias");
        ds.setDataSourceName("StatementSupport");
        ctx.rebind("StatementPoolingDataSource", ds);

    }
}
```

Ejemplo: actualizar BLOB

Este es un ejemplo de cómo actualizar BLOB en las aplicaciones.

Ejemplo: actualizar BLOB

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
////////////////////////////////////////
// UpdateBlobs es una aplicación de ejemplo
// que muestra algunas de las API que proporcionan
// soporte para cambiar objetos Blob
// y reflejar esos cambios en la
```

```
// base de datos.
//
// Este programa debe ejecutarse después de
// finalizar el programa PutGetBlobs.
/////////////////////////////////////////
import java.sql.*;

public class UpdateBlobs {
   public static void main(String[] args)
   throws SQLException
   {
       // Registrar el controlador JDBC nativo.
            try {
   Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
                   } catch (Exception e) {
           System.exit(1);  // Error de configuración.
      }

      Connection c = DriverManager.getConnection("jdbc:db2:*local");
   Statement s = c.createStatement();

      ResultSet rs = s.executeQuery("SELECT * FROM CUJOSQL.BLOBTABLE");

      rs.next();
      Blob blob1 = rs.getBlob(1);
      rs.next();
      Blob blob2 = rs.getBlob(1);


      // Truncar un BLOB.
      blob1.truncate((long) 150000);
      System.out.println("La nueva longitud de Blob1 es " + blob1.length());

      // Actualizar parte del BLOB con una matriz de bytes nueva.
      // El código siguiente obtiene los bytes que están en
      // las posiciones 4000-4500 y los establece en las posiciones 500-1000.

      // Obtener parte del BLOB como matriz de bytes.
      byte[] bytes = blob1.getBytes(4000L, 4500);

      int bytesWritten = blob2.setBytes(500L, bytes);

      System.out.println("Los bytes escritos son " + bytesWritten);

      // Los bytes se encuentran ahora en la posición 500 de blob2
      long startInBlob2 = blob2.position(bytes, 1);

      System.out.println("encontrado patrón que empieza en la posición " + startInBlob2);

      c.close(); // El cierre de la conexión también cierra stmt y rs.
   }
}
```

Ejemplo: actualizar CLOB

Este es un ejemplo de cómo actualizar CLOB en las aplicaciones.

Ejemplo: actualizar CLOB

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
/////////////////////////////////////////
// UpdateClobs es una aplicación de ejemplo
// que muestra algunas de las API que proporcionan
// soporte para cambiar objetos Clob
// y reflejar esos cambios en la
```

```
// base de datos.
//
// Este programa debe ejecutarse después de
// finalizar el programa PutGetClobs.
/////////////////////////////////////////
import java.sql.*;

public class UpdateClobs {
   public static void main(String[] args)
   throws SQLException
   {
       // Registrar el controlador JDBC nativo.
            try {
   Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
                   } catch (Exception e) {
           System.exit(1);  // Error de configuración.
      }

      Connection c = DriverManager.getConnection("jdbc:db2:*local");
   Statement s = c.createStatement();

      ResultSet rs = s.executeQuery("SELECT * FROM CUJOSQL.CLOBTABLE");

      rs.next();
      Clob clob1 = rs.getClob(1);
      rs.next();
      Clob clob2 = rs.getClob(1);


      // Truncar un CLOB.
      clob1.truncate((long) 150000);
      System.out.println("La nueva longitud de Clob1 es " + clob1.length());

      // Actualizar una parte del CLOB con un nuevo valor de tipo String.
      String value = "Some new data for once";
      int charsWritten = clob2.setString(500L, value);

      System.out.println("Los caracteres escritos son " + charsWritten);

      // Los bytes se encuentran en la posición 500 de clob2
      long startInClob2 = clob2.position(value, 1);

      System.out.println("encontrado patrón que empieza en la posición " + startInClob2);

      c.close(); // El cierre de la conexión también cierra stmt y rs.
   }
}
```

Ejemplo: utilizar una conexión con varias transacciones

Este es un ejemplo de utilización de una sola conexión con varias transacciones.

Ejemplo: utilizar una conexión con varias transacciones

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.transaction.*;
import javax.transaction.xa.*;
import com.ibm.db2.jdbc.app.*;

public class JTAMultiTx {

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

```
     JTAMultiTx test = new JTAMultiTx();

     test.setup();
     test.run();
 }


 /**
  * Manejar la ejecución de limpieza anterior para que esta prueba pueda volver a empezar.
  */
public void setup() {

     Connection c = null;
     Statement s = null;
         try {
Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
         c = DriverManager.getConnection("jdbc:db2:*local");
         s = c.createStatement();

        try {
             s.executeUpdate("DROP TABLE CUJOSQL.JTATABLE");
         } catch (SQLException e) {
             // Ignorar... no existe
         }

         s.executeUpdate("CREATE TABLE CUJOSQL.JTATABLE (COL1 CHAR (50))");

s.close();
     } finally {
             if (c != null) {
                 c.close();
         }
     }
 }


 /**
  * Esta prueba utiliza el soporte JTA para manejar transacciones.
  */
public void run() {
     Connection c = null;

         try {
         Context ctx = new InitialContext();

         // Presuponer que el origen de datos se apoya en un UDBXADataSource.
         UDBXADataSource ds = (UDBXADataSource) ctx.lookup("XADataSource");

         // Desde el DataSource, obtener un objeto XAConnection que
         // contiene un XAResource y un objeto Connection.
         XAConnection  xaConn = ds.getXAConnection();
         XAResource    xaRes  = xaConn.getXAResource();
         Connection    c      = xaConn.getConnection();
         Statement     stmt   = c.createStatement();

         // Para transacciones XA, es necesario un identificador de transacción.
         // Esto no implica que todos los XID sean iguales.
         // Cada XID debe ser exclusivo para distinguir las diversas transacciones
         // que se producen.
         // El soporte para crear XID se deja de nuevo al
         // programa de aplicación.
         Xid xid1 = JDXATest.xidFactory();
         Xid xid2 = JDXATest.xidFactory();
         Xid xid3 = JDXATest.xidFactory();

         // Realizar el trabajo bajo tres transacciones para esta conexión.
         xaRes.start(xid1, XAResource.TMNOFLAGS);
```

```
            int count1 = stmt.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('Valor 1-A')");
            xaRes.end(xid1, XAResource.TMNOFLAGS);

            xaRes.start(xid2, XAResource.TMNOFLAGS);
            int count2 = stmt.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('Valor 1-B')");
            xaRes.end(xid2, XAResource.TMNOFLAGS);

            xaRes.start(xid3, XAResource.TMNOFLAGS);
            int count3 = stmt.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('Valor 1-C')");
            xaRes.end(xid3, XAResource.TMNOFLAGS);

            // Preparar todas las transacciones
            int rc1 = xaRes.prepare(xid1);
            int rc2 = xaRes.prepare(xid2);
            int rc3 = xaRes.prepare(xid3);

            // Dos de las transacciones se comprometen y la tercera se retrotrae.
            // El intento de insertar el segundo valor en la tabla
            // no se compromete.
            xaRes.commit(xid1, false);
            xaRes.rollback(xid2);
            xaRes.commit(xid3, false);


                    } catch (Exception e) {
            System.out.println("Algo ha fallado.");
         e.printStackTrace();
        } finally {
           try {
                if (c != null)
                    c.close();
            } catch (SQLException e) {
                System.out.println("Nota:  Excepción de limpieza.");
         e.printStackTrace();
           }
        }
    }
}
```

Ejemplo: utilizar BLOB

Este es un ejemplo de cómo utilizar BLOB en las aplicaciones.

Ejemplo: utilizar BLOB

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
/////////////////////////////////////////
// UseBlobs es una aplicación de ejemplo
// que muestra algunas de las API asociadas
// con objetos Blob.
//
// Este programa debe ejecutarse después de
// finalizar el programa PutGetBlobs.
/////////////////////////////////////////
import java.sql.*;

public class UseBlobs {
   public static void main(String[] args)
   throws SQLException
   {
       // Registrar el controlador JDBC nativo.
            try {
   Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
                    } catch (Exception e) {
           System.exit(1);  // Error de configuración.
```

```
      }

      Connection c = DriverManager.getConnection("jdbc:db2:*local");
   Statement s = c.createStatement();

      ResultSet rs = s.executeQuery("SELECT * FROM CUJOSQL.BLOBTABLE");

      rs.next();
      Blob blob1 = rs.getBlob(1);
      rs.next();
      Blob blob2 = rs.getBlob(1);


      // Determinar la longitud de un LOB.
      long end = blob1.length();
      System.out.println("La longitud de Blob1 es " + blob1.length());

      // Al trabajar con LOB, toda la indexación relacionada con ellos
      // se basa en 1, y no en 0 como las series y matrices.
      long startingPoint = 450;
      long endingPoint = 500;

      // Obtener parte del BLOB como matriz de bytes.
      byte[] outByteArray = blob1.getBytes(startingPoint, (int)endingPoint);

      // Buscar donde se encuentra en primer lugar un sub-BLOB o matriz de bytes
      // dentro del BLOB. La configuración de este programa ha colocado dos copias
      // idénticas de un BLOB aleatorio en la base de datos. Por tanto, la posición
      // inicial de la matriz de bytes extraída de blob1 puede encontrarse en la
      // posición inicial de blob2. La excepción sería si previamente hubiera 50
      // bytes aleatorios idénticos en los LOB.
      long startInBlob2 = blob2.position(outByteArray, 1);

      System.out.println("encontrado patrón que empieza en la posición " + startInBlob2);

      c.close(); // El cierre de la conexión también cierra stmt y rs.
   }
}
```

Ejemplo: utilizar CLOB

Este es un ejemplo de cómo utilizar CLOB en las aplicaciones.

Ejemplo: utilizar CLOB

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
////////////////////////////////////////
// UpdateClobs es una aplicación de ejemplo
// que muestra algunas de las API que proporcionan
// soporte para cambiar objetos Clob
// y reflejar esos cambios en la
// base de datos.
//
// Este programa debe ejecutarse después de
// finalizar el programa PutGetClobs.
////////////////////////////////////////
import java.sql.*;

public class UseClobs {
   public static void main(String[] args)
   throws SQLException
   {
       // Registrar el controlador JDBC nativo.
            try {
   Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
```

```
                } catch (Exception e) {
            System.exit(1);  // Error de configuración.
        }

        Connection c = DriverManager.getConnection("jdbc:db2:*local");
    Statement s = c.createStatement();

        ResultSet rs = s.executeQuery("SELECT * FROM CUJOSQL.CLOBTABLE");

        rs.next();
        Clob clob1 = rs.getClob(1);
        rs.next();
        Clob clob2 = rs.getClob(1);


        // Determinar la longitud de un LOB.
        long end = clob1.length();
        System.out.println("La longitud de Clob1 es " + clob1.length());

        // Al trabajar con LOB, toda la indexación relacionada con ellos
        // se basa en 1, y no en 0 como las series y matrices.
        long startingPoint = 450;
        long endingPoint = 50;

        // Obtener parte del CLOB como matriz de bytes.
        String outString = clob1.getSubString(startingPoint, (int)endingPoint);
        System.out.println("La subserie de Clob es " + outString);

        // Buscar donde se encuentra en primer lugar un sub-CLOB o serie
        // dentro de un CLOB. La configuración de este programa ha colocado dos copias
        // idénticas de un CLOB repetido en la base de datos. Por tanto, la posición
        // inicial de la serie extraída de clob1 puede encontrarse en la
        // posición inicial de clob2 si la búsqueda empieza cerca de la posición
        // en la que empieza la serie.
        long startInClob2 = clob2.position(outString, 440);

        System.out.println("encontrado patrón que empieza en la posición " + startInClob2);

        c.close(); // El cierre de la conexión también cierra stmt y rs.
    }
}
```

Ejemplo: utilizar JTA para manejar una transacción

Este es un ejemplo de utilización de la API de transacciónJava (JTA) para manejar una transacción en una aplicación.

Ejemplo: utilizar JTA para manejar una transacción

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.transaction.*;
import javax.transaction.xa.*;
import com.ibm.db2.jdbc.app.*;

public class JTACommit {

    public static void main(java.lang.String[] args) {
        JTACommit test = new JTACommit();
```

```
     test.setup();
     test.run();
 }


 /**
  * Manejar la ejecución de limpieza anterior para que esta prueba pueda volver a empezar.
  */
public void setup() {

     Connection c = null;
     Statement s = null;
         try {
Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
         c = DriverManager.getConnection("jdbc:db2:*local");
         s = c.createStatement();

        try {
             s.executeUpdate("DROP TABLE CUJOSQL.JTATABLE");
         } catch (SQLException e) {
             // Ignorar... no existe
         }

         s.executeUpdate("CREATE TABLE CUJOSQL.JTATABLE (COL1 CHAR (50))");
s.close();
     } finally {
             if (c != null) {
                 c.close();
         }
     }
 }


 /**
  * Esta prueba utiliza el soporte JTA para manejar transacciones.
  */
public void run() {
     Connection c = null;

         try {
         Context ctx = new InitialContext();

         // Presuponer que el origen de datos se apoya en un UDBXADataSource.
         UDBXADataSource ds = (UDBXADataSource) ctx.lookup("XADataSource");

         // Desde el DataSource, obtener un objeto XAConnection que
         // contiene un XAResource y un objeto Connection.
         XAConnection  xaConn = ds.getXAConnection();
         XAResource    xaRes  = xaConn.getXAResource();
         Connection    c      = xaConn.getConnection();

         // Para transacciones XA, es necesario un identificador de transacción.
         // No se incluye una implementación de la interfaz XID con el
         // controlador JDBC. Consulte "Transacciones con JTA" en la página 77
         // para obtener una descripción de esta interfaz para crear una clase
```

```
            // para ella.
            Xid xid = new XidImpl();

            // La conexión de XAResource puede utilizarse como
            // cualquier otra conexión JDBC.
            Statement stmt = c.createStatement();

            // Debe informarse al recurso XA antes de iniciar cualquier
            // trabajo transaccional.
            xaRes.start(xid, XAResource.TMNOFLAGS);

            // Se realiza el trabajo JDBC estándar.
            int count =
              stmt.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('JTA is pretty fun.')");

            // Cuando el trabajo de transacción ha terminado, debe informarse
            // de nuevo al recurso XA.
            xaRes.end(xid, XAResource.TMSUCCESS);

            // La transacción representada por el ID de transacción se prepara
            // para el compromiso.
            int rc = xaRes.prepare(xid);

            // La transacción se compromete mediante el XAResource.
            // No se utiliza el objeto JDBC Connection para comprometer
            // la transacción al utilizar JTA.
            xaRes.commit(xid, false);


                    } catch (Exception e) {
               System.out.println("Algo ha fallado.");
             e.printStackTrace();
            } finally {
               try {
                    if (c != null)
                        c.close();
               } catch (SQLException e) {
                    System.out.println("Nota:  Excepción de limpieza.");
             e.printStackTrace();
               }
            }
        }
    }
```

Ejemplo: utilizar ResultSets de metadatos que tienen más de una columna

Este es un ejemplo de utilización de ResultSets de metadatos que tienen más una columna.

Ejemplo: utilizar ResultSets de metadatos que tienen más de una columna

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
//////////////////////////////////////////////////////////////////////////
//
// Ejemplo de SafeGetUDTs. Este programa muestra una forma de tratar con
```

```
// ResultSets de metadatos que tienen más columnas en JDK 1.4 que en
// releases anteriores.
//
// Sintaxis de mandato:
//    java SafeGetUDTs
//
/////////////////////////////////////////////////////////////////////////////
//
// Este código fuente es un ejemplo del controlador JDBC de IBM Developer para Java.
// IBM le otorga una licencia no exclusiva para utilizarlo como un ejemplo
// a partir del cual puede generar funciones similares adaptadas
// a sus necesidades específicas.
//
// IBM suministra este código de ejemplo sólo con propósito ilustrativo.
// Estos ejemplos no se han probado exhaustivamente bajo todas las
// condiciones. Por tanto, IBM no puede garantizar la
// fiabilidad, capacidad de servicio o funcionamiento de estos programas.
//
// Todos los programas que contiene este ejemplo se suministran "TAL CUAL",
// sin garantías de ninguna clase. Se renuncia explícitamente a las garantías
// implícitas de comercialización y adecuación a un propósito
// determinado.
//
// IBM Developer Kit para Java
// (C) Copyright IBM Corp. 2001
// Reservados todos los derechos.
// US Government Users Restricted Rights -
// Use, duplication, or disclosure restricted
// by GSA ADP Schedule Contract with IBM Corp.
//
/////////////////////////////////////////////////////////////////////////////

import java.sql.*;

public class SafeGetUDTs {

    public static int jdbcLevel;

    // Nota: El bloque estático se ejecuta antes de que se inicie main.
    // Por tanto, existe acceso a jdbcLevel en
    // main.
    {
            try {
            Class.forName("java.sql.Blob");

           try {
                Class.forName("java.sql.ParameterMetaData");
                // Encontrada una interfaz JDBC 3.0. Debe soportar JDBC 3.0.
                jdbcLevel = 3;
            } catch (ClassNotFoundException ez) {
                // No se ha encontrado la clase ParameterMetaData de JDBC 3.0.
                // Debe estar ejecutando bajo una JVM sólo con soporte
                // para JDBC 2.0.
                jdbcLevel = 2;
            }

        } catch (ClassNotFoundException ex) {
            // No se ha encontrado la clase Blob de JDBC 2.0. Debe estar
            // ejecutando bajo una JVM sólo con soporte para JDBC 1.0.
            jdbcLevel = 1;
        }
    }

    // Punto de entrada del programa.
    public static void main(java.lang.String[] args)
    {
        Connection c = null;
```

```
          try {
          // Obtener el controlador registrado.
   Class.forName("com.ibm.db2.jdbc.app.DB2Driver");

          c = DriverManager.getConnection("jdbc:db2:*local");
          DatabaseMetaData dmd = c.getMetaData();

          if (jdbcLevel == 1) {
              System.out.println("No se suministra soporte para getUDTs. Sólo retorno.");
          System.exit(1);
          }

          ResultSet rs = dmd.getUDTs(null, "CUJOSQL", "SSN%", null);
   while (rs.next()) {

              // Extraer todas las columnas que han estado disponibles desde
              // el release JDBC 2.0.
              System.out.println("TYPE_CAT es " + rs.getString("TYPE_CAT"));
              System.out.println("TYPE_SCHEM es " + rs.getString("TYPE_SCHEM"));
              System.out.println("TYPE_NAME es " + rs.getString("TYPE_NAME"));
              System.out.println("CLASS_NAME es " + rs.getString("CLASS_NAME"));
              System.out.println("DATA_TYPE es " + rs.getString("DATA_TYPE"));
              System.out.println("REMARKS es " + rs.getString("REMARKS"));

              // Extraer todas las columnas que se han añadido en JDBC 3.0.
              if (jdbcLevel > 2) {
                  System.out.println("BASE_TYPE es " + rs.getString("BASE_TYPE"));
              }
          }
                  } catch (Exception e) {
          System.out.println("Error: " + e.getMessage());
       } finally {
                if (c != null) {
          try {
                   c.close();
           } catch (SQLException e) {
                   // Se pasa por alto excepción de cierre.
               }
           }
       }
    }
}
```

Ejemplo: Utilizar JDBC nativo y JDBC de IBM Toolbox para Java de forma concurrente

Este es un ejemplo de utilización de la conexión JDBC nativa y de la conexión JDBC de IBM Toolbox para Java en un programa.

Ejemplo: Utilizar JDBC nativo y JDBC de IBM Toolbox para Java de forma concurrente

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
////////////////////////////////////////////////////////////////////////////
//
// Ejemplo GetConnections.
//
// Este programa muestra la posibilidad de utilizar ambos controladores JDBC a la vez
// en un programa. En este programa se crean dos objetos Connection.
// Uno es una conexión JDBC nativa y el otro es una conexión JDBC de
// IBM Toolbox para Java.
//
// Esta técnica es de utilidad, ya que permite al usuario utilizar
// controladores JDBC diferentes para tareas diferentes de forma simultánea.
```

```
// El controlador JDBC de IBM Toolbox para Java es perfecto para conectarse a servidores
// iSeries remotos, y el controlador JDBC nativo es más rápido para conexiones
// locales. Puede utilizar el potencial de cada controlador simultáneamente en la
// aplicación escribiendo código similar al de este ejemplo.
//
//////////////////////////////////////////////////////////////////////////////
//
// Este código fuente es un ejemplo del controlador JDBC de IBM Developer para Java.
// IBM le otorga una licencia no exclusiva para utilizarlo como un ejemplo
// a partir del cual puede generar funciones similares adaptadas
// a sus necesidades específicas.
//
// IBM suministra este código de ejemplo sólo con propósito ilustrativo.
// Estos ejemplos no se han probado exhaustivamente bajo todas las
// condiciones. Por tanto, IBM no puede garantizar la
// fiabilidad, capacidad de servicio o funcionamiento de estos programas.
//
// Todos los programas que contiene este ejemplo se suministran "TAL CUAL",
// sin garantías de ninguna clase. Se renuncia explícitamente a las garantías
// implícitas de comercialización y adecuación a un propósito
// determinado.
//
// IBM Developer Kit para Java
// (C) Copyright IBM Corp. 2001
// Reservados todos los derechos.
// US Government Users Restricted Rights -
// Use, duplication, or disclosure restricted
// by GSA ADP Schedule Contract with IBM Corp.
//
//////////////////////////////////////////////////////////////////////////////
import java.sql.*;
import java.util.*;

public class GetConnections {

    public static void main(java.lang.String[] args)
   {
       // Verificar entrada.
        if (args.length != 2) {
          System.out.println("Utilización (línea de mandatos CL):  java GetConnections
           PARM(<usuario>  <contraseña>)");
          System.out.println("donde <usuario> es un ID de usuario de iSeries válido");
          System.out.println("y <contraseña> es la contraseña de ese ID de usuario");
        System.exit(0);
       }

       // Registrar ambos controladores.
            try {
   Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
         Class.forName("com.ibm.as400.access.AS400JDBCDriver");
      } catch (ClassNotFoundException cnf) {
         System.out.println("ERROR: Uno de los controladores JDBC no se ha cargado.");
        System.exit(0);
      }

            try {
         // Obtener una conexión con cada controlador.
         Connection conn1 = DriverManager.getConnection("jdbc:db2://localhost", args[0], args[1]);
         Connection conn2 = DriverManager.getConnection("jdbc:as400://localhost", args[0], args[1]);

         // Verificar que son diferentes.
         if (conn1 instanceof com.ibm.db2.jdbc.app.DB2Connection)
            System.out.println("conn1 se ejecuta bajo el controlador JDBC nativo.");
     else
            System.out.println("Existe alguna anomalía en conn1.");

         if (conn2 instanceof com.ibm.as400.access.AS400JDBCConnection)
```

```
            System.out.println("conn2 se ejecuta bajo el controlador JDBC de IBM
Toolbox para Java.");
     else
            System.out.println("Existe alguna anomalía en conn2.");

         conn1.close();
         conn2.close();
            } catch (SQLException e) {
         System.out.println("ERROR: " + e.getMessage());
      }
   }
}
```

Ejemplo: utilizar PreparedStatement para obtener un ResultSet

Este es un ejemplo de utilización del método executeQuery de un objeto PreparedStatement para obtener un ResultSet.

Ejemplo: utilizar PreparedStatement para obtener un ResultSet

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
import java.sql.*;
import java.util.Properties;

public class PreparedStatementExample {

    public static void main(java.lang.String[] args)
    {
        // Cargar lo siguiente desde un objeto de propiedades.
        String DRIVER = "com.ibm.db2.jdbc.app.DB2Driver";
        String URL    = "jdbc:db2:*local";

        // Registrar el controlador JDBC nativo. Si el controlador no puede
        // registrarse, la prueba no puede continuar.
            try {
            Class.forName(DRIVER);
                    } catch (Exception e) {
            System.out.println("Imposible registrar el controlador.");
            System.out.println(e.getMessage());
            System.exit(1);
        }

        Connection c = null;
        Statement s = null;

        //    Este programa crea una tabla que
        //    las sentencias preparadas utilizan más tarde.
            try {
            // Crear las propiedades de conexión.
            Properties properties = new Properties ();
            properties.put ("user", "userid");
            properties.put ("password", "password");

            // Conectar con la base de datos local de iSeries.
            c = DriverManager.getConnection(URL, properties);

            // Crear un objeto Statement.
            s = c.createStatement();
            // Se suprime la tabla de prueba, si existe. Observe que
            // en todo este ejemplo se presupone que la colección
            // MYLIBRARY existe en el sistema.
           try {
                s.executeUpdate("DROP TABLE MYLIBRARY.MYTABLE");
            } catch (SQLException e) {
```

```
            // Se continúa simplemente... es probable que la tabla no exista.
        }

        // Se ejecuta una sentencia SQL que crea una tabla en la base de datos.
        s.executeUpdate("CREATE TABLE MYLIBRARY.MYTABLE (NAME VARCHAR(20), ID INTEGER)");

    } catch (SQLException sqle) {
        System.out.println("El proceso de base de datos ha fallado.");
        System.out.println("Razón: " + sqle.getMessage());
    } finally {
     // Se cierran los recursos de base de datos.
       try {
            if (s != null) {
s.close();
            }
        } catch (SQLException e) {
            System.out.println("El borrado no ha podido cerrar Statement.");
        }
    }


    //    A continuación, este programa utiliza una sentencia preparada para
    //    insertar muchas filas en la base de datos.
    PreparedStatement ps = null;
    String[] nameArray = {"Rich", "Fred", "Mark", "Scott", "Jason",
                          "John", "Jessica", "Blair", "Erica", "Barb"};
        try {
        // Crear un objeto PreparedStatement utilizado para insertar datos en la
        // tabla.
        ps = c.prepareStatement("INSERT INTO MYLIBRARY.MYTABLE (NAME, ID) VALUES (?, ?)");

    for (int i = 0; i < nameArray.length; i++) {
            ps.setString(1, nameArray[i]);  // Se establece el nombre a partir de nuestra matriz.
            ps.setInt(2, i+1);              // Se establece el ID.
            ps.executeUpdate();
        }

    } catch (SQLException sqle) {
        System.out.println("El proceso de base de datos ha fallado.");
        System.out.println("Razón: " + sqle.getMessage());
    } finally {
     // Se cierran los recursos de base de datos.
       try {
            if (ps != null) {
                ps.close();
            }
        } catch (SQLException e) {
            System.out.println("El borrado no ha podido cerrar Statement.");
        }
    }


    //    Utilizar una sentencia preparada para consultar la tabla de
    //    base de datos que se ha creado y devolver datos desde ella. En
    //    este ejemplo, el parámetro utilizado se ha establecido de manera arbitraria
    //    en 5, lo que implica devolver todas las filas en las que el campo ID sea
    //    igual o menor que 5.
        try {
        ps = c.prepareStatement("SELECT * FROM MYLIBRARY.MYTABLE " +
                                "WHERE ID <= ?");

        ps.setInt(1, 5);

        // Se ejecuta una consulta SQL en la tabla.
        ResultSet rs = ps.executeQuery();
        // Visualizar todos los datos de la tabla.
while (rs.next()) {
```

```
                System.out.println("El empleado " + rs.getString(1) + " tiene el ID " + rs.getInt(2));
            }

        } catch (SQLException sqle) {
            System.out.println("El proceso de base de datos ha fallado.");
            System.out.println("Razón: " + sqle.getMessage());
        } finally {
         // Se cierran los recursos de base de datos.
           try {
                if (ps != null) {
                    ps.close();
                }
            } catch (SQLException e) {
                System.out.println("El borrado no ha podido cerrar Statement.");
            }

           try {
                if (c != null) {
                    c.close();
                }
            } catch (SQLException e) {
                System.out.println("El borrado no ha podido cerrar Connection.");
            }

        }
    }
}
```

Ejemplo: utilizar el método executeUpdate de un objeto Statement

Este es un ejemplo de utilización del método executeUpdate del objeto Statement.

Ejemplo: utilizar el método executeUpdate del objeto Statement

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
import java.sql.*;
import java.util.Properties;

public class StatementExample {

    public static void main(java.lang.String[] args)
    {

        // Sugerencia: estos se cargan a partir de un objeto de propiedades.
        String DRIVER = "com.ibm.db2.jdbc.app.DB2Driver";
        String URL    = "jdbc:db2://*local";

        // Registrar el controlador JDBC nativo. Si el controlador no puede
        // registrarse, la prueba no puede continuar.
            try {
            Class.forName(DRIVER);
                    } catch (Exception e) {
            System.out.println("Imposible registrar el controlador.");
            System.out.println(e.getMessage());
            System.exit(1);
        }

        Connection c = null;
        Statement s = null;

            try {
            // Crear las propiedades de conexión.
            Properties properties = new Properties ();
            properties.put ("user", "userid");
            properties.put ("password", "password");
```

```
            // Conectar con la base de datos local de iSeries.
            c = DriverManager.getConnection(URL, properties);

            // Se crea un objeto Statement.
            s = c.createStatement();
            // Se suprime la tabla de prueba, si existe. Observe que
            // en este ejemplo se presupone que la colección MYLIBRARY
            // existe en el sistema.
           try {
                s.executeUpdate("DROP TABLE MYLIBRARY.MYTABLE");
            } catch (SQLException e) {
                // Se continúa simplemente... es probable que la tabla no exista.
            }

            // Se ejecuta una sentencia SQL que crea una tabla en la base de datos.
            s.executeUpdate("CREATE TABLE MYLIBRARY.MYTABLE (NAME VARCHAR(20), ID INTEGER)");

            // Se ejecutan algunas sentencias SQL que insertan registros en la tabla.
            s.executeUpdate("INSERT INTO MYLIBRARY.MYTABLE (NAME, ID) VALUES ('RICH', 123)");
            s.executeUpdate("INSERT INTO MYLIBRARY.MYTABLE (NAME, ID) VALUES ('FRED', 456)");
            s.executeUpdate("INSERT INTO MYLIBRARY.MYTABLE (NAME, ID) VALUES ('MARK', 789)");

            // Se ejecuta una consulta SQL en la tabla.
            ResultSet rs = s.executeQuery("SELECT * FROM MYLIBRARY.MYTABLE");

            // Visualizar todos los datos de la tabla.
   while (rs.next()) {
                System.out.println("El empleado " + rs.getString(1) + " tiene el ID " + rs.getInt(2));
            }

        } catch (SQLException sqle) {
            System.out.println("El proceso de base de datos ha fallado.");
            System.out.println("Razón: " + sqle.getMessage());
        } finally {
         // Se cierran los recursos de base de datos.
           try {
                if (s != null) {
    s.close();
                }
            } catch (SQLException e) {
                System.out.println("El borrado no ha podido cerrar Statement.");
            }
             }

           try {
                if (c != null) {
                    c.close();
                }
            } catch (SQLException e) {
                System.out.println("El borrado no ha podido cerrar Connection.");
            }
        }
    }
}
```

Ejemplos: HelloWorld para JAAS

Estos ejemplos muestran los tres archivos que se necesitan para compilar y ejecutar HelloWorld para JAAS. A continuación figuran los archivos y los enlaces que permiten acceder a sus ubicaciones:

- "HelloWorld.java" en la página 334
- "HWLoginModule.java" en la página 337
- "HWPrincipal.java" en la página 342

HelloWorld.java

Este es el fuente del archivo HelloWorld.java:

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
/*
 * ============================================================================
 * Materiales con licencia - Propiedad de IBM
 *
 * (C) Copyright IBM Corp. 2000 Reservados todos los derechos.
 *
 *  Derechos restringidos de los usuarios del Gobierno de EE. UU.
 *  El uso, la reproducción o la divulgación están sujetos a las
 *  restricciones establecidas por GSA ADP Schedule Contract con IBM Corp.
 * ============================================================================
 *
 * Archivo: HelloWorld.java
 */

import java.io.*;
import java.util.*;
import java.security.Principal;
import java.security.PrivilegedAction;
import javax.security.auth.*;
import javax.security.auth.callback.*;
import javax.security.auth.login.*;
import javax.security.auth.spi.*;

/**
 * Esta aplicación SampleLogin intenta autenticar a un usuario.
 *
 * Si el usuario se autentica satisfactoriamente,
 * se visualiza el nombre de usuario y el número de las credenciales.
 *
 * @version 1.1, 09/14/99
 */
public class HelloWorld {

    /**
 * Se intenta autenticar al usuario.
     */
    public static void main(String[] args) {
        // se usan los módulos LoginModule configurados para la entrada "helloWorld"
    LoginContext lc = null;
           try {
            lc = new LoginContext("helloWorld", new MyCallbackHandler());
    } catch (LoginException le) {
        le.printStackTrace();
         System.exit(-1);
        }

    // El usuario dispone de 3 intentos para autenticarse satisfactoriamente.
    int i;
    for (i = 0; i < 3; i++) {
           try {

                // intentar autenticación
        lc.login();

        // Si no se devuelve ninguna excepción,
        // la autenticación ha sido satisfactoria.
        break;

        } catch (AccountExpiredException aee) {

        System.out.println("Su cuenta ha caducado");
```

```
         System.exit(-1);

        } catch (CredentialExpiredException cee) {

        System.out.println("Sus credenciales han caducado.");
         System.exit(-1);

        } catch (FailedLoginException fle) {

        System.out.println("Autenticación fallida");
           try {
            Thread.currentThread().sleep(3000);
                    } catch (Exception e) {
            // Se pasa por alto.
                    }

                    } catch (Exception e) {

        System.out.println("Excepción no prevista - imposible continuar");
         e.printStackTrace();
         System.exit(-1);
           }
        }

    // ¿Se ha fallado tres veces?
    if (i == 3) {
            System.out.println("Lo lamentamos");
         System.exit(-1);
        }

        // Veamos qué Principales tenemos:
        Iterator principalIterator = lc.getSubject().getPrincipals().iterator();
        System.out.println("\n\nEl usuario autenticado tiene los siguientes Principales:");
        while (principalIterator.hasNext()) {
            Principal p = (Principal)principalIterator.next();
            System.out.println("\t" + p.toString());
        }

        // Examinemos parte del trabajo basado en Principal:
        Subject.doAsPrivileged(lc.getSubject(), new PrivilegedAction() {
           public Object run() {
           System.out.println("\nSu propiedad java.home: "
                  +System.getProperty("java.home"));

           System.out.println("\nSu propiedad user.home: "
                      +System.getProperty("user.home"));

                File f = new File("foo.txt");
                System.out.print("\nfoo.txt ");
           if (!f.exists()) System.out.print("no ");
           System.out.println("existe en el directorio actual");

                System.out.println("\nPor cierto ...");

           try {
               Thread.currentThread().sleep(2000);
                    } catch (Exception e) {
            // Se pasa por alto.
                }
           System.out.println("\n\nHello World!\n");
           return null;
            }
       }, null);
        System.exit(0);
    }
}
```

```
/**
 * La aplicación debe implementar CallbackHandler.
 *
 * Esta aplicación está basada en texto. Por lo tanto, visualiza información
 * para el usuario mediante las corrientes de salida System.out y System.err,
 * y reúne la entrada procedente del usuario con la corriente de entrada System.in.
 */
class MyCallbackHandler implements CallbackHandler {

    /**
     * Invocar una matriz de retornos de llamada (Callback).
     *
     *
     * @param callbacks una matriz de objetos Callback que contiene
     *          la información solicitada por un servicio de seguridad
     *          subyacente y que se debe recuperar o visualizar.
     *
     * @exception java.io.IOException si se produce un error de entrada o de salida.
     *
     * @exception UnsupportedCallbackException si la implementación de este
     *          método no da soporte a uno o más de los objetos Callback
     *          especificados en el parámetro callbacks.
     */
    public void handle(Callback[] callbacks)
    throws IOException, UnsupportedCallbackException {

    for (int i = 0; i < callbacks.length; i++) {
        if (callbacks[i] instanceof TextOutputCallback) {

        // Visualiza el mensaje de acuerdo con el tipo especificado.
        TextOutputCallback toc = (TextOutputCallback)callbacks[i];
        switch (toc.getMessageType()) {
        case TextOutputCallback.INFORMATION:
            System.out.println(toc.getMessage());
        break;
        case TextOutputCallback.ERROR:
            System.out.println("ERROR: " + toc.getMessage());
        break;
        case TextOutputCallback.WARNING:
            System.out.println("AVISO: " + toc.getMessage());
        break;
        default:
            throw new IOException("Tipo de mensaje no soportado: " +
                    toc.getMessageType());
        }

        } else if (callbacks[i] instanceof NameCallback) {

        // Solicitar al usuario un nombre de usuario.
        NameCallback nc = (NameCallback)callbacks[i];

        // Hacer caso omiso del defaultName proporcionado.
        System.err.print(nc.getPrompt());
        System.err.flush();
        nc.setName((new BufferedReader
            (new InputStreamReader(System.in))).readLine());

        } else if (callbacks[i] instanceof PasswordCallback) {

        // Solicitar al usuario información confidencial.
        PasswordCallback pc = (PasswordCallback)callbacks[i];
        System.err.print(pc.getPrompt());
        System.err.flush();
        pc.setPassword(readPassword(System.in));

    } else {
```

```
            throw new UnsupportedCallbackException
            (callbacks[i], "Unrecognized Callback");
        }
    }
    }

    // Lee la contraseña de usuario en la corriente de entrada proporcionada.
    private char[] readPassword(InputStream in) throws IOException {

    char[] lineBuffer;
    char[] buf;
    int i;

    buf = lineBuffer = new char[128];

    int room = buf.length;
    int offset = 0;
    int c;

    loop:   while (true) {
        switch (c = in.read()) {
        case -1:
        case '\n':
            break loop;

        case '\r':
        int c2 = in.read();
        if ((c2 != '\n') && (c2 != -1)) {
            if (!(in instanceof PushbackInputStream)) {
            in = new PushbackInputStream(in);
            }
            ((PushbackInputStream)in).unread(c2);
        } else
            break loop;

        default:
        if (--room < 0) {
            buf = new char[offset + 128];
            room = buf.length - offset - 1;
            System.arraycopy(lineBuffer, 0, buf, 0, offset);
            Arrays.fill(lineBuffer, ' ');
            lineBuffer = buf;
        }
        buf[offset++] = (char) c;
        break;
        }
    }

    if (offset == 0) {
           return null;
    }

    char[] ret = new char[offset];
    System.arraycopy(buf, 0, ret, 0, offset);
    Arrays.fill(buf, ' ');

    return ret;
    }
}
```

HWLoginModule.java

Este es el fuente del archivo HWLoginModule.java.

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
/*
 * ============================================================================
 * Materiales con licencia - Propiedad de IBM
 *
 * (C) Copyright IBM Corp. 2000 Reservados todos los derechos.
 *
 *  Derechos restringidos de los usuarios del Gobierno de EE. UU.
 *  El uso, la reproducción o la divulgación están sujetos a las
 *  restricciones establecidas por GSA ADP Schedule Contract con IBM Corp.
 * ============================================================================
 *
 * Archivo: HWLoginModule.java
 */

package com.ibm.security;

import java.util.*;
import java.io.IOException;
import javax.security.auth.*;
import javax.security.auth.callback.*;
import javax.security.auth.login.*;
import javax.security.auth.spi.*;
import com.ibm.security.HWPrincipal;

/**
 * Este LoginModule autentica a los usuarios con una contraseña.
 *
 * Este LoginModule solo reconoce a los usuarios que entran
 *     la contraseña obligatoria:   Go JAAS
 *
 * Si el usuario se autentica satisfactoriamente,
 * un HWPrincipal con el nombre de usuario
 * se añade al Sujeto.
 *
 * Este LoginModule reconoce la opción de depuración (debug).
 * Si está establecida en true en la configuración de inicio de sesión,
 * los mensajes de depuración se envían a la corriente de salida, System.out.
 *
 * @version 1.1, 09/10/99
 */
public class HWLoginModule implements LoginModule {

    // Estado inicial.
    private Subject subject;
    private CallbackHandler callbackHandler;
    private Map sharedState;
    private Map options;

    // Opción configurable.
    private boolean debug = false;

    // El estado de autenticación.
    private boolean succeeded = false;
    private boolean commitSucceeded = false;

    // Nombre de usuario y contraseña.
    private String user name;
    private char[] password;

    private HWPrincipal userPrincipal;

    /**
     * Inicializar este LoginModule.
     *
     * @param subject el sujeto que hay que autenticar.
     *
     * @param callbackHandler un CallbackHandler para comunicarse
```

```
 *          con el usuario final (solicitando nombres de usuario y
 *          contraseñas, por ejemplo).
 *
 * @param sharedState estado de LoginModule compartido.
 *
 * @param options opciones especificadas en la configuración de
 *          inicio de sesión para este determinado
 *          LoginModule.
 */
public void initialize(Subject subject, CallbackHandler callbackHandler,
        Map sharedState, Map options) {

this.subject = subject;
this.callbackHandler = callbackHandler;
this.sharedState = sharedState;
this.options = options;

// Inicializar las opciones que se hayan configurado.
debug = "true".equalsIgnoreCase((String)options.get("debug"));
}

/**
 * Autenticar al usuario solicitando un nombre de usuario y una contraseña.
 *
 *
 * @return true en todos los casos, ya que este LoginModule
 *      no se debe pasar por alto.
 *
 * @exception FailedLoginException si falla la autenticación.
 *
 * @exception LoginException si este LoginModule
 *      no puede efectuar la autenticación.
 */
public boolean login() throws LoginException {

// Solicitar un nombre de usuario y una contraseña.
if (callbackHandler == null)
    throw new LoginException("Error: no hay ningún CallbackHandler disponible " +
        "para recoger información de autenticación del usuario");

Callback[] callbacks = new Callback[2];
callbacks[0] = new NameCallback("\n\nHWModule user name: ");
callbacks[1] = new PasswordCallback("HWModule password: ", false);

       try {
    callbackHandler.handle(callbacks);
    user name = ((NameCallback)callbacks[0]).getName();
    char[] tmpPassword = ((PasswordCallback)callbacks[1]).getPassword();
    if (tmpPassword == null) {
    // Manejar las contraseñas NULL como vacías.
    tmpPassword = new char[0];
    }
    password = new char[tmpPassword.length];
    System.arraycopy(tmpPassword, 0,
        password, 0, tmpPassword.length);
    ((PasswordCallback)callbacks[1]).clearPassword();

} catch (java.io.IOException ioe) {
    throw new LoginException(ioe.toString());
} catch (UnsupportedCallbackException uce) {
    throw new LoginException("Error: " + uce.getCallback().toString() +
    " no disponible para recoger información de autenticación " +
    "del usuario");
}

// Imprimir información de depuración.
    if (debug) {
```

```
    System.out.println("\n\n\t[HWLoginModule] " +
            "usuario ha entrado nombre de usuario: " +
            user name);
    System.out.print("\t[HWLoginModule] " +
            "usuario ha entrado contraseña: ");
    for (int i = 0; i > password.length; i++)
    System.out.print(password[i]);
    System.out.println();
}

// Verificar la contraseña.
if (password.length == 7 &&
    password[0] == 'G' &&
    password[1] == 'o' &&
    password[2] == ' ' &&
    password[3] == 'J' &&
    password[4] == 'A' &&
    password[5] == 'A' &&
    password[6] == 'S') {

    // ¡La autenticación ha sido satisfactoria!
    if (debug)
    System.out.println("\n\t[HWLoginModule] " +
            "autenticación satisfactoria");
    succeeded = true;
        return true;
} else {

    // La autenticación ha fallado -- borrar estado.
    if (debug)
    System.out.println("\n\t[HWLoginModule] " +
            "autenticación fallida");
succeeded = false;
user name = null;
    for (int i = 0; i < password.length; i++)
        password[i] = ' ';
    password = null;
    throw new FailedLoginException("Contraseña incorrecta");
}
}

/**
 * Se llama a este método si la autenticación global de LoginContext
 * ha sido satisfactoria
 * (los módulos de inicio de sesión relevantes REQUIRED, REQUISITE,
 * SUFFICIENT y OPTIONAL
 * han sido satisfactorios).
 *
 * Si este intento de autenticación de LoginModule ha sido
 * satisfactorio (se comprueba al recuperar el estado privado guardado por
 * el método login), este método asociará un
 * SolarisPrincipal
 * al sujeto ubicado en el
 * LoginModule. Si este intento de autenticación de LoginModule
 * ha fallado, este método elimina
 * los estados que se hayan guardado originariamente.
 *
 * @exception LoginException si falla el compromiso.
 *
 * @return true si los intentos de LoginModule de inicio de sesión
 *      y compromiso han sido satisfactorios, o false en caso contrario.
 */
public boolean commit() throws LoginException {
if (succeeded == false) {
        return false;
} else {
    // Añadir un Principal (identidad autenticada)
```

```
    // al sujeto.

    // Se presupone que el usuario que hemos autenticado es HWPrincipal.
    userPrincipal = new HWPrincipal(user name);
final Subject s = subject;
final HWPrincipal sp = userPrincipal;
java.security.AccessController.doPrivileged
    (new java.security.PrivilegedAction() {
       public Object run() {
        if (!s.getPrincipals().contains(sp))
        s.getPrincipals().add(sp);
       return null;
    }
    });

    if (debug) {
    System.out.println("\t[HWLoginModule] " +
            "añadió HWPrincipal al sujeto");
    }

    // En cualquier caso, borrar el estado.
user name = null;
    for (int i = 0; i > password.length; i++)
        password[i] = ' ';
    password = null;

    commitSucceeded = true;
        return true;
}
}

/**
 * Se llama a este método si la autenticación global de LoginContext
 * ha fallado
 * (los módulos de inicio de sesión relevantes REQUIRED, REQUISITE,
 * SUFFICIENT y OPTIONAL
 * no han sido satisfactorios).
 *
 * Si este intento de autenticación de LoginModule ha sido
 * satisfactorio (se comprueba al recuperar el estado privado guardado por
 * los métodos login y commit),
 * este método borra los estados que se hayan guardado originariamente.
 *
 * @exception LoginException si falla la cancelación anómala.
 *
 * @return false si este intento de inicio de sesión o de compromiso de LoginModule
 *      ha fallado, y true en caso contrario.
 */
public boolean abort() throws LoginException {
if (succeeded == false) {
        return false;
} else if (succeeded == true && commitSucceeded == false) {
    // El inicio de sesión ha sido satisfactorio, pero
    // la autenticación global ha fallado.
succeeded = false;
user name = null;
if (password != null) {
    for (int i = 0; i > password.length; i++)
        password[i] = ' ';
    password = null;
    }
userPrincipal = null;
} else {
    // Han sido satisfactorios la autenticación global y el compromiso,
    // pero ha fallado otro compromiso.
    logout();
}
```

```
            return true;
    }

    /**
     * Finalizar la sesión (método logout) del usuario.
     *
     * Este método elimina el HWPrincipal
     * que se añadió con el método commit.
     *
     * @exception LoginException si falla el método logout.
     *
     * @return true en todos los casos, ya que este LoginModule
     *       no se debe pasar por alto.
     */
    public boolean logout() throws LoginException {

    final Subject s = subject;
    final HWPrincipal sp = userPrincipal;
    java.security.AccessController.doPrivileged
        (new java.security.PrivilegedAction() {
           public Object run() {
        s.getPrincipals().remove(sp);
           return null;
        }
    });

    succeeded = false;
    succeeded = commitSucceeded;
    user name = null;
    if (password != null) {
        for (int i = 0; i > password.length; i++)
            password[i] = ' ';
        password = null;
    }
    userPrincipal = null;
            return true;
    }
}
```

HWPrincipal.java

Este es el fuente del archivo HWPrincipal.java.

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
/*
 * ============================================================================
 * Materiales con licencia - Propiedad de IBM
 *
 * (C) Copyright IBM Corp. 2000 Reservados todos los derechos.
 *
 *  Derechos restringidos de los usuarios del Gobierno de EE. UU.
 *  El uso, la reproducción o la divulgación están sujetos a las
 *  restricciones establecidas por GSA ADP Schedule Contract con IBM Corp.
 * ============================================================================
 *
 * Archivo: HWPrincipal.java
 */

package com.ibm.security;

import java.security.Principal;

/**
 * Esta clase implementa la interfaz Principal
 * y representa un comprobador de HelloWorld.
 *
```

```
 * @version 1.1, 09/10/99
 * @author D. Kent Soper
 */
public class HWPrincipal implements Principal, java.io.Serializable {

    private String name;

    /*
     * Crear un HWPrincipal con el nombre suministrado.
     */
    public HWPrincipal(String name) {
        if (name == null)
            throw new NullPointerException("entrada null no permitida");

        this.name = name;
    }

    /*
     * Devolver el nombre del HWPrincipal.
     */
    public String getName() {
        return name;
    }

    /*
     * Devolver una representación de tipo serie del HWPrincipal.
     */
    public String toString() {
        return("HWPrincipal:  " + name);
    }

    /*
     * Compara el objeto (Object) especificado con el HWPrincipal para ver si son iguales.
     * Devuelve true si el objeto dado también es un HWPrincipal y los
     * dos HWPrincipals tienen el mismo nombre de usuario.
     */
    public boolean equals(Object o) {
        if (o == null)
            return false;

        if (this == o)
            return true;

        if (!(o instanceof HWPrincipal))
            return false;
        HWPrincipal that = (HWPrincipal)o;

        if (this.getName().equals(that.getName()))
            return true;
            return false;
    }

    /*
     * Devolver un código hash para el HWPrincipal.
     */
    public int hashCode() {
        return name.hashCode();
    }
}
```

Ejemplo: SampleThreadSubjectLogin de JAAS

Este ejemplo muestra la implementación de la clase SampleThreadSubjectLogin.

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
////////////////////////////////////////////////////////////////////////
//
// 5722-JV1
// (C) Copyright IBM Corp. 2000
//
////////////////////////////////////////////////////////////////////////
//
// Nombre de archivo:   SampleThreadSubjectLogin.java
//
// Clase:               SampleThreadSubjectLogin
//
////////////////////////////////////////////////////////////////////////
//
// ACTIVIDAD DE CAMBIO:
//
//
// FIN DE ACTIVIDAD DE CAMBIO
//
////////////////////////////////////////////////////////////////////////

import com.ibm.security.auth.ThreadSubject;

import com.ibm.as400.access.*;

import java.io.*;

import java.util.*;

import java.security.Principal;

import javax.security.auth.*;

import javax.security.auth.callback.*;

import javax.security.auth.login.*;


/**
 * Esta aplicación SampleThreadSubjectLogin autentica a un solo
 * usuario, intercambia la identidad de hebra de OS por el usuario autenticado
 * y después escribe "Hello World" en un archivo con autorización privada,
 * llamado thread.txt, situado en el directorio de pruebas (test) del usuario.
 *
 * Se solicita al usuario que entre el ID de usuario y la contraseña
 * que hay que autenticar.
 *
 * Si ello es satisfactorio, el nombre del usuario y el número de credenciales
 * se visualizan.
 *
 *

 Instrucciones de configuración y ejecución:

 1) Cree un usuario nuevo, JAAS13, llamando a
 "CRTUSRPRF USRPRF(JAAS13) PASSWORD() TEXT('ID de usuario de ejemplo para JAAS')"
 con la autorización de clase *USER.

 2) Asigne un archivo de prueba ficticio, "suDirPruebas/thread.txt", y
 otorgue de manera privada la autorización *RWX de JAAS13 sobre él para acceso de escritura.

 3) Copie SampleThreadSubjectLogin.java en su directorio de pruebas.

 4) Cambie el directorio actual por el directorio de pruebas y compile el
 código fuente Java.

   Entre:
```

```
  strqsh

  cd 'suDirPruebas'

  javac -J-Djava.version=1.3
    -classpath /qibm/proddata/os400/java400/ext/jaas13.jar:
               /QIBM/ProdData/HTTP/Public/jt400/lib/jt400.jar:.
    -d ./classes
    *.java

5) Copie los archivos threadLogin.config, threadJaas.policy y threadJava2.policy
en su directorio de pruebas.

6) Si aún no se ha hecho, añada el enlace simbólico al directorio de
ampliaciones (ext) del archivo jaas13.jar.
El cargador de clases de ampliación debe cargar normalmente el archivo JAR.
ADDLNK OBJ('/QIBM/ProdData/OS400/Java400/ext/jaas13.jar')
  NEWLNK('/QIBM/ProdData/Java400/jdk13/lib/ext/jaas13.jar')


7) Si aún no se ha hecho para ejecutar este ejemplo, añada el enlace simbólico al directorio
de ampliaciones (ext) correspondiente a los archivos jt400.jar y jt400ntv.jar. Ello hará que
el cargador de clases de ampliación cargue esos archivos. El cargador de clases de aplicación
también puede cargar dichos archivos si se les incluye en la variable CLASSPATH.
Si esos archivos se cargan a partir del directorio de vía de acceso de clases,
no añada el enlace simbólico al directorio de ampliaciones (ext).
El archivo jaas13.jar necesita estos archivos JAR para las clases de
implementación de credenciales que forman parte del programa producto
bajo licencia IBM Toolbox para Java (5722-JC1).
(En el tema acerca de IBM Toolbox para Java hallará documentación
relacionada con las clases de credenciales situadas en el marco izquierdo
bajo Clases de seguridad => Autenticación. Seleccione el enlace de acceso a la
clase ProfileTokenCredential. En la parte superior, seleccione 'Este paquete' para
obtener todo el paquete Java com/ibm/as400/security/auth. Para obtener el Javadoc
de las clases de autenticación, puede seleccionar 'Javadoc' =>
'Clases de acceso' en el marco izquierdo. Seleccione 'Todos los paquetes' en la parte
superior y localice los paquetes com.ibm.as400.security.*).


ADDLNK OBJ('/QIBM/ProdData/HTTP/Public/jt400/lib/jt400.jar')
  NEWLNK('/QIBM/ProdData/Java400/jdk13/lib/ext/jt400.jar')

ADDLNK OBJ('/QIBM/ProdData/OS400/jt400/lib/jt400Native.jar')
  NEWLNK('/QIBM/ProdData/Java400/jdk13/lib/ext/jt400Native.jar')

////////////////////////////////////
NOTAS IMPORTANTES -
////////////////////////////////////

Si va a actualizar los archivos de política de Java2
para una aplicación real, no olvide otorgar los debidos
permisos sobre las ubicaciones reales de los archivos jar de
IBM Toolbox para Java. Aunque estos archivos estén simbólicamente enlazados
con los directorios de ampliaciones anteriores a los que se ha otorgado
java.security.AllPermission en el archivo
${java.home}/lib/security/java.policy, la autorización se basa en
la ubicación real de los archivos JAR.

Por ejemplo, para utilizar satisfactoriamente las clases de credenciales
de IBM Toolbox para Java, se añadiría el siguiente fragmento de código al
archivo de política Java2 de la aplicación:

grant  codeBase "file:/QIBM/ProdData/HTTP/Public/jt400/lib/jt400.jar"
{
  permission javax.security.auth.AuthPermission "modifyThreadIdentity";
  permission java.lang.RuntimePermission "loadLibrary.*";
  permission java.lang.RuntimePermission "writeFileDescriptor";
```

```
  permission java.lang.RuntimePermission "readFileDescriptor";
 }

 También tendrá que añadir estos permisos para el parámetro
 codeBase de la aplicación, ya que las operaciones efectuadas por los
 archivos JAR de IBM Toolbox para Java no se ejecutan en modalidad privilegiada.

 Este ejemplo ya otorga estos permisos a todas las clases Java
 al omitir el parámetro codeBase del archivo threadJava2.policy.

 8) Asegúrese de que se han iniciado los servidores de sistema principal
 y que están funcionando. Las clases ProfileTokenCredential que
residen en IBM Toolbox para Java,
es decir jt400.jar, se emplean como
credenciales conectadas
al sujeto autenticado mediante el programa SampleThreadSubjectLogin.java.
Las clases de credenciales de IBM Toolbox para Java necesitan acceso
a los servidores de sistema principal.

 9) Invoque SampleThreadSubjectLogin mientras esté conectado como
 usuario que no tiene acceso a 'suDirPruebas/thread.txt'.

 10) Inicie el ejemplo entrando los siguientes mandatos CL  =>

 CHGCURDIR DIR('suDirPruebas')

 JAVA CLASS(SampleThreadSubjectLogin)
  CLASSPATH('suDirPruebas/classes')
  PROP((java.version '1.3')
       (java.security.manager)
       (java.security.auth.login.config
        'suDirPruebas/threadLogin.config')
       (java.security.policy
        'suDirPruebas/threadJava2.policy')
       (java.security.auth.policy
        'suDirPruebas/threadJaas.policy'))

 Cuando se le solicite, entre el ID de usuario y la contraseña del paso 1.

 11) Busque en suDirPruebas/thread.txt la entrada "Hello World".

 *
 **/


public class SampleThreadSubjectLogin {
/**
 * Se intenta autenticar al usuario.
 *
 * @param args
 *      Argumentos de entrada para esta aplicación (se pasan por alto).
 *
 */
    public static void main(String[] args) {

    // Se usan los módulos LoginModule configurados para la entrada "AS400ToolboxApp".
    LoginContext lc = null;
            try {
        // Si se proporciona, se utilizará el mismo sujeto para varios intentos de inicio de sesión.
        lc = new LoginContext("AS400ToolboxApp",
                   new Subject(),
                   new SampleCBHandler());
    } catch (LoginException le) {
        le.printStackTrace();
         System.exit(-1);
    }
```

```
// El usuario dispone de 3 intentos para autenticarse satisfactoriamente.
int i;
for (i = 0; i < 3; i++) {
        try {

    // Intento de autenticación.
    lc.login();

    // Si no se devuelve ninguna excepción,
    // la autenticación ha sido satisfactoria.
    break;

    } catch (AccountExpiredException aee) {

    System.out.println("Su cuenta ha caducado");
     System.exit(-1);

    } catch (CredentialExpiredException cee) {

    System.out.println("Sus credenciales han caducado.");
     System.exit(-1);

    } catch (FailedLoginException fle) {

    System.out.println("Autenticación fallida");
        try {
        Thread.currentThread().sleep(3000);
                } catch (Exception e) {
        // Se pasa por alto.
    }

                } catch (Exception e) {

    System.out.println("Excepción no prevista - imposible continuar");
     e.printStackTrace();
     System.exit(-1);
    }
}

// ¿Se ha fallado tres veces?
if (i == 3) {
    System.out.println("Lo lamentamos, la autenticación ha fallado");
     System.exit(-1);
}


// Se visualizan los Principales autenticados y las credenciales.
System.out.println("Autenticación satisfactoria");

System.out.println("Principales:");

Iterator itr = lc.getSubject().getPrincipals().iterator();

while (itr.hasNext())
    System.out.println(itr.next());

itr = lc.getSubject().getPrivateCredentials().iterator();

while (itr.hasNext())
    System.out.println(itr.next());

itr = lc.getSubject().getPublicCredentials().iterator();

while (itr.hasNext())
    System.out.println(itr.next());
```

```
        // Se hace algún trabajo basado en Principales:
    ThreadSubject.doAsPrivileged(lc.getSubject(), new java.security.PrivilegedAction() {
        public Object run() {
           System.out.println("\nSu propiedad java.home: "
                              +System.getProperty("java.home"));
           System.out.println("\nSu propiedad user.home: "
                              +System.getProperty("user.home"));
           File f = new File("thread.txt");
           System.out.print("\nthread.txt ");
           if (!f.exists()) System.out.print("no ");
           System.out.println("existe en el directorio actual");

           try {
               // Se escribe "Hello World número   x"  en thread.txt.
               PrintStream ps = new PrintStream(new FileOutputStream("thread.txt", true), true);

               long flen = f.length();
               ps.println("Hello World número    " +
                  Long.toString(flen/22) +
                  "\n");
               ps.close();
                    } catch (Exception e) {
         e.printStackTrace();
           }

           System.out.println("\nPor cierto, " + SampleThreadSubjectLogin.getCurrentUser());
           try {
               Thread.currentThread().sleep(2000);
                    } catch (Exception e) {
            // Se pasa por alto.
               }
           System.out.println("\n\nHello World!\n");
           return null;
           }
       }, null);

        System.exit(0);

  }       // Fin de main().


// Devuelve la identidad de OS actual de la hebra principal de la aplicación.
// (Esta rutina utiliza clases de IBM Toolbox para Java)
// Nota: las aplicaciones que se ejecutan en una hebra secundaria no
// pueden emplear esta API para determinar el usuario actual.
    static public String getCurrentUser() {

            try {
        AS400 localSys = new AS400("localhost", "*CURRENT", "*CURRENT");

        int ccsid = localSys.getCcsid();
        ProgramCall qusrjobi = new ProgramCall(localSys);
        ProgramParameter[] parms = new ProgramParameter[6];

        int rLength = 100;
        parms[0] = new ProgramParameter(rLength);
        parms[1] = new ProgramParameter(new AS400Bin4().toBytes(rLength));
        parms[2] = new ProgramParameter(new AS400Text(8, ccsid, localSys).toBytes("JOBI0600"));
        parms[3] = new ProgramParameter(new AS400Text(26,ccsid, localSys).toBytes("*"));
        parms[4] = new ProgramParameter(new AS400Text(16,ccsid, localSys).toBytes(""));
        parms[5] = new ProgramParameter(new AS400Bin4().toBytes(0));

        qusrjobi.setProgram(QSYSObjectPathName.toPath("QSYS", "QUSRJOBI", "PGM"), parms);
        AS400Text uidText = new AS400Text(10, ccsid, localSys);

    // Se invoca la API QUSRJOBI.
        qusrjobi.run();
```

```
        byte[] uidBytes = new byte[10];
        System.arraycopy((qusrjobi.getParameterList())[0].getOutputData(), 90, uidBytes, 0, 10);

        return ((String)(uidText.toObject(uidBytes))).trim();
    }


    catch (Exception e) {
         e.printStackTrace();
    }

    return "";
    }

} //Fin de la clase SampleThreadSubjectLogin.



/**
 * Se pasa un CallbackHandler a los servicios de seguridad
 * subyacentes para que puedan interaccionar con la aplicación
 * con el fin de recuperar datos de autenticación específicos,
 * como los nombres de usuario y las contraseñas, o con el fin
 * de visualizar información concreta, como los mensajes de error
 * y de aviso.
 *
 * CallbackHandlers se implementan de forma dependiente
 * de aplicación y de plataforma. La implementación decide
 * cómo hay que recuperar y visualizar la información en función
 * de los retornos de llamada (Callback) que se le pasan.
 *
 * Esta clase proporciona un CallbackHandler de ejemplo para las aplicaciones
 * que se ejecutan en un entorno OS/400. Sin embargo, no está pensada para
 * satisfacer los requisitos de las aplicaciones en la fase de producción.
 * Según se ha indicado, el CallbackHandler se considera en último término
 * como dependiente de la aplicación, pues las aplicaciones individuales
 * tienen sus propios requisitos de comprobación de errores, manejo de datos
 * e interfaz de usuario.
 *
 * Se manejan los siguientes retornos de llamada (callback):
 *
• *
• NameCallback *
• PasswordCallback *
• TextOutputCallback *
 *
 * A efectos de simplicidad, las solicitudes se manejan de forma interactiva
 * por medio de la entrada y la salida estándar. Sin embargo, cabe mencionar
 * que, cuando la consola proporciona entrada estándar, este
 * enfoque permitirá que se vean las contraseñas según se van
 * tecleando. Este aspecto se debe evitar en el caso de las
 * aplicaciones en fase de producción.
 *
 * Este CallbackHandler también permite adquirir un nombre
 * y una contraseña mediante un mecanismo alternativo
 * y establecerlos directamente en el manejador para eludir
 * la necesidad de la interacción de usuario en los
 * correspondientes retornos de llamada.
 *
 */
class SampleCBHandler implements CallbackHandler {
    private String name_ = null;
    private String password_ = null;
/**
```

```
 * Construye un nuevo SampleCBHandler.
 *
 */
public SampleCBHandler() {
    this(null, null);
}
/**
 * Construye un nuevo SampleCBHandler.
 *
 * Se puede especificar opcionalmente un nombre y una contraseña
 * con el fin de eludir la necesidad de solicitar información
 * en los correspondientes retornos de llamada.
 *
 * @param name
 *      El valor por omisión de los retornos de llamada del nombre.
 *      El valor nulo indica que hay que solicitar esta
 *      información al usuario. Los valores no nulos deben
 *      tener una longitud mayor que cero, sin superar los 10 caracteres.
 *
 * @param password
 *      El valor por omisión de los retornos de llamada de la contraseña.
 *      El valor nulo indica que hay que solicitar esta
 *      información al usuario. Los valores no nulos deben
 *      tener una longitud mayor que cero, sin superar los 10 caracteres.
 */
public SampleCBHandler(String name, String password) {
    if (name != null)
        if ((name.length()==0) || (name.length()>10))
            throw new IllegalArgumentException("name");
    name_ = name;

    if (password != null)
        if ((password.length()==0) || (password.length()>10))
            throw new IllegalArgumentException("password");
    password_ = password;
}
/**
 * Manejar el retorno de llamada del nombre dado.
 *
 * En primer lugar, se comprueba si se ha pasado un nombre
 * en el constructor. Si es así, se asigna el nombre al
 * retorno de llamada y se elude la solicitud.
 *
 * Si no se ha preestablecido ningún valor, se intenta solicitar
 * el nombre utilizando la entrada y la salida estándar.
 *
 * @param c
 *      El retorno de llamada del nombre (NameCallback).
 *
 * @exception java.io.IOException
 *      Si se produce un error de entrada o de salida.
 *
 */
private void handleNameCallback(NameCallback c) throws IOException {
    // Se compueba si existe un valor en antememoria.
    if (name_ != null) {
        c.setName(name_);
      return;
    }
    // No hay ningún valor preestablecido; se intenta stdin/out.
    c.setName(
        stdIOReadName(c.getPrompt(), 10));
}
/**
 * Manejar el retorno de llamada del nombre dado.
 *
 * En primer lugar, se comprueba si se ha pasado una contraseña
```

```
 * en el constructor. Si es así, se asigna la contraseña al
 * retorno de llamada y se elude la solicitud.
 *
 * Si no se ha preestablecido ningún valor, se intenta solicitar
 * la contraseña utilizando la entrada y la salida estándar.
 *
 * @param c
 *      El retorno de llamada de la contraseña (PasswordCallback).
 *
 * @exception java.io.IOException
 *      Si se produce un error de entrada o de salida.
 *
 */
private void handlePasswordCallback(PasswordCallback c) throws IOException {
    // Se compueba si existe un valor en antememoria.
    if (password_ != null) {
        c.setPassword(password_.toCharArray());
      return;
    }

    // No hay ningún valor preestablecido; se intenta stdin/out.
    // Nota:  Uso no adecuado para producción.
    //        La consola estándar de E/S no esconde la contraseña.
    if (c.isEchoOn())
        c.setPassword(
            stdIOReadName(c.getPrompt(), 10).toCharArray());
     else
    {

    //  Nota: la consola estándar de E/S no esconde la contraseña.
        c.setPassword(stdIOReadName(c.getPrompt(), 10).toCharArray());

    }
}
/**
 * Manejar el retorno de llamada de la salida de texto dada.
 *
 * Si el texto es informativo o representa un aviso,
 * se escribe en la salida estándar. Si el
 * retorno de llamada define un mensaje de error, el texto
 * se escribe en la salida de error estándar.
 *
 * @param c
 *      El retorno de llamada de la salida de texto (TextOutputCallback).
 *
 * @exception java.io.IOException
 *      Si se produce un error de entrada o de salida.
 *
 */
private void handleTextOutputCallback(TextOutputCallback c) throws IOException {
    if (c.getMessageType() == TextOutputCallback.ERROR)
        System.err.println(c.getMessage());
     else
        System.out.println(c.getMessage());
}
/**
 * Recuperar o visualizar la información solicitada en
 * los retornos de llamada proporcionados.
 *
 * La implementación del método handle
 * comprueba la(s) instancia(s) del objeto Callback
 * (uno o varios) pasado para recuperar o visualizar la
 * información solicitada.
 *
 * @param callbacks
 *      Una matriz de objetos Callback proporcionada
 *      por un servicio de seguridad subyacente y que contiene
```

```
 *      la información solicitada que hay que recuperar o visualizar.
 *
 * @exception java.io.IOException
 *      Si se produce un error de entrada o de salida.
 *
 * @exception UnsupportedCallbackException
 *      Si la  implementación de este método no da soporte a
 *      uno o varios de los objetos Callback especificados en
 *      el parámetro callbacks.
 *
 */
public void handle(Callback[] callbacks)
    throws IOException, UnsupportedCallbackException
{
    for (int i=0; i<callbacks.length; i++) {
        Callback c = callbacks[i];

        if (c instanceof NameCallback)
            handleNameCallback((NameCallback)c);
        else if (c instanceof PasswordCallback)
            handlePasswordCallback((PasswordCallback)c);
        else if (c instanceof TextOutputCallback)
            handleTextOutputCallback((TextOutputCallback)c);
     else
            throw new UnsupportedCallbackException
                (callbacks[i]);
    }
}
/**
 * Visualiza la serie dada utilizando la salida estándar,
 * seguida de un espacio para separar de la próxima
 * entrada.
 *
 * @param prompt
 *      El texto que hay que visualizar.
 *
 * @exception IOException
 *      Si se produce un error de entrada o de salida.
 *
 */
private void stdIOPrompt(String prompt) throws IOException {
    System.out.print(prompt + ' ');
    System.out.flush();
}
/**
 * Lee una serie de la entrada estándar, terminada según el valor
 * de maxLength o mediante un carácter de nueva línea. *
 * @param prompt
 *      El texto que hay que visualizar inmediatamente en la salida estándar
 *      antes de leer el valor solicitado.
 *
 * @param maxLength
 *      Longitud máxima de la serie que hay que devolver.
 *
 * @return
 *      La serie entrada. El valor devuelto no
 *      contendrá espacio en blanco al principio ni al final
 *      y se convertirá a mayúsculas.
 *
 * @exception IOException
 *      Si se produce un error de entrada o de salida.
 *
 */
private String stdIOReadName(String prompt, int maxLength) throws IOException {
    stdIOPrompt(prompt);
    String s =
        (new BufferedReader
```

```
            (new InputStreamReader(System.in))).readLine().trim();
    if (s.length() < maxLength)
        s = s.substring(0,maxLength);
    return s.toUpperCase();
}

}// Fin de la clase SampleCBHandler.
```

Ejemplo: programa cliente no JAAS de IBM JGSS

Para obtener más información sobre cómo utilizar el programa cliente de ejemplo, consulte Bajar y ejecutar los programas de ejemplo.

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
// Programa cliente de ejemplo de IBM JGSS 1.0

package com.ibm.security.jgss.test;
import org.ietf.jgss.*;
import com.ibm.security.jgss.Debug;

import java.io.*;
import java.net.*;
import java.util.*;

/**
 * Un cliente de ejemplo de JGSS
 * para utilizarse junto con el servidor de ejemplo de JGSS.
 * El cliente primero establece un contexto con el servidor
 * y después envía un mensaje envuelto seguido de un MIC al servidor.
 * El MIC se calcula a partir del texto plano envuelto.
 * El cliente requiere que el servidor se autentique
 * (autenticación mutua) durante el establecimiento del contexto.
 * También delega sus credenciales al servidor.
 *
 * Establece la variable JAVA
 * javax.security.auth.useSubjectCredsOnly en false
 * por lo que JGSS no adquirirá credenciales por medio de JAAS.
 *
 * El cliente toma parámetros de entrada y los complementa
 * con información del archivo jgss.ini; la entrada necesaria no
 * proporcionada en la línea de mandatos se toma del archivo jgss.ini.
 *
 * Uso: Client [opciones]
 *
 * La opción -? genera un mensaje de ayuda que contiene las opciones soportadas.
 *
 * Este cliente de ejemplo no utiliza JAAS.
 * El cliente se puede ejecutar para el cliente y el servidor de ejemplo de JAAS.
 * Vea en {@link JAASClient JAASClient} un cliente de ejemplo que emplea JAAS.
 */

class Client
{
    private Util testUtil      = null;
    private String myName      = null;
    private GSSName gssName    = null;
    private String serverName = null;
    private int servicePort    = 0;
    private GSSManager mgr     = GSSManager.getInstance();
    private GSSName service    = null;
    private GSSContext context        = null;
    private String program     = "Client";
    private String debugPrefix = "Client: ";
    private TCPComms tcp       = null;
    private String data               = null;
```

```
private byte[] dataBytes   = null;
private String serviceHostname= null;
private GSSCredential gssCred = null;

private static Debug debug        = new Debug();

private static final String usageString =
      "\t[-?] [-d | -n name] [-s serverName]"
    + "\n\t[-h serverHost [:port]] [-p port] [-m msg]"
    + "\n"
    + "\n  -?\t\t\thelp; produces this message"
    + "\n  -n name\t\tthe client's principal name (without realm)"
    + "\n  -s serverName\t\tthe server's principal name (without realm)"
    + "\n  -h serverHost[:port]\tthe server's hostname"
    +         " (and optional port number)"
    + "\n  -p port\t\tthe port on which the server will be listening"
    + "\n  -m msg\t\tmessage to send to the server";


// El llamador debe llamar a initialize (puede que primero tenga que llamar a processArgs).
public Client (String programName) throws Exception
{
    testUtil = new Util();
    if (programName != null)
    {
        program = programName;
        debugPrefix = programName + ": ";
    }
}

// El llamador debe llamar a initialize (puede que primero tenga que llamar a processArgs).
Client (String programName, boolean useSubjectCredsOnly) throws Exception
{
    this(programName);
    setUseSubjectCredsOnly(useSubjectCredsOnly);
}

public Client(GSSCredential myCred,
                  String serverNameWithoutRealm,
                  String serverHostname,
                  int    serverPort,
              String message)
throws Exception
{
    testUtil = new Util();

    if (myCred != null)
    {
        gssCred = myCred;
    }
 else
    {
        throw new GSSException(GSSException.NO_CRED, 0,
                                   "Null input credential");
    }

    init(serverNameWithoutRealm, serverHostname, serverPort, message);
}

void setUseSubjectCredsOnly(boolean useSubjectCredsOnly)
{
    final String subjectOnly = useSubjectCredsOnly ? "true" : "false";
    final String property = "javax.security.auth.useSubjectCredsOnly";

    String temp = (String)java.security.AccessController.doPrivileged(
                    new sun.security.action.GetPropertyAction(property));
```

```
    if (temp == null)
    {
              debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
          + "setting useSubjectCredsOnly property to "
          + useSubjectCredsOnly);

        // Propiedad no establecida. Establecerla en el valor especificado.

        java.security.AccessController.doPrivileged(
             new java.security.PrivilegedAction() {
       public Object run() {
                   System.setProperty(property, subjectOnly);
       return null;
               }
             });
    }
 else
    {
              debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
          + "useSubjectCredsOnly property already set "
          + "in JVM  to " + temp);
    }
}

private void init(String myNameWithoutRealm,
                  String serverNameWithoutRealm,
                  String serverHostname,
                  int    serverPort,
                  String message) throws Exception
{
    myName = myNameWithoutRealm;
    init(serverNameWithoutRealm, serverHostname, serverPort, message);
}

private void init(String serverNameWithoutRealm,
                  String serverHostname,
                  int    serverPort,
                  String message) throws Exception
{
    // nombre del igual
    if (serverNameWithoutRealm != null)
    {
        this.serverName = serverNameWithoutRealm;
    }
 else
    {
        this.serverName = testUtil.getDefaultServicePrincipalWithoutRealm();
    }

    // sistema principal del igual
    if (serverHostname != null)
    {
        this.serviceHostname = serverHostname;
    }
 else
    {
        this.serviceHostname = testUtil.getDefaultServiceHostname();
    }

    // puerto del igual
    if (serverPort > 0)
    {
        this.servicePort = serverPort;
    }
 else
    {
        this.servicePort = testUtil.getDefaultServicePort();
```

```
    }

    // mensaje para el igual
    if (message != null)
    {
        this.data = message;
    }
 else
    {
        this.data = "The quick brown fox jumps over the lazy dog";
    }

    this.dataBytes = this.data.getBytes();

    tcp = new TCPComms(serviceHostname, servicePort);
}


void initialize() throws Exception
{
     Oid krb5MechanismOid = new Oid("1.2.840.113554.1.2.2");

    if (gssCred == null)
    {
        if (myName != null)
        {
              debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                               + "creating GSSName USER_NAME for "
                               + myName);

            gssName = mgr.createName(
                           myName,
                           GSSName.NT_USER_NAME,
                              krb5MechanismOid);

              debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                               + "Canonicalized GSSName=" + gssName);
        }
 else
            gssName = null; // para credenciales por omisión


        debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix + "creating"
           + ((gssName == null)? " default " : " ")
           + "credential");

        gssCred = mgr.createCredential(
                                gssName,
                                GSSCredential.DEFAULT_LIFETIME,
                                (Oid)null,
                                GSSCredential.INITIATE_ONLY);
        if (gssName == null)
        {
            gssName = gssCred.getName();

            myName  = gssName.toString();

        debug.out(Debug.OPTS_CAT_APPLICATION,
                debugPrefix + "default credential principal=" + myName);
        }
    }

    debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix + gssCred);

              debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
       + "creating canonicalized GSSName for serverName " + serverName);
```

```
    service = mgr.createName(serverName,
                             GSSName.NT_HOSTBASED_SERVICE,
                              krb5MechanismOid);

              debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
       + "Canonicalized server name = " + service);

        debug.out(Debug.OPTS_CAT_APPLICATION,
                        debugPrefix + "Raw data=" + data);
}


void establishContext(BitSet flags) throws Exception
{
        try {

        debug.out(Debug.OPTS_CAT_APPLICATION,
                        debugPrefix + "creating GSScontext");

      Oid defaultMech = null;
      context = mgr.createContext(service, defaultMech, gssCred,
                                  GSSContext.INDEFINITE_LIFETIME);

      if (flags != null)
      {
          if (flags.get(Util.CONTEXT_OPTS_MUTUAL))
          {
              debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                   + "requesting mutualAuthn");

              context.requestMutualAuth(true);
          }

          if (flags.get(Util.CONTEXT_OPTS_INTEG))
          {
              debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                   + "requesting integrity");

              context.requestInteg(true);
          }

          if (flags.get(Util.CONTEXT_OPTS_CONF))
          {
              context.requestConf(true);
              debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                      + "requesting confidentiality");
          }

          if (flags.get(Util.CONTEXT_OPTS_DELEG))
          {
              context.requestCredDeleg(true);
              debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                       + "requesting delegation");
          }

          if (flags.get(Util.CONTEXT_OPTS_REPLAY))
          {
              context.requestReplayDet(true);
              debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                  + "requesting replay detection");
          }

          if (flags.get(Util.CONTEXT_OPTS_SEQ))
          {
              context.requestSequenceDet(true);
              debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                       + "requesting out-of-sequence detection");
```

```
            }
            // Añadir más posteriormente
        }

        byte[] response = null;
        byte[] request = null;
        int len = 0;
        boolean done = false;
do {
                debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                + "Calling initSecContext");

            request = context.initSecContext(response, 0, len);

            if (request != null)
            {
                debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                  + "Sending initial context token");

                tcp.send(request);
            }
            done = context.isEstablished();

            if (!done)
            {
          debug.out(Debug.OPTS_CAT_APPLICATION,
                    debugPrefix + "Receiving response token");

                byte[] temp = tcp.receive();
                response = temp;
                len = response.length;
            }
        } while(!done);

          debug.out(Debug.OPTS_CAT_APPLICATION,
              debugPrefix + "context established with acceptor");

    } catch (Exception exc) {
          exc.printStackTrace();
          throw exc;
    }
}

void doMIC() throws Exception
{
    debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix + "generating MIC");
    byte[] mic = context.getMIC(dataBytes, 0, dataBytes.length, null);

    if (mic != null)
    {
        debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix + "sending MIC");
        tcp.send(mic);
    }
 else
        debug.out(Debug.OPTS_CAT_APPLICATION,
                         debugPrefix + "getMIC Failed");
}

void doWrap() throws Exception
{
    MessageProp mp = new MessageProp(true);
    mp.setPrivacy(context.getConfState());

    debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix + "wrapping message");

    byte[] wrapped = context.wrap(dataBytes, 0, dataBytes.length, mp);
```

```
    if (wrapped != null)
    {
        debug.out(Debug.OPTS_CAT_APPLICATION,
                debugPrefix + "sending wrapped message");

        tcp.send(wrapped);
    }
 else
        debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix + "wrap Failed");
}

void printUsage()
{
    System.out.println(program + usageString);
}

void processArgs(String[] args) throws Exception
{
    String port            = null;
    String myName          = null;
    int servicePort               = 0;
    String serviceHostname = null;

    String sHost = null;
    String msg = null;

    GetOptions options = new GetOptions(args, "?h:p:m:n:s:");
    int ch = -1;
    while ((ch = options.getopt()) != options.optEOF)
    {
        switch(ch)
        {
            case '?':
                printUsage();
        System.exit(1);

            case 'h':
                if (sHost == null)
                {
                    sHost = options.optArgGet();
                    int p = sHost.indexOf(':');
        if (p != -1)
                    {
                        String temp1 = sHost.substring(0, p);
                if (port == null)
                            port = sHost.substring(p+1, sHost.length()).trim();
                        sHost = temp1;
                    }
                }
    continue;

            case 'p':
                if (port == null)
                    port = options.optArgGet();
    continue;

            case 'm':
                if (msg == null)
                    msg = options.optArgGet();
    continue;

            case 'n':
                if (myName == null)
                    myName = options.optArgGet();
    continue;

            case 's':
```

```
                if (serverName == null)
                    serverName = options.optArgGet();
    continue;
        }
    }

    if ((port != null) && (port.length() > 0))
    {
        int p = -1;
       try {
            p = Integer.parseInt(port);
    } catch (Exception exc) {
            System.out.println("Bad port input: "+port);
        }

        if (p != -1)
            servicePort = p;
    }

    if ((sHost != null) && (sHost.length() > 0)) {
            serviceHostname = sHost;
    }

    init(myName, serverName, serviceHostname, servicePort, msg);
}

void interactWithAcceptor(BitSet flags) throws Exception
{
    establishContext(flags);
    doWrap();
    doMIC();
}

void interactWithAcceptor() throws Exception
{
    BitSet flags = new BitSet();
    flags.set(Util.CONTEXT_OPTS_MUTUAL);
    flags.set(Util.CONTEXT_OPTS_CONF);
    flags.set(Util.CONTEXT_OPTS_INTEG);
    flags.set(Util.CONTEXT_OPTS_DELEG);
    interactWithAcceptor(flags);
}

void dispose() throws Exception
{
    if (tcp != null)
    {
        tcp.close();
    }
}

public static void main(String args[]) throws Exception
{
    System.out.println(debug.toString()); // XXXXXXX
    String programName = "Client";
    Client client = null;
        try {
        client = new Client(programName,
                            false); // no utilizar credenciales del sujeto
        client.processArgs(args);
        client.initialize();
        client.interactWithAcceptor();
    } catch (Exception exc) {
        debug.out(Debug.OPTS_CAT_APPLICATION,
                        programName + " Exception: " + exc.toString());
        exc.printStackTrace();
        throw exc;
```

```
        } finally {
           try {
                if (client != null)
                    client.dispose();
            } catch (Exception exc) {}
        }

        debug.out(Debug.OPTS_CAT_APPLICATION, programName + ": done");
    }
}
```

Ejemplo: programa servidor no JAAS de IBM JGSS

Para obtener más información sobre cómo utilizar el programa servidor de ejemplo, consulte Bajar y ejecutar los ejemplos de IBM JGSS.

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
// Programa servidor de ejemplo de IBM JGSS 1.0

package com.ibm.security.jgss.test;

import org.ietf.jgss.*;
import com.ibm.security.jgss.Debug;
import java.io.*;
import java.net.*;
import java.util.*;

/**
 * Un servidor de ejemplo de JGSS para emplearse con un cliente de ejemplo de JGSS.
 *
 * Está continuamente a la escucha de conexiones de cliente
 * y engendra una hebra para dar servicio a una conexión entrante.
 * Puede ejecutar varias hebras de forma simultánea.
 * Dicho de otro modo, puede dar servicio a varios clientes a la vez.
 *
 * Cada hebra primero establece un contexto con el cliente
 * y después espera un mensaje envuelto seguido de un MIC.
 * Supone que el cliente ha calculado el MIC a partir del texto
 * plano envuelto por el cliente.
 *
 * Si el cliente delega su credencial al servidor, la credencial
 * delegada se utiliza para la comunicación con un servidor secundario.
 *
 * Asimismo, el servidor puede iniciarse para actuar como cliente y
 * servidor (mediante la opción -b). En este caso, la primera hebra
 * engendrada por el servidor utiliza la credencial propia del principal
 * del servidor para comunicarse con el servidor secundario.
 *
 * El servidor secundario debe haberse iniciado antes que el servidor (primario)
 * inicie el contacto con él (el servidor secundario).
 * En la comunicación con el servidor secundario, el servidor primario actúa
 * como iniciador de JGSS (es decir, cliente) que establece un contexto y
 * participa en intercambios de envoltura y MIC por mensaje con el servidor secundario.
 *
 * El servidor toma parámetros de entrada y los complementa
 * con información del archivo jgss.ini; la entrada necesaria no
 * proporcionada en la línea de mandatos se toma del archivo jgss.ini.
 * Los valores por omisión incorporados se emplean si no hay un archivo jgss.ini
 * o una variable determinada no está especificada en el archivo ini.
 *
 * Uso: Server [opciones]
 *
 * La opción -? genera un mensaje de ayuda que contiene las opciones soportadas.
 *
 * Este servidor de ejemplo no utiliza JAAS.
```

```
 * Establece la variable JAVA
 * javax.security.auth.useSubjectCredsOnly en false
 * por lo que JGSS no adquirirá credenciales por medio de JAAS.
 * El servidor se puede ejecutar para los clientes y servidores de ejemplo de JAAS.
 * Vea en {@link JAASServer JAASServer} un servidor de ejemplo que emplea JAAS.
 */

class Server implements Runnable
{
    /*
     * NOTAS:
     * Está previsto que esta clase, Server, se ejecute en varias
     * hebras simultáneas. Las variables estáticas constan de variables
     * establecidas desde argumentos de línea de mandatos y variables (como
     * las propias credenciales del servidor, gssCred) establecidas una vez en
     * la inicialización. Estas variables no cambian después de establecerse
     * y las comparten todas las hebras en ejecución.
     *
     * La única variable estática que cambia tras establecerse inicialmente
     * es la variable beenInitiator, que la establece en true
     * la primera hebra que ejecuta el servidor como iniciador utilizando
     * las credenciales propias del servidor. Así se garantiza que el servidor
     * sólo se ejecuta como iniciador una vez. La consulta y la modificación
     * de beenInitiator están sincronizadas entre las hebras.
     *
     * La variable tcp no es estática y se establece para cada hebra
     * a fin de representar el socket en el que da servicio al cliente
     * la hebra conectada.
     */

    private static Util testUtil             = null;
    private static int myPort                = 0;
    private static Debug debug               = new Debug();
    private static String myName             = null;
    private static GSSCredential gssCred     = null;
    private static String serviceNameNoRealm = null;
    private static String serviceHost        = null;
    private static int    servicePort        = 0;
    private static String serviceMsg         = null;
    private static GSSManager mgr            = null;
    private static GSSName gssName           = null;
    private static String program            = "Server";
    private static boolean clientServer      = false;
    private static boolean primaryServer     = true;

    private static boolean beenInitiator     = false;

    private static final String usageString =
         "\t[-?] [-# number] [-d | -n name] [-p port]"
       + "\n\t[-s serverName] [-h serverHost [:port]] [-P serverPort] [- msg]"
       + "\n"
       + "\n  -?\t\t\thelp; produces this message"
       + "\n  -# number\t\tWhether primary or secondary server"
       +         " \n\t\t\t(1 = primary, 2 = secondary; default = first)"
       + "\n  -n name\t\tthe server's principal name (without realm)"
       + "\n  -p port\t\tthe port on which the server will be listening"
       + "\n  -s serverName\t\tsecondary server's principal name"
       +         " (without realm)"
       + "\n  -h serverHost[:port]\tsecondary server's hostname"
       +         " (and optional port number)"
       + "\n  -P port\t\tsecondary server's port number"
       + "\n  -m msg\t\tmessage to send to secondary server"
       + "\n  -b \t\trun as both client and server"
       +         " using the server's owns credentials";

    // Las variables no estáticas son específicas de las hebras
    // ya que cada hebra ejecuta una instancia aparte de esta clase.
```

```
private String debugPrefix = null;
private TCPComms tcp       = null;

static {
        try {
        testUtil = new Util();
    } catch (Exception exc) {
        exc.printStackTrace();
        System.exit(1);
    }
}

Server (Socket socket) throws Exception
{
    debugPrefix = program + ": ";
    tcp = new TCPComms(socket);
}

Server (String program) throws Exception
{
    debugPrefix = program + ": ";
    this.program = program;
}

Server (String program, boolean useSubjectCredsOnly) throws Exception
{
    this(program);
    setUseSubjectCredsOnly(useSubjectCredsOnly);
}

void setUseSubjectCredsOnly(boolean useSubjectCredsOnly)
{
    final String subjectOnly = useSubjectCredsOnly ? "true" : "false";
    final String property = "javax.security.auth.useSubjectCredsOnly";

    String temp = (String)java.security.AccessController.doPrivileged(
                    new sun.security.action.GetPropertyAction(property));

    if (temp == null)
    {
       debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
          + "setting useSubjectCredsOnly property to "
          + (useSubjectCredsOnly ? "true" : "false"));

        // Propiedad no establecida. Establecerla en el valor especificado.

        java.security.AccessController.doPrivileged(
             new java.security.PrivilegedAction() {
       public Object run() {
                  System.setProperty(property, subjectOnly);
       return null;
               }
             });
    }
 else
    {
       debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
          + "useSubjectCredsOnly property already set "
          + "in JVM  to " + temp);
    }
}

private void init(boolean  primary,
          String   myNameWithoutRealm,
          int      port,
          String   serverNameWithoutRealm,
```

```
          String   serverHostname,
          int      serverPort,
          String   message,
          boolean  clientServer)
    throws Exception
    {
        primaryServer = primary;
        this.clientServer = clientServer;

        myName = myNameWithoutRealm;

        // mi puerto
        if (port > 0)
        {
            myPort = port;
        }
        else if (primary)
        {
            myPort = testUtil.getDefaultServicePort();
        }
     else
        {
            myPort = testUtil.getDefaultService2Port();
        }

        if (primary)
        {
            ///// nombre del igual
            if (serverNameWithoutRealm != null)
            {
                serviceNameNoRealm = serverNameWithoutRealm;
            }
     else
            {
                serviceNameNoRealm =
                      testUtil.getDefaultService2PrincipalWithoutRealm();
            }

            // sistema principal del igual
            if (serverHostname != null)
            {
                if (serverHostname.equalsIgnoreCase("localHost"))
                {
                    serverHostname = InetAddress.getLocalHost().getHostName();
                }

                serviceHost = serverHostname;
            }
     else
            {
                serviceHost = testUtil.getDefaultService2Hostname();
            }

            // puerto del igual
            if (serverPort > 0)
            {
                servicePort = serverPort;
            }
     else
            {
                servicePort = testUtil.getDefaultService2Port();
            }

            // mensaje para el igual
            if (message != null)
            {
                serviceMsg = message;
```

```
        }
 else
        {
            serviceMsg = "Hi there! I am a server."
                         + "But I can be a client, too";
        }
    }

    String temp = debugPrefix + "details"
                  + "\n\tPrimary:\t" + primary
                  + "\n\tName:\t\t" + myName
                  + "\n\tPort:\t\t" + myPort
                  + "\n\tClient+server:\t" + clientServer;
    if (primary)
    {
        temp += "\n\tOther Server:"
                  + "\n\t\tName:\t" + serviceNameNoRealm
                  + "\n\t\tHost:\t" + serviceHost
                  + "\n\t\tPort:\t" + servicePort
                  + "\n\t\tMsg:\t" + serviceMsg;
    }

    debug.out(Debug.OPTS_CAT_APPLICATION, temp);
}


void initialize() throws GSSException
{
       debug.out(Debug.OPTS_CAT_APPLICATION,
                     debugPrefix + "creating GSSManager");

    mgr = GSSManager.getInstance();

    int usage = clientServer ? GSSCredential.INITIATE_AND_ACCEPT
                             : GSSCredential.ACCEPT_ONLY;

    if (myName != null)
    {
       debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                           + "creating GSSName for " + myName);

        gssName = mgr.createName(myName,
                                 GSSName.NT_HOSTBASED_SERVICE);

        Oid krb5MechanismOid = new Oid("1.2.840.113554.1.2.2");
        gssName.canonicalize(krb5MechanismOid);

       debug.out(Debug.OPTS_CAT_APPLICATION,
              debugPrefix + "Canonicalized GSSName=" + gssName);
    }
 else
        gssName = null;

    debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix + "creating"
           + ((gssName == null)? " default " : " ")
           + "credential");

    gssCred = mgr.createCredential(
                           gssName, GSSCredential.DEFAULT_LIFETIME,
                           (Oid)null, usage);
    if (gssName == null)
    {
        gssName = gssCred.getName();
        myName = gssName.toString();

       debug.out(Debug.OPTS_CAT_APPLICATION,
           debugPrefix + "default credential principal=" + myName);
```

```
    }
}


void processArgs(String[] args) throws Exception
{
    String port    = null;
    String name    = null;
    int iport      = 0;

    String sport   = null;
    int isport     = 0;
    String sname   = null;
    String shost   = null;
    String smessage        = null;

    boolean primary = true;
    String status  = null;

    boolean defaultPrinc = false;
    boolean clientServer = false;

    GetOptions options = new GetOptions(args, "?#:p:n:P:s:h:m:b");
    int ch = -1;
    while ((ch = options.getopt()) != options.optEOF)
    {
        switch(ch)
        {
            case '?':
                printUsage();
        System.exit(1);

            case '#':
                if (status == null)
                    status = options.optArgGet();
    continue;

            case 'p':
                if (port == null)
                    port = options.optArgGet();
    continue;

            case 'n':
    if (name == null)
                    name = options.optArgGet();
    continue;

            case 'b':
                clientServer = true;
    continue;

            ////// El otro servidor

            case 'P':
                        if (sport == null)
                    sport = options.optArgGet();
    continue;

            case 'm':
                if (smessage == null)
                    smessage = options.optArgGet();
    continue;

            case 's':
                if (sname == null)
                    sname = options.optArgGet();
```

```
continue;

        case 'h':
            if (shost == null)
            {
                shost = options.optArgGet();
                int p = shost.indexOf(':');
    if (p != -1)
                {
                    String temp1 = shost.substring(0, p);
                    if (sport == null)
                        sport = shost.substring
                                    (p+1, shost.length()).trim();
                    shost = temp1;
                }
            }
continue;
    }
}

if (defaultPrinc && (name != null))
{
    System.out.println(
      "ERROR: '-d' and '-n ' options are mutually exclusive");
          printUsage();
    System.exit(1);
}

if (status != null)
{
    int p = -1;
   try {
        p = Integer.parseInt(status);
} catch (Exception exc) {
        System.out.println( "Bad status input: "+status);
    }

    if (p != -1)
    {
        primary = (p == 1);
    }
}

if (port != null)
{
    int p = -1;
   try {
        p = Integer.parseInt(port);
} catch (Exception exc) {
        System.out.println( "Bad port input: "+port);
    }
    if (p != -1)
        iport = p;
}

if (sport != null)
{
    int p = -1;
   try {
        p = Integer.parseInt(sport);
} catch (Exception exc) {
        System.out.println( "Bad server port input: "+port);
    }
    if (p != -1)
        isport = p;
}
```

```
    init(primary,  // el servidor primero o segundo
         name,     // mi nombre
         iport,    // mi puerto
         sname,    // el nombre del otro servidor
         shost,    // el nombre de sistema principal del otro servidor
         isport,   // el puerto del otro servidor
         smessage, // mensaje para el otro servidor
         clientServer); // si se ejecutará como iniciador con credenciales propias
}

void processRequests() throws Exception
{
    ServerSocket ssocket = null;
    Server server = null;
        try {
        ssocket = new ServerSocket(myPort);
do {
       debug.out(Debug.OPTS_CAT_APPLICATION,
                 debugPrefix + "listening on port " + myPort + " ...");
            Socket csocket = ssocket.accept();

       debug.out(Debug.OPTS_CAT_APPLICATION,
                debugPrefix + "incoming connection on " + csocket);

            server = new Server(csocket); // establecer socket de cliente por hebra
            Thread thread = new Thread(server);
           thread.start();
            if (!thread.isAlive())
                server.dispose(); // cerrar el socket de cliente
      } while(true);
    } catch (Exception exc) {
       debug.out(Debug.OPTS_CAT_APPLICATION,
            debugPrefix + "*** ERROR processing requests ***");
        exc.printStackTrace();
    } finally {
       try {
            if (ssocket != null)
               ssocket.close(); // cerrar el socket de servidor
            if (server != null)
                server.dispose(); // cerrar el socket de cliente
        } catch (Exception exc) {}
    }
}

void dispose()
{
        try {
        if (tcp != null)
        {
            tcp.close();
            tcp = null;
        }
        } catch (Exception exc) {}
}

boolean establishContext(GSSContext context) throws Exception
{
    byte[] response               = null;
    byte[] request         = null;

       debug.out(Debug.OPTS_CAT_APPLICATION,
                     debugPrefix + "establishing context");

do {
          request = tcp.receive();
          if (request == null || request.length == 0)
        {
```

```
       debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                 + "Received no data; perhaps client disconnected");

        return false;
        }

        debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix + "accepting");
        if ((response = context.acceptSecContext
                             (request, 0, request.length)) != null)
        {
       debug.out(Debug.OPTS_CAT_APPLICATION,
                       debugPrefix + "sending response");
            tcp.send(response);
        }
    } while(!context.isEstablished());

       debug.out(Debug.OPTS_CAT_APPLICATION,
                debugPrefix + "context established - " + context);

        return true;
}

byte[] unwrap(GSSContext context, byte[] msg) throws Exception
{
    debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix + "unwrapping");

    MessageProp mp = new MessageProp(true);
    byte[] unwrappedMsg = context.unwrap(msg, 0, msg.length, mp);

       debug.out(Debug.OPTS_CAT_APPLICATION,
                         debugPrefix + "unwrapped msg is:");
    debug.out(Debug.OPTS_CAT_APPLICATION, unwrappedMsg);

    return unwrappedMsg;
}

void verifyMIC (GSSContext context, byte[] mic, byte[] raw) throws Exception
{
    debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix + "verifying MIC");

    MessageProp mp = new MessageProp(true);
    context.verifyMIC(mic, 0, mic.length, raw, 0, raw.length, mp);

       debug.out(Debug.OPTS_CAT_APPLICATION,
                   debugPrefix + "successfully verified MIC");
}

void useDelegatedCred(GSSContext context) throws Exception
{
    GSSCredential delCred = context.getDelegCred();
    if (delCred != null)
    {
        if (primaryServer)
        {
        debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix +
                    "Primary server received delegated cred; using it");
            runAsInitiator(delCred); // utilizar credenciales delegadas
        }
 else
        {
        debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix +
                   "Non-primary server received delegated cred; "
                        + "ignoring it");

        }
    }
 else
```

```
    {
        debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix +
                                    "ERROR: null delegated cred");
    }
}

public void run()
{
    byte[] response               = null;
    byte[] request         = null;
    boolean unwrapped      = false;
    GSSContext context     = null;

        try {
      Thread currentThread = Thread.currentThread();
      String threadName    = currentThread.getName();

      debugPrefix = program + " " + threadName + ": ";

       debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                           + "servicing client ...");

       debug.out(Debug.OPTS_CAT_APPLICATION,
          debugPrefix + "creating GSSContext");

      context = mgr.createContext(gssCred);

      // Establecer primero el contexto con el iniciador.
      if (!establishContext(context))
  return;

      // A continuación procesar los mensajes del iniciador.
      // Se espera recibir un mensaje envuelto seguido de un MIC.
      // El MIC debe haberse calculado a partir del texto plano
      // que se ha recibido envuelto.
      // Utilizar las credenciales delegadas si existen.
      // A continuación, ejecutar como iniciador utilizando las credenciales propias
      // si es necesario; sólo hace esto la primera hebra.

do {
       debug.out(Debug.OPTS_CAT_APPLICATION,
              debugPrefix + "receiving per-message request");

          request = tcp.receive();
          if (request == null || request.length == 0)
          {
       debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                 + "Received no data; perhaps client disconnected");

  return;
          }

          // Esperar primero mensaje envuelto.
          if (!unwrapped)
          {
              response = unwrap(context, request);
              unwrapped = true;
              continue; // obtener siguiente petición
          }

          // Seguido de un MIC.
          verifyMIC(context, request, response);

          // Suplantar al iniciador si ha delegado sus credenciales.
          if (context.getCredDelegState())
              useDelegatedCred(context);
```

```
       debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
              + "clientServer=" + clientServer
              + ", beenInitiator=" + beenInitiator);

         // Si es necesario, ejecutar como iniciador utilizando las credenciales propias.
    if (clientServer)
              runAsInitiatorOnce(currentThread);

         debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix + "done");
  return;

      } while(true);

    } catch (Exception exc) {
       debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix + "ERROR");
        exc.printStackTrace();

        // Silenciar las excepciones para cada hebra para no
        // desactivar el servidor debido a excepciones en
        // hebras individuales.
  return;
    } finally {
        if (context != null)
        {
       try {
                context.dispose();
        } catch (Exception exc) {}
        }
    }
}

synchronized void runAsInitiatorOnce(Thread thread)
    throws InterruptedException
{
    if (!beenInitiator)
    {
        // establecer pronto distintivo en true para evitar que hebras
        // posteriores intenten ejecutar runAsInitiator.
        beenInitiator = true;

        debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix +
                "About to run as initiator with own creds ...");

        //thread.sleep(30*1000, 0);
        runAsInitiator();
    }
}


void runAsInitiator(GSSCredential cred)
{
    Client client = null;
        try {
        client = new Client(cred,
                            serviceNameNoRealm,
                            serviceHost,
                            servicePort,
                            serviceMsg);

        client.initialize();

        BitSet flags = new BitSet();
        flags.set(Util.CONTEXT_OPTS_MUTUAL);
        flags.set(Util.CONTEXT_OPTS_CONF);
        flags.set(Util.CONTEXT_OPTS_INTEG);

        client.interactWithAcceptor(flags);
```

```
        } catch (Exception exc) {
           debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                  + "Exception running as initiator");

            exc.printStackTrace();
        } finally {
           try {
                client.dispose();
            } catch (Exception exc) {}
        }
    }

    void runAsInitiator()
    {
        if (clientServer)
        {
           debug.out(Debug.OPTS_CAT_APPLICATION,
               debugPrefix + "running as initiator with own creds");

            runAsInitiator(gssCred); // utilizar credenciales propias;
        }
     else
        {
           debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
               + "Cannot run as initiator with own creds "
               + "\nbecause not running as both initiator and acceptor.");
        }
    }

    void printUsage()
    {
        System.out.println(program + usageString);
    }

    public static void main(String[] args) throws Exception
    {
        System.out.println(debug.toString()); // XXXXXXX
        String programName = "Server";
            try {
            Server server = new Server(programName,
                                       false); // no utilizar credenciales del sujeto
            server.processArgs(args);
            server.initialize();
            server.processRequests();
        } catch (Exception exc) {
            debug.out(Debug.OPTS_CAT_APPLICATION, programName + ": EXCEPTION");
            exc.printStackTrace();
            throw exc;
        }
    }
}
```

Ejemplo: programa cliente habilitado para JAAS de IBM JGSS

Para obtener más información sobre cómo utilizar el programa cliente de ejemplo, consulte Bajar y ejecutar los ejemplos de IBM JGSS.

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
// Programa cliente habilitado para JAAS de ejemplo IBM Java GSS 1.0

package com.ibm.security.jgss.test;
import com.ibm.security.jgss.Debug;
import com.ibm.security.auth.callback.Krb5CallbackHandler;
import javax.security.auth.Subject;
```

```
import javax.security.auth.login.LoginContext;
import java.security.PrivilegedExceptionAction;

/**
 * Un cliente de ejemplo Java GSS que utiliza JAAS.
 *
 * Inicia una sesión en JAAS y opera en el contexto de inicio de sesión de JAAS creado con ese fin.
 *
 * No establece la variable JAVA
 * javax.security.auth.useSubjectCredsOnly, dejando
 * que la variable utilice el valor por omisión true
 * para que Java GSS adquiera credenciales del sujeto JAAS
 * asociado al contexto de inicio de sesión (creado por el cliente).
 *
 * JAASClient equivale a su superclase {@link Client Client}
 * en todos los demás aspectos y puede
 * ejecutarse para los servidores y clientes de ejemplo no JAAS.
 */

class JAASClient extends Client
{
    JAASClient(String programName) throws Exception
    {
        // No establecer useSubjectCredsOnly. Establecer sólo el nombre de programa.
        // useSubjectCredsOnly toma el valor por omisión "true" si no se establece.
        super(programName);
    }

    static class JAASClientAction implements PrivilegedExceptionAction
    {
         private JAASClient client;

         public JAASClientAction(JAASClient client)
         {
             this.client = client;
         }

         public Object run () throws Exception
         {
             client.initialize();
             client.interactWithAcceptor();
           return null;
         }
    }

    public static void main(String args[]) throws Exception
    {
        String programName = "JAASClient";
        JAASClient client = null;
        Debug dbg = new Debug();

        System.out.println(dbg.toString()); // XXXXXXX

            try {
            client = new JAASClient(programName);//utilizar credenciales del sujeto
            client.processArgs(args);

            LoginContext loginCtxt = new LoginContext("JAASClient",
                                       new Krb5CallbackHandler());

            loginCtxt.login();

        dbg.out(Debug.OPTS_CAT_APPLICATION,
                      programName + ": Kerberos login OK");

            Subject subject = loginCtxt.getSubject();
```

```
            PrivilegedExceptionAction jaasClientAction
                            = new JAASClientAction(client);

            Subject.doAsPrivileged(subject, jaasClientAction, null);

        } catch (Exception exc) {
        dbg.out(Debug.OPTS_CAT_APPLICATION,
                        programName + " Exception: " + exc.toString());
            exc.printStackTrace();
            throw exc;
        } finally {
           try {
                if (client != null)
                    client.dispose();
            } catch (Exception exc) {}
        }

        dbg.out(Debug.OPTS_CAT_APPLICATION,
                        programName + ": Done ...");
    }
}
```

Ejemplo: programa servidor habilitado para JAAS de IBM JGSS

Para obtener más información sobre cómo utilizar el programa servidor de ejemplo, consulte Bajar y ejecutar los ejemplos de IBM JGSS.

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
// Programa servidor habilitado para JAAS de ejemplo IBM Java GSS 1.0

package com.ibm.security.jgss.test;
import com.ibm.security.jgss.Debug;
import com.ibm.security.auth.callback.Krb5CallbackHandler;
import javax.security.auth.Subject;
import javax.security.auth.login.LoginContext;
import java.security.PrivilegedExceptionAction;

/**
 * Un servidor de ejemplo Java GSS que utiliza JAAS.
 *
 * Inicia una sesión en JAAS y opera en el contexto de inicio de sesión de JAAS creado con ese fin.
 *
 * No establece la variable JAVA
 * javax.security.auth.useSubjectCredsOnly, dejando
 * que la variable utilice el valor por omisión true
 * para que Java GSS adquiera credenciales del sujeto JAAS
 * asociado al contexto de inicio de sesión (creado por el servidor).
 *
 * JAASServer equivale a su superclase {@link Server Server}
 * en todos los demás aspectos y puede
 * ejecutarse para los servidores y clientes de ejemplo no JAAS.
 */

class JAASServer extends Server
{
    JAASServer(String programName) throws Exception
    {
        super(programName);
    }

    static class JAASServerAction implements PrivilegedExceptionAction
    {
        private JAASServer server = null;

        JAASServerAction(JAASServer server)
```

```
        {
            this.server = server;
        }

        public Object run() throws Exception
        {
            server.initialize();
            server.processRequests();

           return null;
        }
    }

    public static void main(String[] args) throws Exception
    {
        String programName     = "JAASServer";
        Debug dbg              = new Debug();

        System.out.println(dbg.toString()); // XXXXXXX

            try {
            // No establecer useSubjectCredsOnly.
            // useSubjectCredsOnly toma el valor por omisión "true" si no se establece.

            JAASServer server = new JAASServer(programName);

            server.processArgs(args);

            LoginContext loginCtxt = new LoginContext(programName,
                                        new Krb5CallbackHandler());

            dbg.out(Debug.OPTS_CAT_APPLICATION, programName + ": Login in ...");

            loginCtxt.login();

            dbg.out(Debug.OPTS_CAT_APPLICATION, programName +
                                            ": Login successful");

            Subject subject = loginCtxt.getSubject();

            JAASServerAction serverAction = new JAASServerAction(server);

            Subject.doAsPrivileged(subject, serverAction, null);
        } catch (Exception exc) {
            dbg.out(Debug.OPTS_CAT_APPLICATION, programName +  " EXCEPTION");
            exc.printStackTrace();
            throw exc;
        }
    }
}
```

Ejemplo: llamar a un programa CL con java.lang.Runtime.exec()

Este ejemplo muestra cómo ejecutar programas CL desde un programa Java. Consulte la sección Llamar a un mandato CL para obtener un ejemplo de cómo llamar a un mandato CL desde un programa Java. En este ejemplo, la clase Java CallCLPgm ejecuta un programa CL. Este utiliza el mandato Visualizar programa Java (DSPJVAPGM) para visualizar el programa asociado al archivo de clase Hello. En este ejemplo, se supone que el programa CL se ha compilado y está en una biblioteca que se llama JAVSAMPLIB. La salida del programa CL está en el archivo en spool QSYSPRT.

Nota: JAVSAMPLIB no se crea como parte del proceso de instalación del programa bajo licencia (LP) IBM Developer Kit, número 5722-JV1. Tendrá que crear la biblioteca de manera explícita.

Ejemplo 1: clase CallCLPgm

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
import java.io.*;

public class CallCLPgm
{
   public static void main(String[] args)
   {
       try
      {
         Process theProcess =
                 Runtime.getRuntime().exec("/QSYS.LIB/JAVSAMPLIB.LIB/DSPJVA.PGM");
      }
      catch(IOException e)
      {
         System.err.println("Error en el método exec()");
         e.printStackTrace();
      }
   } // Fin del método main().
} // Fin de la clase.
```

Ejemplo 2: visualizar un programa CL Java

```
PGM
             DSPJVAPGM  CLSF('/QIBM/ProdData/Java400/com/ibm/as400/system/Hello.class') +
                          OUTPUT(*PRINT)
ENDPGM
```

Para obtener más información, consulte la sección Utilizar java.lang.Runtime.exec().

Ejemplo: llamar a un mandato CL con java.lang.Runtime.exec()

Este ejemplo muestra cómo ejecutar un mandato CL (Language Control) desde un programa Java. En este ejemplo, la clase Java ejecuta un mandato CL. El mandato CL utiliza el mandato CL Visualizar programa Java (DSPJVAPGM) para visualizar el programa asociado al archivo de clase Hello. La salida del mandato CL está en el archivo en spool QSYSPRT.

Al establecer la propiedad del sistema os400.runtime.exec como EXEC (que es el valor por omisión), los mandatos que pase a la función `Runtime.getRuntime().exec()` utilizarán el siguiente formato:

```
Runtime.getRuntime()Exec("system CLCOMMAND");
```

donde *CLCOMMAND* es el mandato CL que desea ejecutar.

Nota: Al establecer os400.runtime.exec como QSHELL, debe añadir una barra inclinada y comillas (\"). Por ejemplo, el mandato anterior tendrá este aspecto:

```
Runtime.getRuntime()Exec("system \"CLCOMMAND\"");
```

Para obtener más información sobre os400.runtime.exec y el efecto que tiene sobre el uso de java.lang.Runtime.exec(), consulte las siguientes páginas:

Utilizar java.lang.Runtime.exec()

Lista de propiedades del sistema Java

Ejemplo: Clase para llamar a un mandato CL

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

El código siguiente presupone que utilizará el valor por omisión de EXEC para la propiedad del sistema os400.runtime.exec.

```
import java.io.*;

public class CallCLCom
{
   public static void main(String[] args)
   {
       try
      {
         Process theProcess =
            Runtime.getRuntime().exec("system DSPJVAPGM CLSF('/com/ibm/as400/system/Hello.class')
                                      OUTPUT(*PRINT)");
      }
      catch(IOException e)
      {
         System.err.println("Error en el método exec()");
         e.printStackTrace();
      }
   } // Fin del método main().
} // Fin de la clase.
```

Para obtener más información, consulte la sección Utilizar java.lang.Runtime.exec().

Ejemplo: llamar a otro programa Java con java.lang.Runtime.exec()

Este ejemplo muestra cómo llamar a otro programa Java con java.lang.Runtime.exec(). Esta clase llama al programa Hello que se entrega como parte de IBM Developer Kit para Java. Cuando la clase Hello escribe en System.out, este programa obtiene un handle para acceder a la corriente y puede leer en ella.

Nota: para llamar al programa, utilice el intérprete de Qshell.

Ejemplo 1: clase CallHelloPgm

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
import java.io.*;

public class CallHelloPgm
{
   public static void main(String args[])
   {
      Process theProcess = null;
      BufferedReader inStream = null;

      System.out.println("CallHelloPgm.main() invocado");

      // Se llama a la clase Hello.
       try
      {
          theProcess = Runtime.getRuntime().exec("java com.ibm.as400.system.Hello");
      }
      catch(IOException e)
```

```
      {
         System.err.println("Error en el método exec()");
         e.printStackTrace();
      }

      // Se lee en la corriente de salida estándar del programa llamado.
       try
      {
         inStream = new BufferedReader(
                             new InputStreamReader( theProcess.getInputStream() ));
         System.out.println(inStream.readLine());
      }
      catch(IOException e)
      {
         System.err.println("Error en inStream.readLine()");
         e.printStackTrace();
      }

   } // Fin del método.

} // Fin de la clase.
```

Para obtener más información, consulte la sección Utilizar java.lang.Runtime.exec().

Ejemplo: llamar a Java desde C

Este es un ejemplo de programa C que utiliza la función system() para llamar al programa Java Hello.

Ejemplo: llamar a Java desde C

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
#include <stdlib.h>

 int main(void)
 {
   int result;

 /* La función system pasa la serie dada al procesador de mandatos CL
    para el proceso. */

   result = system("JAVA CLASS('com.ibm.as400.system.Hello')");
 }
```

Ejemplo: llamar a Java desde RPG

Este es un ejemplo de un programa RPG que utiliza la API QCMDEXC para llamar al programa Java Hello.

Ejemplo 1: llamar a Java desde RPG

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
     D*             SE DEFINEN LOS PARÁMETROS DE LA API QCMDEXC
     D*
     DCMDSTRING        S             25    INZ('JAVA CLASS(''com.ibm.as400.system.Hello'')')
     DCMDLENGTH        S             15P 5 INZ(25)
     D*             AHORA SE LLAMA A QCMDEXC CON EL MANDATO CL 'JAVA'
     C                   CALL      'QCMDEXC'
     C                   PARM                    CMDSTRING
     C                   PARM                    CMDLENGTH
     C*       La siguiente línea visualiza 'DID IT' después de salir de la
     C*       shell Java por medio de F3 o F12.
```

```
     C     'DID IT'      DSPLY
     C*       Se activa LR para salir del programa RPG
     C                   SETON                                        LR
     C
```

Ejemplo: utilizar corrientes de entrada y de salida para la comunicación entre procesos

Este ejemplo muestra cómo llamar a un programa C desde Java y utilizar corrientes de entrada y salida para la comunicación entre procesos. En este ejemplo, el programa C escribe una serie en su corriente de salida estándar y el programa Java la lee y la visualiza. En este ejemplo, se supone que se ha creado una biblioteca llamada JAVSAMPLIB y que en ella se ha creado el programa CSAMP1.

Nota: JAVSAMPLIB no se crea como parte del proceso de instalación del programa bajo licencia (LP) IBM Developer Kit, número 5722-JV1. Debe crearse de manera explícita.

Ejemplo 1: clase CallPgm

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
import java.io.*;

public class CallPgm
{
   public static void main(String args[])
   {
      Process theProcess = null;
      BufferedReader inStream = null;

      System.out.println("CallPgm.main() invocado");

      // se llama al programa CSAMP1
       try
      {
          theProcess = Runtime.getRuntime().exec(
                       "/QSYS.LIB/JAVSAMPLIB.LIB/CSAMP1.PGM");
      }
      catch(IOException e)
      {
         System.err.println("Error en el método exec()");
         e.printStackTrace();
      }

      // Se lee en la corriente de salida estándar del programa llamado.
       try
      {
         inStream = new BufferedReader(new InputStreamReader
                           (theProcess.getInputStream()));
         System.out.println(inStream.readLine());
      }
      catch(IOException e)
      {
         System.err.println("Error en inStream.readLine()");
         e.printStackTrace();
      }

   } // Fin del método.

} // Fin de la clase.
```

Ejemplo 2: programa C CSAMP1

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char* args[])
{
     /* Se convierte la serie a ASCII en tiempo de compilación */
   #pragma convert (819)
     printf("Se ha invocado el programa JAVSAMPLIB/CSAMP1\n");
#pragma convert(0)
     /* es posible que Stdout esté en el almacenamiento intermedio, así que
     este se vacía */

     fflush(stdout);

}
```

Para obtener más información, consulte el apartado Utilizar corrientes de entrada y de salida para la comunicación entre procesos.

Ejemplo: API de invocación Java

Este ejemplo sigue el paradigma de la API de invocación estándar. Por ejemplo, en él se hace lo siguiente:

- Se crea una máquina virtual Java mediante JNI_CreateJavaVM.
- Se utiliza la máquina virtual Java para buscar el archivo de clase que se desea ejecutar.
- Se busca el ID del método main de la clase.
- Se llama al método main de la clase.
- Se notifican los errores si se produce una excepción.

Al crear el programa, el programa de servicio QJVAJNI o QJVAJNI64 proporciona la función de API de invocación JNI_CreateJavaVM. JNI_CreateJavaVM crea la máquina virtual Java.

Nota: QJVAJNI64 es un nuevo programa de servicio para teraespacio/método nativo LLP64 y soporte de API de invocación.

Estos programas de servicio residen en el directorio de enlace del sistema y no es necesario identificarlos explícitamente en un mandato crear de lenguaje de control (CL). Por ejemplo, no identificaría explícitamente los programas de servicio mencionados anteriormente al utilizar el mandato Crear programa (CRTPGM) o el mandato Crear programa de servicio (CRTSRVPGM).

Una manera de ejecutar este programa es utilizar el siguiente mandato de lenguaje de control:

```
SBMJOB CMD(CALL PGM(YOURLIB/PGMNAME)) ALWMLTTHD(*YES)
```

Todo trabajo que cree una máquina virtual Java debe tener capacidad multihebra. La salida del programa principal, así como cualquier salida del programa, va a parar a los archivos en spool QPRINT. Estos archivos en spool resultan visibles si se utiliza el mandato de lenguaje de control (CL) Trabajar con trabajos sometidos (WRKSBMJOB) y se visualiza el trabajo iniciado utilizando el mandato CL Someter trabajo (SBMJOB).

Ejemplo: Utilizar la API de invocación Java

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
#define OS400_JVM_12
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <jni.h>

/* Especificar el pragma que provoca el almacenamiento de todas las series
 * literales del código fuente en ASCII (que, para las series
 * utilizadas, es equivalente a UTF-8)
 */

   #pragma convert (819)

/* Procedimiento: Oops
 *
 * Descripción: Rutina de ayuda a la que se llama cuando una función JNI
 *              devuelve un valor cero, indicando un error grave.
 *              Esta rutina informa de la excepción a stderr y
 *              finaliza la JVM abruptamente con una llamada a FatalError.
 *
 * Parámetros:  env -- JNIEnv* que debe utilizarse para llamadas JNI
 *              msg -- char* que señala a la descripción del error en UTF-8
 *
 * Nota:        El control no se devuelve después de la llamada a FatalError
 *              y no se devuelve desde este procedimiento.
 */

void Oops(JNIEnv* env, char *msg) {
    if ((*env)->ExceptionOccurred(env)) {
        (*env)->ExceptionDescribe(env);
    }
    (*env)->FatalError(env, msg);
}

/* Esta es la rutina "main" de este programa. */
int main (int argc, char *argv[])
{

    JavaVMInitArgs initArgs; /* Estructura de inicialización de la máquina virtual
                              * (VM) pasada por referencia a JNI_CreateJavaVM().
                              * Consulte jni.h obtener detalles
                              */
    JavaVM* myJVM;           /* Puntero de JavaVM establecido por llamada a
                              * JNI_CreateJavaVM */
    JNIEnv* myEnv;           /* Puntero de JNIEnv establecido por llamada a
                              * JNI_CreateJavaVM */
    char*   myClasspath;     /* 'Serie' de vía de acceso de clases modificable */
    jclass  myClass;         /* La clase a la que debe llamarse, 'NativeHello'. */
    jmethodID mainID;        /* El ID de método de su rutina 'main'. */
    jclass  stringClass;     /* Necesaria para crear el arg String[] arg para main */
    jobjectArray args;       /* El propio String[] */
    JavaVMOption options[1]; /* Matriz de opciones -- utilizar opciones para
                              * establecer vía de clases */
    int     fd0, fd1, fd2;   /* Descriptores de archivo para IO */

    /* Abrir los descriptores de archivo para que la ES funcione. */
     fd0 = open("/dev/null1", O_CREAT|O_TRUNC|O_RDWR,   S_IRUSR|S_IROTH);
     fd1 = open("/dev/null2", O_CREAT|O_TRUNC|O_WRONLY, S_IWUSR|S_IWOTH);
     fd2 = open("/dev/null3", O_CREAT|O_TRUNC|O_WRONLY, S_IWUSR|S_IWOTH);

    /*  Se establece el campo versión de los argumentos de inicialización para J2SDK v1.3. */
    initArgs.version = 0x00010002;
```

```
/*  Para utilizar J2SDK v1.4, establezca initArgs.version = 0x00010004; */

    /* Ahora, interesa especificar el directorio para que la clase
    * se ejecute en la vía de acceso de clases. Con Java2, la vía de
    * acceso de clases se pasa como opción.
    * Nota: el nombre de directorio debe especificarse en formato UTF-8. Así pues,
    * hay que envolver los bloques de código con sentencias #pragma convert.
    */
   options[0].optionString="-Djava.class.path=/CrtJvmExample";
/*Para utilizar J2SDK v1.4, sustituya '1.3' por '1.4'.
options[1].optionString="-Djava.version=1.3" */

   initArgs.options=options;  /* Se pasa la vía de acceso de clases que configuramos. */
   initArgs.nOptions = 2;     /* Se pasa la vía de acceso de clases y opciones de versión */

   /*  Crear la JVM -- un código de retorno no cero indica que se ha producido
    *  un error. Vuelva a EBCDIC y escriba un mensaje en stderr
    *  antes de salir del programa.
    */
   if (JNI_CreateJavaVM("myJVM, (void **)"myEnv, (void *)"initArgs)) {
  #pragma convert (0)
       fprintf(stderr, "No se ha podido crear la JVM\n");
  #pragma convert (819)
       exit(1);
   }

   /*  Se utiliza la JVM recién creada para buscar la clase de ejemplo,
    *  denominada 'NativeHello'.
    */
   myClass = (*myEnv)->FindClass(myEnv, "NativeHello");
   if (! myClass) {
       Oops(myEnv, "No se ha encontrado la clase 'NativeHello'");
   }

   /* Ahora, hay que obtener el identificador de método del punto de entrada
    * de 'main' de la clase.
    * Nota: La firma de 'main' es siempre la misma para cualquier
    *       clase llamada mediante el siguiente mandato java:
    *           "main" , "([Ljava/lang/String;)V"
    */
   mainID = (*myEnv)->GetStaticMethodID(myEnv,myClass,"main",
                                        "([Ljava/lang/String;)V");
   if (! mainID) {
       Oops(myEnv, "No se ha encontrado jmethodID de 'main'");
   }

   /* Obtener la jclass para String para crear la matriz
    * de String que debe pasarse a 'main'.
    */
   stringClass = (*myEnv)->FindClass(myEnv, "java/lang/String");
   if (! stringClass) {
       Oops(myEnv, "No se ha encontrado java/lang/String");
   }

   /*  Ahora, es necesario crear una matriz de series vacía,
    *  dado que main requiere una matriz de este tipo como parámetro.
    */
   args = (*myEnv)->NewObjectArray(myEnv,0,stringClass,0);
   if (! args) {
       Oops(myEnv, "No se ha podido crear la matriz de args");
   }

   /* Ahora, ya tiene el methodID de main y la clase, así que puede
    * llamar al método main.
    */
   (*myEnv)->CallStaticVoidMethod(myEnv,myClass,mainID,args);
```

```
    /* Se comprueba si hay errores. */
    if ((*myEnv)->ExceptionOccurred(myEnv)) {
            (*myEnv)->ExceptionDescribe(myEnv);
    }

    /* Finalmente, se destruye la máquina virtual Java creada. */
    (*myJVM)->DestroyJavaVM(myJVM);

    /* Eso es todo. */
    return 0;
}
```

Consulte el apartado API de llamada Java para obtener más información.

Ejemplo: método nativo IBM OS/400 PASE para Java

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

El ejemplo de método nativo IBM OS/400 PASE para Java llama a una instancia de un método nativo C que, a continuación, utiliza la interfaz Java nativa (JNI) para efectuar una llamada de retorno al código Java.

Para ver las versiones HTML de los archivos fuente de ejemplo, utilice los enlaces siguientes:

- PaseExample1.java
- PaseExample1.c

Antes de poder ejecutar el ejemplo de método nativo OS/400 PASE, debe llevar a cabo las tareas siguientes:

1. Bajar el código fuente de ejemplo a la estación de trabajo AIX
2. Preparar el código fuente de ejemplo
3. Preparar el servidor iSeries

Ejecutar el ejemplo de método nativo OS/400 PASE para Java

Tras completar las tareas anteriores, puede ejecutar el ejemplo. Utilice cualquiera de los mandatos siguientes para ejecutar el programa de ejemplo:

- Desde un indicador de mandatos de servidor iSeries:

  ```
  JAVA CLASS(PaseExample1) CLASSPATH('/home/example')
  ```

- Desde un indicador de mandatos de Qshell o una sesión de terminal OS/400 PASE:

  ```
  cd /home/example
  java PaseExample1
  ```

Ejemplos: utilizar la interfaz nativa Java para métodos nativos

Este programa es un ejemplo sencillo de la interfaz nativa Java (JNI) en el que se utiliza un método nativo C para visualizar "Hello, World". Para generar el archivo NativeHello.h, se utiliza la herramienta `javah` con el archivo de clase NativeHello. En este ejemplo, se supone que la implementación C de NativeHello forma parte de un programa de servicio llamado NATHELLO.

Nota: para poder ejecutar este ejemplo, la biblioteca en la que se encuentra el programa de servicio NATHELLO debe estar en la lista de bibliotecas.

Ejemplo 1: clase NativeHello

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
public class NativeHello {

    //  Se declara un campo de tipo 'String' (serie) en el objeto NativeHello.
    //  Se trata de un campo de 'instancia', de manera que cada objeto
    //  NativeHello contiene uno.
    public String theString;           // variable de instancia

    //  Se declara el método nativo propiamente dicho. El método nativo
    //  crea un nuevo objeto de tipo serie y coloca una referencia a dicho objeto
    //  en 'theString'
    public native void setTheString(); // método nativo para establecer la serie

    //  Se llama a este código de 'inicializador estático' antes de
    //  utilizar la clase por primera vez.
    static {

        //  Se intenta cargar la biblioteca de método nativo. Si no se
        //  encuentra, se escribe un mensaje en 'out' y se intenta una vía de acceso
        //  codificada de manera fija. Si esto falla, se sale.
            try {

            //  System.loadLibrary utiliza la lista de bibliotecas de iSeries de
            //  JDK 1.1 y utiliza la propiedad java.library.path o la variable
            //  de entorno LIBPATH en JDK1.2
              System.loadLibrary("NATHELLO");
            }

        catch (UnsatisfiedLinkError e1) {

            //  No se ha encontrado el programa de servicio.
            System.out.println
              ("No he encontrado NATHELLO *SRVPGM.");
            System.out.println ("(Lo intentaré con una vía codificada de manera fija)");

           try {

                //  System.load toma el formato completo de la vía de acceso del
                //  sistema de archivos integrado.
                System.load ("/qsys.lib/jniexample.lib/nathello.srvpgm");
                }

        catch (UnsatisfiedLinkError e2) {

            //  Si se llega a este punto, ya se ha acabado todo. Se escribe el
            //  mensaje y se sale.
            System.out.println
              ("<suspiro> no he encontrado NATHELLO *SRVPGM por ninguna parte. Adiós");
            System.exit(1);
            }
        }
    }

    //  Aquí está el código 'main' de la clase. Es lo que se ejecuta
    //  al entrar 'java NativeHello' en la línea de mandatos.
    public static void main(String argv[]){

        //  Se asigna un objeto NativeHello nuevo.
        NativeHello nh = new NativeHello();

        //  Se hace eco de la ubicación.
        System.out.println("(Java) Objeto NativeHello instanciado");
        System.out.println("(Java) campo serie es '" + nh.theString + "'");
        System.out.println("(Java) Llamar a método nativo para establecer la serie");

        //  Aquí está la llamada al método nativo.
        nh.setTheString();
```

```
        //  Ahora, se imprime el valor tras la llamada para mayor seguridad.
        System.out.println("(Java) Devuelto por el método nativo");
        System.out.println("(Java) campo serie es '" + nh.theString + "'");
        System.out.println("(Java) Eso es todo...");
    }
}
```

Ejemplo 2: archivo de cabecera NativeHello.h generado

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
/* NO EDITE ESTE ARCHIVO - está generado por máquina */
#include <jni.h>
/* Cabecera de la clase NativeHello */

#ifndef _Included_NativeHello
#define _Included_NativeHello
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Clase:     NativeHello
 * Método:    setTheString
 * Firma:     ()V
 */
JNIEXPORT void JNICALL Java_NativeHello_setTheString
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Este ejemplo de NativeHello.c muestra la implementación del método nativo en C, así como la manera de enlazar Java con los métodos nativos. No obstante, también señala las complicaciones que surgen por el hecho de que el servidor iSeries es internamente una máquina EBCDIC (código de intercambio decimal ampliado codificado en binario). Asimismo, indica las complicaciones derivadas de la falta de verdaderos elementos de internacionalización en JNI.

Estas razones, aunque no son una novedad en JNI, originan algunas diferencias exclusivas de iSeries en el código C escrito por el usuario. Conviene que recuerde que si escribe en stdout o en stderr o bien si lee en stdin, es probable que los datos estén codificados en el formato EBCDIC.

En código C, resulta fácil convertir la mayoría de las series literales, aquellas que contienen únicamente caracteres de 7 bits, al formato UTF-8 que requiere JNI. Para ello, hay que incluir las series literales entre sentencias pragma de conversión de página de códigos. No obstante, dado que es posible que se escriba información directamente en stdout o en stderr desde el código C, podría permitirse que algunos literales conservasen el formato EBCDIC.

Nota: las sentencias #pragma convert(0) convierten los datos de tipo carácter a EBCDIC. Las sentencias #pragma convert(819) convierten los datos de tipo carácter a ASCII (American Standard Code for Information Interchange). Estas sentencias realizan la conversión de los datos de tipo carácter del programa C en tiempo de compilación.

Ejemplo 3: implementación del método nativo NativeHello.c realizada por la clase Java NativeHello

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
#include <stdlib.h>       /* malloc, free, etcétera */
#include <stdio.h>        /* fprintf(), and so forth */
#include <qtqiconv.H>     /* iconv() interface */
#include <string.h>       /* memset(), and so forth */
#include "NativeHello.h"  /* generated by 'javah-jni' */

/*  La página de códigos de todas las series literales es la ISO-8859-1 Latin 1
(y en el caso de los caracteres de 7 bits, el formato es automáticamente UTF-8
también). */
#pragma convert(819)  /* se manejan todas las series literales como ASCII */

/*  Se notifica y se borra una excepción de JNI.  */
static void HandleError(JNIEnv*);

/*  Se imprime una serie UTF-8 en stderr con el ID de juego de caracteres */
(CCSID) del trabajo actual.  */
static void JobPrint(JNIEnv*, char*);

/*  Constantes que indican en qué dirección debe realizarse la conversión:  */
#define CONV_UTF2JOB 1
#define CONV_JOB2UTF 2

/*  Se convierte una serie del CCSID del trabajo a UTF-8 o viceversa.  */
int StringConvert(int direction, char *sourceStr, char *targetStr);

/*  Implementación de método nativo de 'setTheString()'.  */

JNIEXPORT void JNICALL Java_NativeHello_setTheString
(JNIEnv *env, jobject javaThis)
{
    jclass thisClass;     /* clase del objeto 'this' */
    jstring stringObject; /* serie nueva; se colocará en un campo de 'this' */
    jfieldID fid;         /* ID de campo necesario para actualizar campo de 'this' */
    jthrowable exception; /* excepción; se recupera con ExceptionOccurred */

    /*  Se escribe el estado en la consola.  */
    JobPrint(env, "( C ) En el método nativo\n");

    /* Se construye el nuevo objeto de tipo serie */
    if (! (stringObject = (*env)->NewStringUTF(env, "Hello, native world!")))
    {
        /*  Para casi todas las funciones de JNI, un valor de retorno nulo indica
        que ha habido un error y que se ha colocado una excepción en un punto en
        que 'ExceptionOccurred()' puede recuperarla. En este caso, el error sería
        normalmente muy grave, pero a efectos de este ejemplo, se sigue adelante
        y se captura el error y después se continúa.  */
        HandleError(env);
      return;
    }

    /* se obtiene la clase del objeto 'this', requisito para obtener fieldID  */
    if (! (thisClass = (*env)->GetObjectClass(env,javaThis)))
    {
        /*  Si GetObjectClass devuelve una clase nula, significa que ha
        habido un problema. En lugar de manejarlo, basta con usar return y
        saber que el retorno a Java 'lanza' automáticamente la excepción
        Java almacenada.  */
      return;
    }

    /*  Se obtiene fieldID para la actualización.  */
    if (! (fid = (*env)->GetFieldID(env,
                                    thisClass,
                                    "theString",
                                    "Ljava/lang/String;")))
    {
        /*  Si GetFieldID devuelve un fieldID nulo, significa que ha
```

```
        habido un problema. Hay que notificarlo desde aquí y borrarlo.
        No hay que modificar la serie.  */
        HandleError(env);
      return;
    }

    JobPrint(env, "( C ) Establecer el campo\n");

    /*  Se realiza la actualización propiamente dicha.
    Nota: SetObjectField es un ejemplo de interfaz que no devuelve
    un valor de retorno que pueda probarse. En este caso, resulta
    necesario llamar a ExceptionOccurred() para ver si ha habido
    un problema relacionado con el almacenamiento del valor  */
    (*env)->SetObjectField(env, javaThis, fid, stringObject);

    /*  Se comprueba si la actualización ha sido satisfactoria. Si no lo ha
    sido, se notifica el error.  */
    if ((*env)->ExceptionOccurred(env)) {

        /*  Se ha devuelto un objeto excepción no nulo desde
        ExceptionOccurred; así pues, hay un problema y debe notificarse
        el error. */
        HandleError(env);
    }

    JobPrint(env, "( C ) Volver del método nativo\n");
      return;

}

static void HandleError(JNIEnv *env)
{
    /*  Rutina sencilla para notificar y manejar una excepción.  */
    JobPrint(env, "( C  ) Error producido en llamada JNI: ");
    (*env)->ExceptionDescribe(env); /* se escriben los datos de la excepción en la consola */
    (*env)->ExceptionClear(env);    /* se borra la excepción pendiente */
}

static void JobPrint(JNIEnv *env, char *str)
{
    char   *jobStr;
    char   buf[512];
    size_t len;

    len = strlen(str);

    /*  Solo se imprime la serie no vacía. */
    if (len) {
        jobStr = (len >= 512) ? malloc(len+1) : &buf;
        if (! StringConvert(CONV_UTF2JOB, str, jobStr))
            (*env)->FatalError
              (env,"ERROR en JobPrint: imposible convertir UTF2JOB");
        fprintf(stderr, jobStr);
        if (len >= 512) free(jobStr);
    }
}

int StringConvert(int direction, char *sourceStr, char *targetStr)
{
    QtqCode_T source, target;     /* parámetros para instanciar iconv           */
    size_t    sStrLen, tStrLen;   /* copias locales de las longitudes de serie  */
    iconv_t   ourConverter;       /* descriptor de conversión propiamente dicho */
    int       iconvRC;            /* código de retorno de la conversión         */
    size_t    originalLen;        /* longitud original de sourceStr             */

    /*  Se realizan copias locales de los tamaños de entrada y salida que se
    inicializan según el tamaño de la serie de entrada. iconv() requiere que
```

```
    los parámetros de longitud se pasen por dirección (es decir, como int*). */
    originalLen = sStrLen = tStrLen = strlen(sourceStr);

    /*  Se inicializan los parámetros de QtqIconvOpen() a cero.  */
    memset(&source,0x00,sizeof(source));
    memset(&target,0x00,sizeof(target));

    /*  En función del parámetro direction, se establece el CCSID
    ORIGEN (source) o DESTINO (target) en ISO 8859-1 Latin.  */
    if (CONV_UTF2JOB == direction ) {
        source.CCSID = 819;
    }
    else {
        target.CCSID = 819;
    }

    /*  Se crea el objeto convertidor iconv_t.  */
    ourConverter = QtqIconvOpen(&target,&source);

    /*  Se comprueba que el convertidor es válido; si no es así, se devuelve 0.  */
    if (-1 == ourConverter.return_value) return 0;

    /*  Se realiza la conversión. */
    iconvRC = iconv(ourConverter,
                    (char**) &sourceStr,
                    &sStrLen,
                    &targetStr,
                    &tStrLen);

    /*  Si la conversión falla, se devuelve un cero.  */
    if (0 != iconvRC ) return 0;

    /*  Se cierra el descriptor de conversión.  */
    iconv_close(ourConverter);

    /*  Se devuelve targetStr que señala hacia el carácter que está justo
    detrás del último carácter convertido; así pues, se establece
    el nulo en este punto.  */
    *targetStr = '\0';

    /*  Se devuelve el número de caracteres procesados. */
    return originalLen-tStrLen;

}

#pragma convert(0)
```

Consulte la sección Utilizar la interfaz nativa Java para métodos nativos para obtener información previa.

Ejemplo: utilizar sockets para la comunicación entre procesos

En este ejemplo se utilizan sockets para la comunicación entre un programa Java y un programa C. El programa C, que está a la escucha en un socket, debe iniciarse primero. Una vez que el programa Java se ha conectado al socket, el programa C le envía una serie utilizando la conexión por socket. La serie que se envía desde el programa C es una serie ASCII (American Standard Code for Information Interchange) de la página de códigos 819.

El programa Java debe iniciarse con el mandato `java TalkToC xxxxx nnnn` en la línea de mandatos del intérprete de Qshell o en otra plataforma Java. O bien, especifique `JAVA TALKTOC PARM(xxxxx nnnn)` en la línea de mandatos de iSeries para iniciar el programa Java. `xxxxx` es el nombre de dominio o dirección IP (Protocolo Internet) del sistema en el que se ejecuta el programa C. `nnnn` es el número de puerto del socket utilizado por el programa C. Este número de puerto también se tiene que utilizar como primer parámetro de la llamada al programa C.

Ejemplo 1: clase cliente TalkToC

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
import java.net.*;
import java.io.*;

class TalkToC
{
   private String host = null;
   private int port = -999;
   private Socket socket = null;
   private BufferedReader inStream = null;


   public static void main(String[] args)
   {
      TalkToC caller = new TalkToC();
      caller.host = args[0];
      caller.port = new Integer(args[1]).intValue();
      caller.setUp();
      caller.converse();
      caller.cleanUp();

   } // Fin del método main().

   public void setUp()
   {
      System.out.println("TalkToC.setUp() invocado");

       try
      {
         socket = new Socket(host, port);
         inStream = new BufferedReader(new InputStreamReader(
                                     socket.getInputStream()));
      }
      catch(UnknownHostException e)
      {
         System.err.println("No se puede encontrar el sistema principal llamado: " + host);
         e.printStackTrace();
         System.exit(-1);
      }
      catch(IOException e)
      {
         System.err.println("No se ha podido establecer conexión para " + host);
         e.printStackTrace();
         System.exit(-1);
      }
   } // Fin del método setUp().

   public void converse()
   {
      System.out.println("TalkToC.converse() invocado");

      if (socket != null && inStream != null)
      {
       try
         {
         System.out.println(inStream.readLine());
         }
      catch(IOException e)
         {
            System.err.println("Error de conversación con sistema principal " + host);
         e.printStackTrace();
         }
```

```
      } // Fin de if.

   }  // Fin del método converse().

   public void cleanUp()
   {
       try
      {

         if(inStream != null)
         {
            inStream.close();
         }
         if(socket != null)
         {
            socket.close();
         }
      } // Fin de try.
      catch(IOException e)
      {
         System.err.println("Error de borrado");
         e.printStackTrace();
         System.exit(-1);
      }
   } // Fin del método cleanUp().

} // Fin de la clase TalkToC.
```

SockServ.C se inicia pasando un parámetro correspondiente al número de puerto. Por ejemplo, `CALL SockServ '2001'`.

Ejemplo 2: programa servidor SockServ.C

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <unistd.h>
#include <sys/time.h>

void main(int argc, char* argv[])
{
   int    portNum = atoi(argv[1]);
   int    server;
   int    client;
   int    address_len;
   int    sendrc;
   int    bndrc;
   char*  greeting;
   struct sockaddr_in  local_Address;
   address_len = sizeof(local_Address);

   memset(&local_Address,0x00,sizeof(local_Address));
   local_Address.sin_family = AF_INET;
   local_Address.sin_port = htons(portNum);
   local_Address.sin_addr.s_addr = htonl(INADDR_ANY);

   #pragma convert (819)
   greeting = "Este es un mensaje del servidor de sockets C.";
   #pragma convert (0)
```

```
   /*  Se asigna el socket.  */
   if((server = socket(AF_INET, SOCK_STREAM, 0))<0)
   {
      printf("anomalía en la asignación de socket\n");
     perror(NULL);
     exit(-1);
   }

   /* Se realiza el enlace (bind). */
   if((bndrc=bind(server,(struct sockaddr*)&local_Address, address_len))<0)
   {
     printf("Enlace fallido\n");
     perror(NULL);
     exit(-1);
   }

   /* Se invoca el método listen. */
   listen(server, 1);

   /* En espera de la petición del cliente. */
   if((client = accept(server,(struct sockaddr*)NULL, 0))<0)
   {
     printf("aceptar ha fallado\n");
     perror(NULL);
     exit(-1);
   }


   /* Se envía un saludo (greeting) al cliente. */
   if((sendrc = send(client, greeting, strlen(greeting),0))<0)
   {
      printf("Envío fallido\n");
     perror(NULL);
     exit(-1);
   }

   close(client);
   close(server);

}
```

Para obtener más información, consulte la sección Utilizar sockets para la comunicación entre procesos.

Ejemplo: ejecutar Java Performance Data Converter

Para ejecutar Java Performance Data Converter (JPDC), puede utilizar la línea de mandatos de iSeries o el entorno Qshell.

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

Con la línea de mandatos de iSeries:

1. Especifique el mandato Ejecutar (RUNJVA) o JAVA en la línea de mandatos de iSeries.
2. Especifique `com.ibm.as400.jpdc.JPDC` en la línea del parámetro clase.
3. Especifique `general defpex midir/miarch midirebdr` en la línea de parámetro.
4. Especifique '`/QIBM/ProdData/Java400/ext/JPDC.jar`' en la línea del parámetro vía de acceso de clases.

   Nota: puede omitir la vía de acceso de clases si la serie '`/QIBM/ProdData/Java400/ext/JPDC.jar`' está en la variable de entorno CLASSPATH. Para añadir esta serie a la variable de entorno CLASSPATH, puede utilizar el mandato Añadir variable de entorno (ADDENVVAR), Cambiar variable de entorno (CHGENVVAR) o Trabajar con variable de entorno (WRKENVVAR).

Con el entorno Qshell:

1. Especifique el mandato Arrancar Qshell (STRQSH) para iniciar el intérprete de Qshell.
2. Especifique lo siguiente en la línea de mandatos:

   ```
   java -classpath /QIBM/ProdData/Java400/ext/JPDC.jar com.ibm.as400/jpdc/JPDC
   jinsight pexdfn mydir/myfile myrdbdire
   ```

   Nota: puede omitir la vía de acceso de clases si se ha añadido la serie '/QIBM/ProdData/Java400/ext/JPDC.jar' al entorno actual. Para añadir esta serie al entorno actual, puede utilizar el mandato ADDENVVAR, CHGENVVAR o WRKENVVAR.

Para obtener información previa, consulte el apartado Ejecutar Java Performance Data Converter.

Ejemplos: cambiar el código Java para que utilice fábricas de sockets de cliente

Estos ejemplos muestran cómo cambiar una clase de socket simple, denominada simpleSocketClient, de forma que utilice fábricas de sockets para crear todos los sockets. El primer ejemplo muestra la clase simpleSocketClient sin fábricas de sockets. El segundo muestra la clase simpleSocketClient con fábricas de sockets. En el segundo ejemplo, simpleSocketClient cambia de nombre y pasa a llamarse factorySocketClient.

Ejemplo 1: programa Socket Client sin fábricas de sockets

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
/* Programa Simple Socket Client */

import java.net.*;
import java.io.*;

public class simpleSocketClient {
  public static void main (String args[]) throws IOException {

    int     serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java simpleSocketClient serverHost serverPort");
      System.out.println("serverPort toma por omisión el valor 3000 si no se especifica.");
      return;
    }
    if (args.length == 2)
      serverPort = new Integer(args[1]).intValue();

    System.out.println("Conectando a sistema principal " + args[0] + " en el puerto " +
                       serverPort);

    // Se crea el socket y se conecta con el servidor.
    Socket  s = new Socket(args[0], serverPort);
    .
    .
    .

    // El resto del programa sigue a partir de aquí.
```

Ejemplo 2: programa Simple Socket Client con fábricas de sockets

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
/* Programa Simple Socket Factory Client */

// Observe que se importa javax.net.* para tomar la clase SocketFactory.
```

```
import javax.net.*;
import java.net.*;
import java.io.*;

public class factorySocketClient {
  public static void main (String args[]) throws IOException {

    int     serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java factorySocketClient serverHost serverPort");
      System.out.println("serverPort toma por omisión el valor 3000 si no se especifica.");
      return;
    }
    if (args.length == 2)
      serverPort = new Integer(args[1]).intValue();

    System.out.println("Conectando a sistema principal " + args[0] + " en el puerto " +
                       serverPort);

    // Se cambia el programa simpleSocketClient original para crear una
    // fábrica de sockets, que luego se utiliza para crear sockets.

    SocketFactory socketFactory = SocketFactory.getDefault();

    // Ahora la fábrica crea el socket. Es el último cambio
    // que se realiza en el programa simpleSocketClient original.

    Socket  s = socketFactory.createSocket(args[0], serverPort);
    .
    .
    .

    // El resto del programa sigue a partir de aquí.
```

Para obtener información previa, consulte la sección Cambiar el código Java para que utilice fábricas de sockets.

Ejemplos: cambiar el código Java para que utilice fábricas de sockets de servidor

Estos ejemplos muestran cómo cambiar una clase de socket simple, denominada simpleSocketServer, de forma que utilice fábricas de sockets para crear todos los sockets. El primer ejemplo muestra la clase simpleSocketServer sin fábricas de sockets. El segundo muestra la clase simpleSocketServer con fábricas de sockets. En el segundo ejemplo, simpleSocketServer cambia de nombre y pasa a llamarse factorySocketServer.

Ejemplo 1: programa Socket Server sin fábricas de sockets

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
/* Archivo simpleSocketServer.java*/

import java.net.*;
import java.io.*;

public class simpleSocketServer {
  public static void main (String args[]) throws IOException {

    int     serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java simpleSocketServer serverPort");
      System.out.println("Toma por omisión el puerto 3000 porque no se ha especificado serverPort.");
```

```
    }
     else
        serverPort = new Integer(args[0]).intValue();

    System.out.println("Estableciendo socket de servidor en el puerto " + serverPort);

    ServerSocket    serverSocket =
      new ServerSocket(serverPort);

    // Un servidor real manejaría más de un único cliente así...

    Socket s = serverSocket.accept();
    BufferedInputStream is = new BufferedInputStream(s.getInputStream());
    BufferedOutputStream os = new BufferedOutputStream(s.getOutputStream());

    // Este servidor tan solo se hace eco de lo que se le envía...

    byte buffer[] = new byte[4096];

    int bytesRead;

    // Se lee hasta que se devuelve "eof".
    while ((bytesRead = is.read(buffer)) > 0) {
      os.write(buffer, 0, bytesRead); // Se escribe lo recibido.
      os.flush();    // Se vacía el almacenamiento intermedio de salida.
    }


   s.close();
    serverSocket.close();
  }       // Fin de main().

}       // Fin de la definición de clase.
```

Ejemplo 2: programa Simple Socket Server con fábricas de sockets

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
/* Archivo factorySocketServer.java */

// Hay que importar javax.net para tomar la clase ServerSocketFactory
import javax.net.*;
import java.net.*;
import java.io.*;

public class factorySocketServer {
  public static void main (String args[]) throws IOException {

    int     serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java simpleSocketServer serverPort");
      System.out.println("Toma por omisión el puerto 3000 porque no se ha especificado serverPort.");
    }
     else
        serverPort = new Integer(args[0]).intValue();

    System.out.println("Estableciendo socket de servidor en el puerto " + serverPort);

    // Se cambia la clase simpleSocketServer original para que utilice
    // ServerSocketFactory con el fin de crear sockets de servidor.
    ServerSocketFactory serverSocketFactory =
      ServerSocketFactory.getDefault();
    // Ahora, la fábrica ha de crear el socket de servidor. Es
    // el último cambio que se realiza en el programa original.
    ServerSocket    serverSocket =
```

```
      serverSocketFactory.createServerSocket(serverPort);

    // Un servidor real manejaría más de un único cliente así...

    Socket s = serverSocket.accept();
    BufferedInputStream is = new BufferedInputStream(s.getInputStream());
    BufferedOutputStream os = new BufferedOutputStream(s.getOutputStream());

    // Este servidor tan solo se hace eco de lo que se le envía...

    byte buffer[] = new byte[4096];

    int bytesRead;

    while ((bytesRead = is.read(buffer)) > 0) {
      os.write(buffer, 0, bytesRead);
      os.flush();
    }


   s.close();
    serverSocket.close();
  }

}
```

Para obtener información previa, consulte la sección Cambiar el código Java para que utilice fábricas de sockets.

Ejemplos: cambiar el cliente Java para que utilice la capa de sockets segura

Estos ejemplos muestran cómo cambiar una clase, denominada factorySocketClient, de forma que utilice SSL (capa de sockets segura).

El primer ejemplo muestra la clase factorySocketClient sin SSL. El segundo muestra la misma clase, que cambia de nombre y pasa a llamarse factorySSLSocketClient, con SSL.

Ejemplo 1: clase factorySocketClient simple sin soporte SSL

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
/* Programa Simple Socket Factory Client */

import javax.net.*;
import java.net.*;
import java.io.*;

public class factorySocketClient {
  public static void main (String args[]) throws IOException {

    int     serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java factorySocketClient serverHost serverPort");
      System.out.println("serverPort toma por omisión el valor 3000 si no se especifica.");
      return;
    }
    if (args.length == 2)
      serverPort = new Integer(args[1]).intValue();

    System.out.println("Conectando a sistema principal " + args[0] + " en el puerto " +
                       serverPort);
```

```
    SocketFactory socketFactory = SocketFactory.getDefault();

    Socket  s = socketFactory.createSocket(args[0], serverPort);
    .
    .
    .

    // El resto del programa sigue a partir de aquí.
```

Ejemplo 2: clase factorySocketClient simple con soporte SSL

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
// Observe que importamos javax.net.ssl.* para tomar el soporte SSL
import javax.net.ssl.*;
import javax.net.*;
import java.net.*;
import java.io.*;

public class factorySSLSocketClient {
  public static void main (String args[]) throws IOException {

    int     serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java factorySSLSocketClient serverHost serverPort");
      System.out.println("serverPort toma por omisión el valor 3000 si no se especifica.");
      return;
    }
    if (args.length == 2)
      serverPort = new Integer(args[1]).intValue();

    System.out.println("Conectando a sistema principal " + args[0] + " en el puerto " +
                       serverPort);

    // Cambiamos esto para crear SSLSocketFactory en lugar de SocketFactory.
    SocketFactory socketFactory = SSLSocketFactory.getDefault();

    // No es necesario cambiar nada más.
    // ¡Ahí está la gracia de utilizar fábricas!
    Socket  s = socketFactory.createSocket(args[0], serverPort);
    .
    .
    .

    // El resto del programa sigue a partir de aquí.
```

Para obtener información previa, consulte la sección Cambiar el código Java para que utilice la capa de sockets segura.

Ejemplos: cambiar el servidor Java para que utilice la capa de sockets segura

Estos ejemplos muestran cómo cambiar una clase, denominada factorySocketServer, de forma que utilice SSL (capa de sockets segura).

El primer ejemplo muestra la clase factorySocketServer sin SSL. El segundo muestra la misma clase, que cambia de nombre y pasa a llamarse factorySSLSocketServer, con SSL.

Ejemplo 1: clase factorySocketServer simple sin soporte SSL

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
/* Archivo factorySocketServer.java */
// Hay que importar javax.net para tomar la clase ServerSocketFactory
import javax.net.*;
import java.net.*;
import java.io.*;

public class factorySocketServer {
  public static void main (String args[]) throws IOException {

    int     serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java simpleSocketServer serverPort");
      System.out.println("Toma por omisión el puerto 3000 porque no se ha especificado serverPort.");
    }
     else
        serverPort = new Integer(args[0]).intValue();

    System.out.println("Estableciendo socket de servidor en el puerto " + serverPort);

    // Se cambia la clase simpleSocketServer original para que utilice
    // ServerSocketFactory con el fin de crear sockets de servidor.
    ServerSocketFactory serverSocketFactory =
      ServerSocketFactory.getDefault();
    // Ahora, la fábrica ha de crear el socket de servidor. Es
    // el último cambio que se realiza en el programa original.
    ServerSocket    serverSocket =
      serverSocketFactory.createServerSocket(serverPort);

    // Un servidor real manejaría más de un único cliente así...

    Socket s = serverSocket.accept();
    BufferedInputStream is = new BufferedInputStream(s.getInputStream());
    BufferedOutputStream os = new BufferedOutputStream(s.getOutputStream());

    // Este servidor tan solo se hace eco de lo que se le envía.

    byte buffer[] = new byte[4096];

    int bytesRead;

    while ((bytesRead = is.read(buffer)) > 0) {
      os.write(buffer, 0, bytesRead);
      os.flush();
    }

   s.close();
    serverSocket.close();
  }
}
```

Ejemplo 2: clase factorySocketServer simple con soporte SSL

Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

```
/* Archivo factorySocketServer.java */

// Hay que importar javax.net para tomar la clase ServerSocketFactory
import javax.net.*;
import java.net.*;
import java.io.*;

public class factorySocketServer {
  public static void main (String args[]) throws IOException {

    int     serverPort = 3000;
```

```
    if (args.length < 1) {
      System.out.println("java simpleSocketServer serverPort");
      System.out.println("Toma por omisión el puerto 3000 porque no se ha especificado serverPort.");
    }
     else
        serverPort = new Integer(args[0]).intValue();

    System.out.println("Estableciendo socket de servidor en el puerto " + serverPort);

    // Se cambia la clase simpleSocketServer original para que utilice
    // ServerSocketFactory con el fin de crear sockets de servidor.
    ServerSocketFactory serverSocketFactory =
      ServerSocketFactory.getDefault();
    // Ahora, la fábrica ha de crear el socket de servidor. Es
    // el último cambio que se realiza en el programa original.
    ServerSocket    serverSocket =
      serverSocketFactory.createServerSocket(serverPort);

    // Un servidor real manejaría más de un único cliente así...

    Socket s = serverSocket.accept();
    BufferedInputStream is = new BufferedInputStream(s.getInputStream());
    BufferedOutputStream os = new BufferedOutputStream(s.getOutputStream());

    // Este servidor tan solo se hace eco de lo que se le envía.

    byte buffer[] = new byte[4096];

    int bytesRead;

    while ((bytesRead = is.read(buffer)) > 0) {
      os.write(buffer, 0, bytesRead);
      os.flush();
    }

   s.close();
    serverSocket.close();
  }
}
```

Para obtener información previa, consulte la sección Cambiar el código Java para que utilice la capa de sockets segura.

Resolución de problemas de IBM Developer Kit para Java

Si detecta problemas al utilizar IBM Developer Kit para Java, lleve a cabo cualquiera de estos pasos para determinar el origen del problema.

- Puede observar algunas limitaciones al utilizar IBM Developer Kit para Java. En este tema se señalan las limitaciones, las restricciones o los comportamientos propios que se conocen.
- Busque el archivo de anotaciones del trabajo que ha ejecutado el mandato Java, así como el del trabajo inmediato de proceso por lotes (BCI) en el que se ha ejecutado el programa Java con el fin de analizar la causa de la anomalía.
- Reúna los datos de utilidad para el informe autorizado de análisis de programa (APAR).
- Aplique los arreglos temporales del programa (PTF).
- Averigüe cómo obtener soporte técnico si detecta un defecto potencial en IBM Developer Kit para Java.

Si el rendimiento del programa se degrada al ejecutarse durante mucho más tiempo, es posible que haya codificado una fuga de memoria erróneamente. Puede utilizar JavaWatcher, un componente de iSeries iDoctor, como ayuda para depurar el programa y localizar las posibles fugas de memoria.

Para obtener más información, consulte JavaWatcher.

Limitaciones

Al emplear IBM Developer Kit para Java, puede que observe que la forma de utilización presenta ciertas limitaciones. En esta lista se señalan las limitaciones, las restricciones o los comportamientos propios que se conocen.

- Cuando se carga una clase y no se encuentran las superclases de la misma, el error indica que no se ha encontrado la clase original. Por ejemplo, si la clase B amplía la clase A y no se encuentra la segunda al cargar la primera, el error indica que no se ha encontrado la clase B, aunque en realidad sea la clase A la que no se ha encontrado. Si aparece un error que indica que no se ha encontrado una clase, compruebe que la clase y todas sus superclases están en la vía de acceso de clases. Esto también es válido para las interfaces implementadas por la clase que se esté cargando.
- El almacenamiento dinámico de la recogida de basura está limitado a 240 GB.
- No hay un límite explícito para el número de objetos construidos.
- El parámetro backlog de java.net puede tener en un servidor iSeries un comportamiento distinto del que tiene en otras plataformas. Por ejemplo:
  - Si el parámetro backlog es 0 ó 1 en Listen:
    - Listen(0) significa que se permite una conexión pendiente; no inhabilita un socket.
    - Listen(1) significa que se permite una conexión pendiente y equivale a Listen(0).
  - Si el parámetro backlog es mayor que 1 en Listen:
    - Esto permite que haya muchas peticiones pendientes en la cola de Listen. Si llega una nueva petición de conexión y la cola está al límite, se suprimirá una de las peticiones pendientes.
- Sólo puede utilizar la máquina virtual Java, independientemente de la versión de JDK que esté utilizando, en entornos con capacidad multihebra (es decir, seguros en ejecución multihebra). El servidor iSeries es seguro en ejecución multihebra, pero no así algunos sistemas de archivos. Para obtener una lista de los sistemas de archivos no seguros en ejecución multihebra, consulte el apartado Sistema de archivos integrado.
- El soporte de Protocolo Internet versión 6 (IPv6) no está totalmente implementado, por lo que podrían producirse algunos efectos colaterales. Para obtener más información, consulte Sockets.

Buscar las anotaciones de trabajo para el análisis de problemas Java

Para analizar las causas de una anomalía de Java, utilice las anotaciones de trabajo del trabajo que ha ejecutado el mandato Java y las del trabajo inmediato de proceso por lotes (BCI) en el que se ha ejecutado el programa Java. Ambos pueden contener información de error importante.

Existen dos formas de buscar el archivo de anotaciones del trabajo BCI. Puede buscar el nombre del trabajo BCI que figura en las anotaciones del trabajo que ha ejecutado el mandato Java. A continuación, utilice el nombre de trabajo para buscar el archivo de anotaciones del trabajo BCI.

También puede buscar el archivo de anotaciones del trabajo BCI realizando los pasos siguientes:

1. Entre el mandato Trabajar con trabajos sometidos (WRKSBMJOB) en la línea de mandatos de iSeries.
2. Vaya al final de la lista.
3. Busque el último trabajo de la lista; se llama QJVACMDSRV.

4. Entre la opción 8 (Trabajar con archivos en spool) para el trabajo.
5. Se visualizará un archivo llamado QPJOBLOG.
6. Pulse F11 para ver la vista 2 de los archivos en spool.
7. Verifique que la fecha y la hora coinciden con la fecha y la hora en que se ha producido la anomalía.

   Nota: si la fecha y la hora no coinciden con la fecha y la hora en que usted ha finalizado la sesión, siga buscando en la lista de trabajos sometidos. Intente localizar un archivo de anotaciones llamado QJVACMDSRV cuya fecha y hora coincida con el momento en que ha finalizado la sesión.

Si no puede hallar ningún archivo de anotaciones para el trabajo BCI, es posible que no se haya generado ninguno. Esto ocurre si ha establecido un valor ENDSEP demasiado alto para la descripción de trabajo QDFTJOBD o si el valor LOG de la descripción de trabajo QDFTJOBD especifica *NOLIST. Compruebe estos valores y cámbielos para que se genere un archivo de anotaciones para el trabajo BCI.

Para generar un archivo de anotaciones para el trabajo que ha ejecutado el mandato Ejecutar Java (RUNJVA), lleve a cabo los siguientes pasos:

1. Entre SIGNOFF *LIST.
2. A continuación, vuelva a iniciar la sesión.
3. Entre el mandato Trabajar con archivos en spool (WRKSPLF) en la línea de mandatos de iSeries.
4. Vaya al final de la lista.
5. Busque un archivo llamado QPJOBLOG.
6. Pulse F11.
7. Verifique que la fecha y la hora coincidan con la fecha y la hora en que ha entrado el mandato de fin de sesión.

   Nota: si la fecha y la hora no coinciden con la fecha y la hora en que usted ha finalizado la sesión, siga buscando en la lista de trabajos sometidos. Intente localizar un archivo de anotaciones llamado QJVACMDSRV cuya fecha y hora coincida con el momento en que ha finalizado la sesión.

Recoger datos para el análisis de problemas de Java

Para reunir datos con el fin de elaborar un informe autorizado de análisis de programa (APAR), siga estos pasos:

1. Incluya una descripción completa del problema.
2. Guarde el archivo de clase Java que ha originado el problema al ejecutarse.
3. Puede utilizar el mandato SAV para guardar objetos del sistema de archivos integrado. Es posible que necesite guardar otros archivos de clase que este programa debe ejecutar. Asimismo, tal vez le interese guardar y enviar un directorio completo con el fin de que IBM lo utilice, si fuese necesario, cuando intente reproducir el problema. He aquí un ejemplo de cómo guardar un directorio entero.

   Ejemplo: guardar un directorio

   Nota: lea el apartado Declaración de limitación de responsabilidad sobre el código de ejemplo para obtener información legal importante.

   ```
   SAV DEV('/QSYS.LIB/TAP01.DEVD') OBJ(('/midir'))
   ```

   Si es posible, guarde los archivos fuente de las clases Java implicadas en el problema. Le servirán de ayuda a IBM a la hora de reproducir y analizar el problema.
4. Guarde los programas de servicio que contienen los métodos nativos que se necesitan para ejecutar el programa.
5. Guarde los archivos de datos que se necesitan para ejecutar el programa Java.
6. Añada una explicación completa de cómo reproducir el problema. Incluirá lo siguiente:
   - El valor de la variable de entorno CLASSPATH.
   - La descripción del mandato Java ejecutado.
   - Una indicación de cómo debe responderse a la entrada que necesite el programa.

7. Incluya las anotaciones del código interno vertical bajo licencia (VLIC) que se hayan realizado alrededor del momento en que produjo la anomalía.
8. Añada los archivos de anotaciones del trabajo interactivo y del trabajo BCI en el que se ejecutaba la máquina virtual Java.

Obtener soporte para IBM Developer Kit para Java

Los servicios de soporte de IBM Developer Kit para Java se prestan en los términos y las condiciones habituales de los productos de software de iSeries. Dichos servicios incluyen servicio técnico de programas, atención telefónica y servicios de consultoría. Utilice la información en línea que se suministra en la página de presentación de IBM iSeries

, bajo el tema "Soporte" para obtener más información. Emplee Servicios de soporte de IBM para 5722-JV1 (IBM Developer Kit para Java). O bien, póngase en contacto con el representante local de IBM.

Es posible que, por indicación de IBM, deba obtener un nivel más reciente de IBM Developer Kit para Java con el fin de poder recibir el servicio técnico de programas de forma continuada. Para obtener más información, consulte el apartado Soporte para varios Java Development Kits (JDK).

Los defectos que pueda presentar el programa IBM Developer Kit para Java pueden resolverse por medio del servicio técnico de programas o la atención telefónica. Las cuestiones relacionadas con la depuración o la programación de la aplicación deben resolverse a través de los servicios de consultoría.

Las cuestiones relacionadas con llamadas a la interfaz de programa de aplicación (API) de IBM Developer Kit para Java deben remitirse a los servicios de consultoría, a menos que:

1. Se trate claramente de un defecto de la API Java, que pueda demostrarse por medio de su reproducción en un programa de relativa simplicidad.
2. Sea una cuestión que requiere una aclaración de lo explicado en la documentación.
3. Sea una cuestión referente a la ubicación de los ejemplos o la documentación.

Los servicios de consultoría comprenden también el prestar ayuda de programación. Esto incluye los ejemplos de programa que se facilitan en el producto programa bajo licencia (LP) IBM Developer Kit para Java. Puede haber más ejemplos disponibles en Internet en la página de presentación de IBM iSeries

no basados en soporte.

El programa bajo licencia IBM Developer Kit para Java incluye información para la resolución de problemas. Si usted cree que existe un defecto potencial en la API de IBM Developer Kit para Java, será necesario que suministre un programa simple que demuestre la existencia del error.

Información relacionada para IBM Developer Kit para Java

La siguiente información de consulta en Javadoc está relacionada con IBM Developer Kit para Java:

- Javadoc JAAS específico de iSeries
- Especificación de la API JAAS
- Especificación de la API Java 2 Platform, Standard Edition, v1.4

La siguiente información de consulta está relacionada con IBM Developer Kit para Java:

- Java Naming and Directory Interface
- JavaMail
- JavaPrintService

Apéndice. Avisos

Esta información se ha desarrollado para productos y servicios ofrecidos en los EE.UU.

IBM puede no ofrecer los productos, servicios o características tratados en este documento en otros países. Póngase en contacto con el representante local de IBM que le informará sobre los productos y servicios disponibles actualmente en su área. Las referencias hechas a productos, programas o servicios de IBM no pretenden afirmar ni dar a entender que únicamente puedan utilizarse dichos productos, programas o servicios de IBM. Puede utilizarse en su lugar cualquier otro producto, programa o servicio funcionalmente equivalente que no vulnere ninguno de los derechos de propiedad intelectual de IBM. No obstante, es responsabilidad del usuario evaluar y verificar el funcionamiento de cualquier producto, programa o servicio que no sea de IBM.

IBM puede tener patentes o solicitudes de patente pendientes de aprobación que cubran temas descritos en este documento. La posesión de este documento no le otorga ninguna licencia sobre dichas patentes. Puede enviar consultas sobre las licencias, por escrito, a:

```
IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY  10594-1785
Estados Unidos
```

Para consultas sobre licencias relativas a la información de doble byte (DBCS), póngase en contacto con el departamento de propiedad intelectual de IBM en su país o envíe las consultas, por escrito, a:

```
IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japón
```

El párrafo siguiente no se aplica al Reino Unido ni a ningún otro país en que dichas disposiciones entren en contradicción con las leyes locales: INTERNATIONAL BUSINESS MACHINES CORPORATION PROPORCIONA ESTA PUBLICACIÓN "TAL CUAL" SIN GARANTÍA DE NINGÚN TIPO, NI EXPLÍCITA NI IMPLÍCITA, INCLUYENDO, PERO NO LIMITÁNDOSE, A LAS GARANTÍAS IMPLÍCITAS DE NO VULNERABILIDAD, COMERCIALIZACIÓN O ADECUACIÓN A UN PROPÓSITO DETERMINADO. Algunas legislaciones no contemplan la declaración de limitación de responsabilidad, ni implícita ni explícita, en determinadas transacciones, por lo que cabe la posibilidad de que esta declaración no se aplique en su caso.

Esta información puede contener imprecisiones técnicas o errores tipográficos. Periódicamente se efectúan cambios en la información incluida en este documento; estos cambios se incorporarán en nuevas ediciones de la publicación. IBM puede efectuar mejoras y/o cambios en los productos y/o programas descritos en esta publicación en cualquier momento y sin previo aviso.

Cualquier referencia hecha en esta información a sitios Web no de IBM se proporciona únicamente para su comodidad y no debe considerarse en modo alguno como promoción de esos sitios Web. Los materiales de estos sitios Web no forman parte de los materiales de IBM para este producto y el uso que se haga de estos sitios Web es de la entera responsabilidad del usuario.

IBM puede utilizar o distribuir la información que usted le suministre del modo que IBM considere conveniente sin incurrir por ello en ninguna obligación para con usted.

Los licenciatarios de este programa que deseen obtener información acerca del mismo con el fin de: (i) intercambiar la información entre programas creados independientemente y otros programas (incluyendo éste) y (ii) utilizar mutuamente la información que se ha intercambiado, deben ponerse en contacto con:

IBM Corporation
Software Interoperability Coordinator, Department 49XA
3605 Highway 52 N
Rochester, MN 55901
Estados Unidos

Esta información puede estar disponible, sujeta a los términos y condiciones adecuados, incluyendo en algunos casos el pago de una tarifa.

El programa bajo licencia descrito en esta información y todo el material bajo licencia disponible para el mismo, se proporciona bajo los términos del Acuerdo de Cliente IBM, el Acuerdo de Licencia de Programa Internacional IBM o cualquier otro acuerdo equivalente entre ambas partes.

Los datos de rendimiento aquí contenidos se han determinado en un entorno controlado. Por consiguiente, los resultados obtenidos en otros entornos operativos pueden variar significativamente. Pueden haberse realizado mediciones en sistemas en nivel de desarrollo, por lo que no hay garantía de que dichas mediciones sean iguales en los sistemas disponibles en general. Incluso algunas de esas mediciones pueden haberse estimado mediante extrapolación. Los resultados reales pueden variar. Los usuarios de este documento deberán verificar los datos aplicables a su entorno específico.

La información concerniente a productos no IBM se ha obtenido de los suministradores de esos productos, de sus anuncios publicados o de otras fuentes de información pública disponibles. IBM no ha comprobado los productos y no puede afirmar la exactitud en cuanto a rendimiento, compatibilidad u otras características relativas a productos no IBM. Las consultas acerca de las posibilidades de productos no IBM deben dirigirse a los suministradores de los mismos.

Todas las declaraciones respecto a las intenciones futuras de IBM están sujetas a cambios o a su retirada sin aviso, y solamente representan metas y objetivos.

Esta información contiene ejemplos de datos e informes utilizados en operaciones comerciales diarias. Para ilustrarlos tan completamente como sea posible, los ejemplos pueden incluir nombres de individuos, compañías, marcas y productos. Todos estos nombres son ficticios y cualquier parecido con los nombres y direcciones utilizados por una empresa real es pura coincidencia.

LICENCIA DE COPYRIGHT:

Esta información contiene programas de aplicación de ejemplo en lenguaje fuente, que muestran técnicas de programación en varias plataformas operativas. Puede copiar, modificar y distribuir estos programas de ejemplo de cualquier forma sin pagar nada a IBM, bajo el propósito de desarrollo, uso, márketing o distribución de programas de aplicación de acuerdo con la interfaz de programación de la aplicación para la plataforma operativa para la cual se han escrito los programas de ejemplo. Estos ejemplos no se han probado exhaustivamente bajo todas las condiciones. Por tanto, IBM no puede garantizar la fiabilidad, capacidad de servicio o funcionamiento de estos programas. Puede copiar, modificar y distribuir estos programas de ejemplo de cualquier forma sin pagar nada a IBM bajo el propósito de desarrollo, uso, márketing o distribución de programas de aplicación de acuerdo con los interfaces de programación de aplicaciones de IBM.

Cada copia o parte de estos programas de ejemplo o de los trabajos derivados de ellos, debe incluir un aviso de copyright de este tipo:

(C) (el nombre de la empresa) (año). Partes de este código derivan de IBM Corp. Programas de ejemplo.
(C) Copyright IBM Corp. _entre el año o años_. Reservados todos los derechos.

Si está viendo esta información en copia software, las fotografías y las ilustraciones a color podrían no aparecer.

Marcas registradas

Los términos siguientes son marcas registradas de International Business Machines Corporation en Estados Unidos y/o en otros países:
AS/400
e (logo)
IBM
iSeries
Operating System/400
OS/400

Microsoft, Windows, Windows NT, y el logotipo de Windows son marcas registradas de Microsoft Corporation en los Estados Unidos y/o en otros países.

Java y todas las marcas registradas basadas en Java son marcas registradas de Sun Microsystems, Inc. en los Estados Unidos y/o en otros países.

UNIX es una marca registrada de The Open Group en los Estados Unidos y en otros países.

Otros nombres de empresas, productos y nombres de servicio pueden ser marcas registradas o marcas de servicio de otros.

Términos y condiciones para bajar e imprimir publicaciones

Los permisos para el uso de las publicaciones que ha seleccionado para bajar se otorgan de acuerdo con los siguientes términos y condiciones y la indicación de que los ha aceptado.

Uso personal: Puede reproducir estas publicaciones para su uso personal, no comercial, siempre que se respeten todos los avisos de propiedad. No puede distribuir, exhibir ni realizar trabajos derivados de estas publicaciones, ni de cualquier parte de ellas, sin el consentimiento expreso de IBM.

Uso comercial: Puede reproducir, distribuir y exhibir estas publicaciones solamente dentro de su empresa, siempre que se respeten todos los avisos de propiedad. No puede realizar trabajos derivados de estas publicaciones, ni reproducir, distribuir o exhibir estas publicaciones ni parte de ellas, fuera de su empresa sin el consentimiento expreso de IBM.

No se otorgan otros permisos, licencias o derechos, excepto los otorgados expresamente en este permiso, ya sea de forma expresa o implícita, sobre las publicaciones o la información, datos, software u otra propiedad intelectual contenida en ellas.

IBM se reserva el derecho a retirar los permisos aquí otorgados siempre que, a su juicio, el uso de las publicaciones vaya en detrimento de su interés o, tal como lo determina IBM, no se hayan seguido las instrucciones anteriores correctamente.

No puede bajar, exportar o reexportar esta información a menos que se cumplan totalmente todas las leyes y reglamentos aplicables, incluidas todas las leyes y reglamentos de exportación de los Estados Unidos. IBM NO OFRECE NINGUNA GARANTÍA SOBRE EL CONTENIDO DE ESTAS PUBLICACIONES. LAS PUBLICACIONES SE SUMINISTRAN ″TAL CUAL″ Y SIN GARANTÍA DE NINGUNA CLASE, YA SEA EXPRESA O IMPLÍCITA, INCLUYENDO PERO NO LIMITÁNDOSE A LAS GARANTÍAS IMPLÍCITAS DE COMERCIABILIDAD Y APTITUD PARA UNA FINALIDAD CONCRETA.

Todo el material bajo copyright de IBM Corporation.

Al bajar o imprimir una publicación desde este sitio, ha indicado estar de acuerdo con estos términos y condiciones.

Información de limitación de responsabilidad sobre el código

Este documento contiene ejemplos de programación.

IBM otorga al usuario una licencia de copyright no exclusiva para utilizar todos los ejemplos de código de programación, a partir de los que puede generar funciones similares adaptadas a sus necesidades específicas.

IBM proporciona la totalidad del código de ejemplo sólo con propósito ilustrativo. Estos ejemplos no se han probado exhaustivamente bajo todas las condiciones. Por tanto, IBM no puede garantizar la fiabilidad, capacidad de servicio o funcionamiento de estos programas.

Todos los programas que contiene esta documentación se suministran ″TAL CUAL″, sin garantías de ninguna clase. Se renuncia explícitamente a las garantías implícitas de no infringibilidad, comercialización y adecuación a un propósito determinado.

Impreso en España