G2 JavaLink

User's Guide

Version 2015

G2 JavaLink User's Guide, Version 2015

December 2015

The information in this publication is subject to change without notice and does not represent a commitment by Gensym Corporation.

Although this software has been extensively tested, Gensym cannot guarantee error-free performance in all applications. Accordingly, use of the software is at the customer's sole risk.

Gensym Corporation
52 Second Avenue
Burlington, MA 01803 USA
Telephone: (781) 265-7100
Fax: (781) 265-7101

Part Number: DOC055-1200

Contents

Preface

Describes this guide and the conventions that it uses.

About this Guide

This guide describes the basic features of G2 JavaLink (JavaLink) and explains how you can use these features to create applications.

Software Requirements

For information about the software requirements and installation of this version of G2 JavaLink, refer to the *readme-javalink.html* file supplied with G2 JavaLink, which includes this and other important instructions.

Audience

This document assumes you are familiar with the features and syntax of the Java programming language, and you know how to develop Java applications. It also assumes you are familiar with the syntax of G2.

Conventions

This guide uses the following typographic conventions and conventions for defining system procedures.

Typographic

Convention Examples	Description
g2-window, g2-window-1, ws-top-level, sys-mod	User-defined and system-defined G2 class names, instance names, workspace names, and module names
history-keeping-spec, temperature	User-defined and system-defined G2 attribute names
true, 1.234, ok, “Burlington, MA”	G2 attribute values and values specified or viewed through dialogs
Main Menu > Start KB Workspace > New Object create subworkspace Start Procedure	G2 menu choices and button labels
conclude that the x of y ...	Text of G2 procedures, methods, functions, formulas, and expressions
new-argument	User-specified values in syntax descriptions
text-string	Return values of G2 procedures and methods in syntax descriptions
File Name, OK, Apply, Cancel, General, Edit Scroll Area	GUIDE and native dialog fields, button labels, tabs, and titles
File > Save Properties	GMS and native menu choices
workspace	Glossary terms

Convention Examples	Description
c:\Program Files\Gensym	Windows pathnames
/usr/gensym/g2/kbs	UNIX pathnames
spreadsh.kb	File names
g2 -kb top.kb	Operating system commands
public void main() *gsi_start*	Java, C and all other external code

Note Syntax conventions are fully described in the *G2 Reference Manual*.

Procedure Signatures

A procedure signature is a complete syntactic summary of a procedure or method. A procedure signature shows values supplied by the user in *italics*, and the value (if any) returned by the procedure *underlined*. Each value is followed by its type:

g2-clone-and-transfer-objects
(*list*: class item-list, *to-workspace*: class kb-workspace,
delta-x: integer, *delta-y*: integer)
-> *transferred-items*: g2-list

Related Documentation

G2 Core Technology

- *G2 Bundle Release Notes*
- *Getting Started with G2 Tutorials*
- *G2 Reference Manual*
- *G2 Language Reference Card*
- *G2 Developer's Guide*
- *G2 System Procedures Reference Manual*

- *G2 System Procedures Reference Card*
- *G2 Class Reference Manual*
- *Telewindows User's Guide*
- *G2 Gateway Bridge Developer's Guide*

G2 Utilities

- *G2 ProTools User's Guide*
- *G2 Foundation Resources User's Guide*
- *G2 Menu System User's Guide*
- *G2 XL Spreadsheet User's Guide*
- *G2 Dynamic Displays User's Guide*
- *G2 Developer's Interface User's Guide*
- *G2 OnLine Documentation Developer's Guide*
- *G2 OnLine Documentation User's Guide*
- *G2 GUIDE User's Guide*
- *G2 GUIDE/UIL Procedures Reference Manual*

G2 Developers' Utilities

- *Business Process Management System Users' Guide*
- *Business Rules Management System User's Guide*
- *G2 Reporting Engine User's Guide*
- *G2 Web User's Guide*
- *G2 Event and Data Processing User's Guide*
- *G2 Run-Time Library User's Guide*
- *G2 Event Manager User's Guide*
- *G2 Dialog Utility User's Guide*
- *G2 Data Source Manager User's Guide*
- *G2 Data Point Manager User's Guide*
- *G2 Engineering Unit Conversion User's Guide*
- *G2 Error Handling Foundation User's Guide*
- *G2 Relation Browser User's Guide*

Bridges and External Systems

- *G2 ActiveXLink User's Guide*
- *G2 CORBALink User's Guide*
- *G2 Database Bridge User's Guide*
- *G2-ODBC Bridge Release Notes*
- *G2-Oracle Bridge Release Notes*
- *G2-Sybase Bridge Release Notes*
- *G2 JMail Bridge User's Guide*
- *G2 Java Socket Manager User's Guide*
- *G2 JMSLink User's Guide*
- *G2 OPCLink User's Guide*
- *G2 PI Bridge User's Guide*
- *G2-SNMP Bridge User's Guide*
- *G2 CORBALink User's Guide*
- *G2 WebLink User's Guide*

G2 JavaLink

- *G2 JavaLink User's Guide*
- *G2 DownloadInterfaces User's Guide*
- *G2 Bean Builder User's Guide*

G2 Diagnostic Assistant

- *GDA User's Guide*
- *GDA Reference Manual*
- *GDA API Reference*

Customer Support Services

You can obtain help with this or any Gensym product from Gensym Customer Support. Help is available online, by telephone, by fax, and by email.

To obtain customer support online:

➔ Access G2 HelpLink at *www.gensym-support.com*.

You will be asked to log in to an existing account or create a new account if necessary. G2 HelpLink allows you to:

- Register your question with Customer Support by creating an Issue.
- Query, link to, and review existing issues.
- Share issues with other users in your group.
- Query for Bugs, Suggestions, and Resolutions.

To obtain customer support by telephone, fax, or email:

➔ Use the following numbers and addresses:

	Americas	Europe, Middle-East, Africa (EMEA)
Phone	(781) 265-7301	+31-71-5682622
Fax	(781) 265-7255	+31-71-5682621
Email	*service@gensym.com*	*service-ema@gensym.com*

To obtain G2 JavaLink specific customer support by e-mail:

➔ Use the following address:

javalink-service@gensym.com

1

Introduction

Provides an overview of G2 JavaLink features and components.

Introduction

G2 JavaLink (JavaLink) allows you to write G2 bridges entirely in Java. By using Java, you as a developer will immediately feel the benefit from a modern multi-threaded, object-oriented language. Additionally, extra features exist in JavaLink that are not available in the standard G2 Gateway (GSI) product.

G2 JavaLink defines the mapping from Java types to G2 types and from G2 types to Java types. It hides the finer details of G2 communications, thus making it easier and more productive to write bridges.

For example, the following Java class connects to a G2 server and sends a message to its message board:

```
import com.gensym.jgi.*;

/**
* Simple JavaLink example to send a message to G2.
**/

public class JavaLinkExample {
   public static void main(String args[]) {
      G2Connection g2_connection = null;
      try {
         g2_connection =
            G2Gateway.getOrMakeConnection("localhost", "1111");
         } catch (G2AccessInitiationException e) {
            System.out.println("Problem connecting to G2
               exception was:" + e.toString());
         }
         g2_connection.returnMessage("Hi from JavaLink");
         System.exit(0);
   }
}
```

Developers can use G2 JavaLink to create small, stand-alone, windowless bridges or sophisticated clients that can use the increasingly rich set of standard Java APIs, such as RMI, Java Beans, and Java AWT.

Note G2 JavaLink is a connectivity solution; it is not an applet or application development tool. To create G2 client applets and applications, you can use Telewindows2 Toolkit.

Summary of Features

This section summarizes the major features of G2 JavaLink.

Mapping Between G2 and Java Types and Classes

G2 JavaLink defines a mapping between G2 and Java data types and classes. The data conversion between the two environments is automatic and requires no developer intervention. JavaLink exports G2 classes, providing Java classes that represent each G2 class that Java accesses externally.

You can pass objects from G2 by copy or by handle. Objects passed by handle do not retain a local copy of the object's attribute data but instead automatically refer to G2 for attribute data, as required. By default, JavaLink sends G2 objects to Java "by handle" when calling one of its built-in RPCs and for G2 methods called from

Java and their return values. For Java methods called from G2 and their return values, JavaLink sends G2 objects to Java according to the RPC declaration in G2.

Object-Oriented API

G2 JavaLink exists as a set of standard Java packages, where communication to G2 is handled by using a connection class.

Multi-Threaded Bridges

G2 JavaLink works within Java's multi-threaded environment. Therefore, multiple Java threads can safely call G2 procedures and methods simultaneously. Additionally, G2 can call Java methods within existing threads or from newly created threads.

Synchronous RPC Calling

Using G2 Gateway (GSI), it has traditionally been difficult to create a synchronous RPC call into G2 from a C-based bridge, that is, where the bridge waits for the result of an RPC call. G2 JavaLink provides synchronous RPC calls as the default behavior. While a Java thread is waiting for a synchronous RPC, other Java threads can communicate with G2 without interruption.

Unicode Support

Both Java and G2 use Unicode character sets. G2 JavaLink receives and sends all strings as Unicode between G2 and Java.

Comprehensive Error Handling

G2 JavaLink provides extensive error handling:

- When G2 calls a Java method and a Java exception is raised, JavaLink reports the Java exception back to the G2 procedure or method that initiated the call. You can trap such errors by using the standard G2 on error syntax, or you can leave them to appear in the G2 Operator Logbook.
- When Java calls a G2 procedure or method and a G2 error occurs, JavaLink raises a Java exception containing the details of the G2 error that occurred and reports back to the Java thread that originally called the G2 RPC.
- JavaLink reports all other communication errors as error events to the Java client.

Multiple Connections and Bridge-Initiated Connections

G2 JavaLink allows many G2 servers to connect to a single Java client, where JavaLink creates a separate connection object for each connection. You can initiate connections between a G2 and a Java client from G2 or from Java.

Java Remote Method Invocation (RMI) Support

G2 JavaLink works with the standard Java Remote Method Invocation (RMI) API that was introduced with Sun's JDK 1.1. JavaLink provides classes that allow you to access G2 via RMI.

G2 Support for the JavaBeans Event Model

By using the optional KB modules supplied with G2 JavaLink, *javalink.kb* and *g2evliss.kb*, you can define G2 classes so that external Java classes can listen for events generated inside G2. This mechanism implements the event listener specification defined for the JavaBeans event model.

Ease of Migration

Existing G2 bridge developers who have learned Java will find few problems when migrating to G2 JavaLink.

Components of G2 JavaLink

G2 JavaLink consists of two components:

- **Java packages**. These packages and their contents provide full javadoc documentation. The Java packages are contained in the *.jar* files located in the *classes* subdirectory of the *javalink* directory.
- **Supporting G2 KB modules**. These modules provide optional KB support and are not required for JavaLink to function. The *.kb* files are located in the *kbs* subdirectory of the *javalink* directory.

G2 JavaLink Packages

Package Name	Summary
com.gensym.jgi	Provides Java support for communication between G2 and Java.
com.gensym.jgi.download	Provides support for creating Java interfaces and stubs that represent G2 user classes.
com.gensym.jgi.dataservice	Provides JavaLink support to enable GSI variable data service in Java.
com.gensym.util	Provides various common utility classes required for the correct operation of JavaLink and other Gensym Java products, such as Telewindows2 Toolkit.
com.gensym.classes	Contains Java classes to represent every built-in system class in G2.
*com.gensym.classes.modules.**	Contain classes that represent G2 classes found in KBs that Gensym supplies and user-defined classes whose interfaces you download.
com.gensym.jgi.rmi	Provides the support classes that allow communication with G2, using the Java Remote Method Invocation API (RMI).

The com.gensym.jgi Package

The *com.gensym.jgi* package provides Java classes that support communication between G2 and Java. The most important of these classes is *G2Gateway*, which provides a network interface to a gsi-interface object created within a connected G2.

The three distinct ways of communicating with G2 through a *G2Gateway* are:

- **Simple messages**: You can send messages to G2's message board by calling *returnMessage*, and you can receive messages from G2 by implementing the *g2MessageReceived* method in the *G2ConnectionListener* interface.
- **Remote procedure calls**: You can register Java methods as callable from G2 with *registerJavaMethod*. You can call and start G2 procedures and methods synchronously by calling *callRPC* and *startRPC*, respectively.
- **Data service**: You can enable Java data service to supply data on demand to GSI variables defined in G2.

JavaLink converts data types in RPC calls between G2 and Java automatically. For a description of type and class mapping between Java and G2, see How G2 Classes Map to Java Classes.

A *G2ConnectionListener* can register interest in connection events by calling *addG2ConnectionListener* and can remove interest by calling *removeG2ConnectionListener*.

G2Gateway allows multiple Java threads to access a G2 connection.

The *G2Gateway* class implements the interface *G2Connection*, which is a contract that ensures that *G2Gateway* provides all the required interfaces for communication with G2. Normally, you declare JavaLink methods, such as *G2Gateway.getOrMakeConnection* to return a *G2Connection*, not a *G2Gateway*. This allows Java applications to use the contract defined by *G2Connection* and not worry about an implementation such as *G2Gateway*.

You can enable debugging by calling *setInterfaceDebugOptions* on *G2Gateway* at any point in the program execution. However, enabling debugging can significantly slow execution if it is on all the time. Therefore, we recommend that you enable it only when you need to debug a problem.

Establishing a Connection with G2

A *G2Gateway* can communicate with a G2 once a connection has been established. *G2Gateway* can create the connection by using one of these two methods:

- *initiateConnection*
- *getOrMakeConnection*

G2Gateway can also wait for a G2 server to make the connection. When establishing a connection to G2, you can make the connection permanent, even across KB resets.

Any G2 can attempt to make an unsolicited connection to Java, but the connection must be accepted by an installed *G2ConnectionHandlerFactory*. The static method *G2Gateway.setG2ConnectionHandlerFactory* allows a *G2ConnectionHandlerFactory* instance to select or create a *G2Connection* for any incoming unsolicited network connections from G2.

A *G2Gateway* can only be connected to one G2 at any one time. If you require multiple connections within a Java application, you must create multiple instances of *G2Gateway*. Alternatively, you may use a *ConnectionManager*, which is available by using Telewindows2 Toolkit. For details, see the *Telewindows2 Toolkit Java Developer's Guide: Application Classes*.

G2 JavaLink supports secure communication on Windows and Linux platforms, using SSPI and OpenSSL, respectively. Support for OpenSSL on other UNIX platforms will be available in a future release. For details, see the description of *G2ConnectionInfo* in the Java doc.

To establish a secure listener within G2 JavaLink, use the *-secure* and *-cert* G2 Gateway command-line arguments in *G2Gateway.initialize*.

Summary of Classes Found in Package com.gensym.jgi

The following table lists and describes commonly used classes in the *com.gensym.jgi* package:

Class	Description
G2Gateway	An implementation of a network interface to a gsi-interface object created within a connected G2. It implements the *G2Connection* interface.
G2ConnectionInfo	An object that contains all the information necessary to get a handle on a specific instance of a *G2Connection*.
G2ConnectionEvent	An event object that encapsulates information fired during a G2 connection event.
G2CommunicationErrorEvent	An event raised when an asynchronous G2 communication error has occurred.
G2ConnectionAdapter	An empty implementation of *G2ConnectionListener*, provided as a convenience for creating a connection listener by extending the class and overriding only the methods of interest.

Class	Description
G2Connector	A visual component provided as a Java Bean that provides the event and method interfaces of a *G2Connection*. You use this bean in builder environments where the static *getOrMakeConnection* method on *G2Gateway* cannot be called.
G2ConnectorBeanInfo	A Java Bean Info for *G2Connector*.

The following table lists and describes commonly used interfaces in the *com.gensym.jgi* package:

Interface Name	Description
G2Connection	Defines access to and from a G2 network connection. A *G2Gateway* implements a *G2Connection*. *G2Connection* itself is defined by implementing the outbound interface, *G2Access*, and the inbound *G2Callbacks*.
G2Access	Provides outbound methods for accessing a connected G2.
G2ConnectionHandlerFactory	Defines a handler for incoming unsolicited calls initiated by G2.
G2ConnectionListener	Informed when any G2 connection event occurs, such as when the connected G2 is being paused.

The following table lists and describes the exceptions and errors in the *com.gensym.jgi* package:

Exceptions/Errors	Description
ConnectionTimedOutException	A time-out has occurred when attempting to connect to G2.
TimeoutException	A time-out has occurred during an interface operation, such as a remote procedure call.
G2AccessException	A problem has occurred while accessing G2.
ConnectionNotAliveException	A problem has occurred while attempting to access G2 via a *G2Connection* that is not alive.
ConnectionNotAvailableException	An attempt has been made to access a G2 connection that is not available or has not been created.
G2CommunicationException	A G2 communication problem has occurred.
G2ItemDeletedException	A deleted item has been passed to G2 as an argument of an RPC call.
G2SecurityException	The requested access to G2 is not valid from this class of access.

The com.gensym.jgi.download Package

The *com.gensym.jgi.download* package supports the creation and loading of Java interfaces and classes that represent G2 classes.

The following table lists and describes public classes and exceptions in this package:

Class	Description
DownloadInterfaces	A utility class that allows JavaLink developers to pre-export user-defined classes from G2 to Java.
G2StubCreationException	An exception that is raised when G2 stub creation has failed.
StubCreationException	An exception that is raised at run time when stub creation has failed.

The com.gensym.jgi.dataservice Package

The *com.gensym.jgi.dataservice* package supports the ability to implement Java data service that services data requests from GSI variables created within G2.

For more information about configuring GSI variables in G2, please refer to "Configuring GSI Variables in the Knowledge Base" in Chapter 2 of the *G2 Gateway Bridge Developer's Guide.*

The following table lists and describes public classes and interfaces in this package:

Class	Description
DataService	Provides a JavaLink connection with G2 variable data service capabilities.
DataServiceListener	Informed of *DataService* events from a G2 connection.
DataServiceEvent	Contains data detailing a *DataServiceListener* event.

The com.gensym.util Package

The *com.gensym.util* package contains various common utility classes required for the correct operation of JavaLink and other Gensym Java products.

The following table lists and describes classes, interfaces, and exceptions in this package important to G2 JavaLink:

Class	Description
Sequence	Maps to the G2 sequence data type.
Structure	Maps to the G2 structure data type.
Symbol	Maps to the G2 symbol data type.
ItemListener	Informed of events regarding changes to G2 items.
ItemEvent	Encapsulates information that represents changes to a G2 item.
VariableValueListener	Receives notification when the value of a G2 variable or parameter changes. **Note**: An *ItemListener* does not receive such events.
VariableValueEvent	Encapsulates information that represents changes to the value of a G2 variable or parameter.
JavaClassCompilerException	Raised when JavaLink fails to compile a Java source file, for example, when creating a Java class to represent a G2 user-defined class.
NoSuchAttributeException	Signals an attempt to access a non-existent attribute in a G2 structure.

The G2 Javalink Module

The G2 side of G2 JavaLink consists of an optional G2 module named javalink, which exists when you load *javalink.kb*. The javalink module provides:

- JavaLink class definitions.
- Procedures for creating Java types.
- Support for exporting Java Bean events from correctly configured G2 classes.

The javalink module is *not* required for JavaLink to function; it provides G2 classes that enable G2 to map to basic Java types that are not directly available in G2.

2

Mapping between G2 and Java

Describes how JavaLink maps data types and classes between G2 and Java.

Introduction

G2 JavaLink (JavaLink) defines a mapping between G2 and Java data types and classes. The data conversion between the two environments is automatic and requires no developer intervention. JavaLink exports G2 classes at run time by providing Java classes that represent each G2 class that Java accesses externally. It does this by performing a "thin download," which uses existing JavaLink classes or pseudo classes to represent the G2 class. If you need to access the attributes and/or methods of user-defined G2 classes, you can also explicitly download G2 classes to make them available at compile time.

How JavaLink Maps Data Types

This section describes how JavaLink:

- Automatically maps data types in RPC calls and exported methods.
- Supports G2 symbols in Java.
- Supports Java built-in types.

Automatic Conversion of Data Types

The following table lists Java data types and the G2 items or values into which they are converted when they are passed to G2 as arguments of remote procedure calls (RPCs). JavaLink performs these conversions even when the Java data types are embedded as attributes of other objects.

Type Conversion in RPC Calls by G2 JavaLink to G2

An argument of this Java data type...	Is converted to this G2 data type when passed to G2...
java.lang.Integer	integer
java.lang.String	text
java.lang.Boolean	truth-value
java.lang.Double	float
java.lang.Character	text (of one character)
java.lang.Float	float
java.lang.Byte	integer
java.lang.Short	integer
java.lang.Long	java.lang.type.long **Note**: This class is defined in the optional G2 KB *javalink.kb*.
com.gensym.util.Symbol	symbol
com.gensym.util.Sequence	sequence
com.gensym.util.Structure	structure
java.lang.Object[]	g2-array
boolean[]	truth-value-array
com.gensym.util.Symbol[]	symbol-array
java.lang.String[]	text-array
java.lang.Number[]	quantity-array
double[]	float-array
int[]	integer-array

Additionally, the following table lists the Java data types and the G2 values into which they are converted when they are passed to G2 as arguments of methods found in exported Java classes that represent G2 classes:

Additional Type Conversion for Exported Methods to G2

An argument of this Java data type...	Is converted to this G2 data type when passed to G2...
int	integer
boolean	truth-value
double	float

The following table lists G2 data types and the Java data types into which they are converted when passed as arguments in remote procedure calls (RPCs) from G2.

Type Conversion in RPC Calls by G2 to Java Methods

An argument of this G2 data type...	Is converted to this Java data type when passed to a Java method...
text	*String*
truth-value	*boolean* or *java.lang.Boolean*
float	*double* or *java.lang.Double*
symbol	*com.gensym.util.Symbol*
sequence	*com.gensym.util.Sequence*
structure	*com.gensym.util.Structure*
integer	*int* or *java.lang.Integer*

Note When data types are embedded within other objects, such as attributes, array elements, or sequence elements, JavaLink converts them to their appropriate object-wrapped class, for example, *double* becomes *java.lang.Double*.

Supporting G2 Symbols in Java

JavaLink represents G2 symbol data types by using the *com.gensym.util.Symbol* class, which has no public constructors. Thus, you must create all Java symbols by using this method:

```
public static Symbol intern(String symbolText)
```

The *intern* method takes as its argument the text representing the G2 symbol value and returns a *Symbol* object. Symbol objects have the following property:

```
Symbol.intern(String s1).equals(Symbol.intern(String s2)
if s1.equals(s2)
```

Similar to G2, JavaLink caches symbols; therefore, symbol objects for equivalent strings normally return the same object. However, you should ***always*** use the *equals* method to check if two symbols are == because symbol objects created during deserialisation, for example, from a Java RMI call, might not be equivalent. Also note that every symbol object that JavaLink creates uses heap space, which cannot be freed.

You can create mixed-case symbols in G2 by appending the @ character to each character, for example, *@a@b@cabc == abcABC*. However, because G2 by default assumes symbol text is uppercase, you must provide symbol text in uppercase in Java before creating a symbol object, by calling *intern*.

By convention, the variable used to hold the symbol in Java should be upper case and end with an underscore.

For example, to pass the symbol a-handy-symbol to G2, you would create the symbol in Java as follows:

```
Symbol A_HANDY_SYMBOL_ = Symbol.intern("A-HANDY-SYMBOL");
```

However, if you pass the symbol as lowercase text, as follows, G2 would receive the symbol @a-@h@a@n@d@y-@s@y@m@b@o@l:

```
Symbol A_HANDY_SYMBOL_ = Symbol.intern("a-handy-symbol");
```

Supporting Java Built-in Types

Some Java types have no corresponding G2 type. To pass values of these Java types from G2 to Java, you can use classes and procedures provided in the optional javalink KB module. You can then pass these objects to G2 JavaLink through remote procedure calls (RPCs).

The following table describes the G2 classes in the javalink module that convert directly to Java data types:

G2 Classes in the JAVALINK Module that Convert Directly to Java Types

An argument of this G2 data type...	Converts to this Java data type when passed to a Java method...
java.lang.long.type	*long* or *java.lang.Long*
java.lang.byte.type	*byte* or *java.lang.Byte*
java.lang.short.type	*short* or *java.lang.Short*
java.lang.integer.type	*int* or *java.lang.Integer* **Note**: Use this class when you need to send the full 32 bits of a Java *int* to Java. Normally, you would use the G2 integer type, which is 30 bits.
java.lang.character.type	*character* or *java.lang.Character*
java.lang.float.type	*float* or *java.lang.Float*

Note When data types are embedded within other objects, such as attributes, array elements, or sequence elements, JavaLink converts them to their appropriate object wrapped class, for example, *int* becomes *java.lang.Integer*.

Java Type Creation Procedures

All the above Java type classes subclass jgi-java-type. To create instances of these classes, you must use the new method, which ensures that the relevant attributes get initialized correctly. The new method signature is:

```
new(g2-type-class: class jgi-java-type-definition,
      reserved: item-or-value,
      val: item-or-value) = (class jgi-java-type)
```

For example, to create a java.lang.long.type :

```
long = call new(java.lang.long.type,
                the symbol none,
                "1234567890");
```

How G2 Classes Map to Java Classes

When a JavaLink client refers to the attributes or methods of a G2 class, JavaLink performs what is called a "thin download," which uses existing Java classes or pseudo classes to represent the G2 item. By using a "thin download," JavaLink avoids the proliferation of unwanted Java classes on the client, particularly when the G2 class uses multiple inheritance.

Because no corresponding Java classes exist for user-defined classes, JavaLink allows you to create a Java interface class and a Java implementation of that interface, called a **stub**, which represent a G2 class. To do this, you use the G2 DownloadInterfaces feature of G2 JavaLink, which creates property accessor methods for each G2 class-specific attribute and class methods for each declared method. Once these classes exist, you can import them into your Java code to get and set user-defined attributes, and call user-defined methods of G2 items.

Naming Conventions

Java identifiers follow a convention that differs from G2:

- Java identifiers are case sensitive; G2 identifiers are not.
- Java identifiers do not accept certain characters, such as hyphens (-); whereas, you often use hyphens to separate words within G2 identifiers, for example, this-is-a-g2-class-name.

JavaLink, therefore, converts G2 identifiers to Java identifiers, using the following rules:

1. First, all characters are made lowercase.
2. Any "-" or "/" character is removed and each character before the "-" or "/" is capitalized.
3. Any !@#%^&*()[].,<>+|=`~\"' characters are replaced by a "_" character.
4. Any "$" character causes the next character to be capitalized.
5. If the Java identifier is a class name, the first character is capitalized.

For example:

G2 Identifier	Java Identifier
this-is-a-g2-class-name	*ThisIsAG2ClassName*
a-g2-method-name	*aG2MethodName*
a-funny&*attribute-name	*aFunny__AttributeName*
a-g2-procedure-$x	*aG2ProcedureX*

Attributes Accessors

The downloaded JavaLink class provides two categories of methods for accessing attributes of the G2 class:

- **Getters**, which allow you to obtain values for any attribute defined by the class.
- **Setters**, which allow you to modify the value of any attribute of the class.

These methods follow the Java Bean specification for "Property Accessors."

For example, suppose you have a G2 class named tank, with an attribute level defined as:

```
Level is a float, initially is 0.0;
```

The Java interface exported for the tank class would have:

- A getter called *getLevel*, which returns the current value of the level attribute of a tank. The method would have the following Java signature:

```
public float getLevel();
```

- A setter called *setLevel*, which sets the current value of the level attribute of a tank. The method would have the following Java signature:

```
public void setLevel(double level);
```

Class Methods

JavaLink exports the signatures for any methods that are defined for a G2 class, which you can call from Java. For example, suppose the G2 tank class defines a method with this signature:

```
fill-tank(this: class tank) = ()
```

The exported Java interface for tank would provide a method with the following signature:

```
public void fillTank();
```

Exported method signatures automatically take into account any data type or class mapping defined by JavaLink. Additionally, the exported stubs attempt to convert to and from object-wrapped types, for convenience, for example, *double* becomes *Double*. For example, suppose the tank class defined another method with this signature:

```
set-pump-status(this: class tank,
                the-pump: class pump,
                status: symbol,
                reason: text,
                time: float) = (truth-value)
```

The exported Java method signature would be:

```
public boolean setPumpStatus(Pump the-pump,
                             Symbol status,
                             String reason,
                             double time);
```

During export of the tank class, JavaLink automatically exports any superior classes to tank, as well as any classes that are used when defining attributes or method arguments. In the above example, JavaLink would automatically export the interface for the pump class, because it is used by one of the tank class's methods.

Class Mapping and Multiple Inheritance

The Java class model does not support multiple inheritance, whereas G2 does. JavaLink circumvents this by mapping G2 classes as Java interfaces, which do support multiple inheritance. Exported Java classes are therefore free to inherit from multiple G2 parents, thus mapping G2's multiple inheritance to the Java object environment.

When a G2 class is exported to Java, JavaLink always generates a Java interface that defines the signatures for all the attributes and methods that belong to that class. Any inherited attributes and methods are defined by extending the interface for each direct superior class. JavaLink automatically exports any superior class interfaces, as required. When JavaLink returns a G2 object from, say a G2 procedure call, it is guaranteed to implement the corresponding Java interface for the G2 class returned.

Local or Remote Access

You can use exported G2 classes locally or as a proxy to access a remote G2 object.

When passing a G2 object "by handle" from G2 to JavaLink, JavaLink automatically routes any calls to methods or property accessors to the appropriate method or attribute on the original G2 object.

Java applications can also instantiate exported G2 classes and call property accessors directly. In this case, JavaLink makes any property (attribute) changes locally in Java, which does not affect the corresponding G2 object, if there is one. If you then send the same Java object to G2 via JavaLink, G2 creates a G2 object corresponding to the Java class, updating any attributes that differ from the default. In this case, the Java object is said to have been passed "by copy" to G2.

When calling one of the built-in RPCs that pass objects from G2 to Java, such as `getUniqueNamedItem`, JavaLink passes objects "by handle," by default. It also passes values and return values of an RPC call from Java to a G2 method "by handle," by default. However, when calling a Java method from G2, JavaLink passes the target object and return values "by copy," by default, unless

specifically indicated in the RPC declaration in G2 that it should be sent "by handle."

For information on how to pass an object "by copy," see Creating a Local G2 Object in the Java Client and Accessing G2 Objects by Handle or by Copy.

How JavaLink Resolves a G2 Class to a Java Class

When G2 sends any reference of a G2 class instance to JavaLink, JavaLink must resolve the reference into an applicable Java object that represents the G2 item within Java. Normally, JavaLink attempts to find the best applicable class that was previously exported to Java from G2, using JavaLink. Because JavaLink has already exported all built-in G2 classes to the Java package `com.gensym.classes`, JavaLink normally resolves a G2 class to a Java class by representing it as a built-in G2 class. These classes obviously do not contain any user-defined method or attribute signatures.

To force JavaLink to resolve a user-defined G2 class to an appropriate Java class that contains the corresponding user-defined method signatures and property accessors, you must first export the user-defined G2 class to Java, using the G2 DownloadInterfaces facility. For information on how to do this, see How to Export a G2 Class as a Java Class.

JavaLink uses the following algorithm to determine how to resolve G2 classes to Java classes:

1. JavaLink refers to a client-specific, cached lookup table, which contains previously resolved G2 class names to Java class definitions. If a Java class is already associated with the G2 class name, JavaLink uses this class to create a Java object that represents the G2 instance in Java. If the cache lookup fails, JavaLink proceeds to step 2.
2. JavaLink reads the class hierarchy for the referenced G2 class from G2.
3. JavaLink then moves up recursively through this hierarchy, ensuring that the Java interfaces that represent each G2 class in the hierarchy are also loaded. JavaLink uses this algorithm recursively during this step to ensure that any superior G2 classes that do not have directly corresponding Java interfaces are also resolved to their nearest superior interface.
4. JavaLink constructs a fully qualified Java interface name by using the G2 class and module name. For information on how JavaLink determines the fully qualified name, see Location and Package Name of an Exported G2 Class. JavaLink then attempts to load this interface into the Java Virtual Machine (VM).
5. If JavaLink cannot find this interface, it attempts to select the interface corresponding to its direct superior G2 class, which is guaranteed to have been previously loaded in step 3. If the G2 class directly inherits from multiple G2 classes, JavaLink automatically constructs a Java "pseudo class," which is

an interface that inherits from the interface created for the direct superior class in G2, thus preserving multiple inheritance. JavaLink has now associated a Java interface with a G2 class and module name.

6 JavaLink now attempts to load an implementation of the selected Java interface. If JavaLink cannot load the implementation class from the current *CLASSPATH*, it automatically constructs an implementation for the selected interface. JavaLink caches the implementation and uses it to create an instance for any future references to the G2 class in the Java VM, as in step 1.

How to Export a G2 Class as a Java Class

If you write a Java class that references an interface corresponding to a particular user-defined G2 class, the Java interface must be available before you can compile the Java source file.

JavaLink supplies a Java utility called G2 DownloadInterfaces that connects to a specified G2, and downloads the appropriate Java interfaces and implementations (stubs) corresponding to specified G2 classes and their respective hierarchies.

G2 DownloadInterfaces provides an easy-to-use, graphical user interface (GUI) to this utility. To start G2 DownloadInterfaces with the GUI, enter this command:

```
java com.gensym.beanbuilder.G2DownloadInterfaces
```

On Windows platforms, you can run it from the Start menu by choosing Download Interfaces from the G2 JavaLink submenu of you Gensym G2 program group.

For more information, see the *G2 DownloadInterfaces User's Guide*.

DownloadInterfaces Synopsis

```
java com.gensym.jgi.download.DownloadInterfaces
    [-host hostname]
    [-port portnumber]
    -classes g2classname[,g2classname...]
    [-force]
    [-stubs]
    [+g]
    [+t]
    [+v]
```

Options

Option Name	Description
-host	(Optional) Specifies the host machine name or the IP address of the G2 from which to download. Default is *localhost*.
-port	(Optional) Specifies the port number of the G2 host machine from which to download. Default is *1111*.
-class or *-classes*	Specifies the G2 class name or a space-separated list of G2 class names to be downloaded.
-force	(Optional) Download even if a G2 user-defined class has been previously downloaded.
-stubs	(Optional) Generate the implementations for every Java interface created, even if the G2 user-defined class has been previously downloaded.
+g	Switches on the graphical user interface. If you use this switch when launching the utility with the *-class* or *-classes* option, the wizard is invoked. Otherwise, the wizard is invoked, by default.
+t	Sets the trace output for the G2 DownloadInterfaces utility to true. By default, tracing is not enabled.
+v	Runs the G2 DownloadInterfaces verification test. By default, the verification test is not run when you launch the utility.

Example

This example connects to a G2 on the machine named *localhost* at TCP/IP port *1111* and downloads the G2 class named a-g2-class. Because the *-force* and *-stubs* options are specified, JavaLink downloads interfaces and generates implementations, even if the classes have been previously downloaded.

```
java com.gensym.jgi.download.DownloadInterfaces
-host localhost -port 1111 -class A-G2-CLASS -force -stubs
```

Notes

In general, you should not change the class inheritance path in the G2 server of a class that inherits from G2 classes whose interfaces you have downloaded. In particular, if your Java code gets a handle to instances of such classes, those instances will be represented by the wrong Java class of stub.

If you do change the G2 inheritance path of a class that you have previously downloaded, you should download the class again, using the *-force* and *-stubs* options.

G2 DownloadInterfaces generates an error if the G2 class uses an attribute or method name that is already implemented in G2 JavaLink and is, therefore, reserved. Your classes must use unique names.

How JavaLink Creates an Exported Java Class

When JavaLink exports a G2 class, it performs the following steps:

1. Downloads class information for the specified class from G2, starting at the most superior class.
2. Recursively executes step 1 for any G2 class referenced in attribute and method parameter definitions.
3. Creates and compiles a Java interface that contains signatures for all attributes and methods. This interface extends any interfaces created for superior classes, thus maintaining the G2 class's inheritance structure. If the *-stubs* option is specified, it also creates and compiles the Java stub class that implements this interface.
4. Repeats step 3 for each superior class in order, then executes step 3 for the actual class.
5. Creates a stub class that implements the interface created in steps 3 and 4.

If you change the definition of a class in the G2 server, JavaLink correctly handles updates to existing instances of that class in the client. The exception is the case when you edit the inheritance path of a G2 class that inherits from one or more classes whose interfaces you have explicitly downloaded. In this case, JavaLink does not update existing instances of that class in the client.

When JavaLink downloads class information, it saves the information in a *.ser* file, whose name corresponds to the Java class name. The *.ser* file is only required when using G2 DownloadInterfaces and during runtime when JavaLink cannot use an existing downloaded class due to G2 multi-inheritance. When required, they need to be present in the downloaded package, even if the package is placed in a JAR file. The *.ser* file is not needed once G2 user-defined classes have been downloaded, and if during runtime JavaLink does not need to generate a Java stub.

Location and Package Name of an Exported G2 Class

JavaLink has downloaded all built-in G2 system classes for you. These classes are located in this JavaLink package:

com.gensym.classes

However, you must tell JavaLink of the physical location and package name for any user-defined G2 classes that you have downloaded. By default, user-defined classes exported for Gensym KBs are exported to:

com.gensym.classes.modules.<module-name-of-class>

JavaLink uses the following system properties when exporting G2 classes:

Property Name	Description
com.gensym.class.user.repository	The physical location of the root directory to which JavaLink should export user-defined classes. This directory should also be listed in your system's *CLASSPATH*. The default location is: Windows: *\javalink\classes* UNIX: */javalink/classes*
com.gensym.class.g2.package	The default package root for G2 classes. The default package is: *com.gensym.classes*

Property Name	Description
com.gensym.class.user.package	The default package root for user-defined classes. The default package is: *com.gensym.classes.modules*
com.gensym.class.user.pkgs	A list of package names of previously downloaded user-defined class packages that JavaLink searches through when locating a class not found by the system. For example: *com.gensym.classes.modules* *com.acme.kbapp.modules*

JavaLink loads these properties from the JavaLink properties file named:

.com.gensym.properties

This file must exist in a directory defined by the *HOME* environment variable. When JavaLink is being used in processes where the *HOME* environment variable cannot be retrieved, the file must also exist in the directory defined by */javalink/classes* in your G2 installation directory. The properties file in the *HOME* directory takes precedence. The properties file in the */classes* directory is used as a fallback when the other properties file cannot be found.

JavaLink places the *.com.gensym.properties* file in the correct directory for your system during the JavaLink installation process.

You can use the following command-line argument to load the properties file when running an application:

```
java -Dcom.gensym.properties.url
   file://mydir/.com.gensym.properties
```

In general, the *.com.gensym.properties* is not required at runtime. It is only required at runtime if you have downloaded user-defined classes, using G2 DownloadInterfaces or G2 Bean Builder.

For more information on this properties file, see the *readme-javalink.html* file.

When JavaLink Exports a G2 Class

JavaLink exports a G2 class when:

- A Java interface cannot be found for a corresponding multiply inherited G2 class that is being passed between G2 and Java.
- A Java implementation cannot be found for a corresponding G2 class that is being passed between G2 and Java, whose previously exported Java interface was found.

- JavaLink has been forced to download a G2 class with the *-force* options on G2 DownloadInterfaces. In this case, the existing version of the downloaded interface, and implementation if the *-stubs* option is also set, is overwritten.

JavaLink uses the following steps to ascertain the package name and location when creating any Java class or interface to represent a G2 class definition:

1. If the G2 class definition belongs to a module that defines the module-annotation attribute in the Module Information system table by specifying the attribute java-package-for-export, JavaLink uses the specified Java package name for the exported Java files. For example:

   java-package-for-export is "com.gensym.classes.modules.uilroot"

2. If no java-package-for-export is specified, JavaLink reads the class definition's module name, converts it to a Java identifier, using all lowercase, and appends this module name to the Java package name specified in the *com.gensym.class.user.package* property. If the class does not belong to any module, JavaLink uses the module named unspecified.

3. JavaLink reads the directory name from the property *com.gensym.classes.user.repository* and uses it as the root directory to which JavaLink should export user-defined classes. If the package's directory does not exist, JavaLink automatically creates it.

Examples

Suppose you have a G2 class definition named computer-supplier that has no module and resides in a G2 running on host "*london*" on port *1111*. And suppose you are running on a Windows platform, using the default *com.gensym.class.user.package* and *com.gensym.class.user.repository* properties, described earlier.

You use the JavaLink G2 DownloadInterfaces utility as follows:

```
java com.gensym.jgi.download.DownloadInterfaces
-force -stubs -host london -port 1111
-class COMPUTER-SUPPLIER
```

Because computer-supplier is not assigned to any module, JavaLink exports the class to this directory and assigns the class to this package in your G2 installation directory (Windows style):

```
\javalink\classes\com\gensym\classes\modules\unspecified
```

```
com.gensym.classes.modules.unspecified
```

Now, suppose you move the class definition for computer-supplier to the module named suppliers and download again. This time, JavaLink exports the class to this directory and assigns it to this package in your G2 installation directory:

```
\javalink\classes\com\gensym\classes\modules\suppliers
```

```
com.gensym.classes.modules.suppliers
```

Suppose you now add the following as the value of the module-annotation attribute of the supplier module's Module Information system table:

java-package-for-export is "com.acme.classes.modules.suppliers"

JavaLink exports the class to this directory and assigns it to this package in your G2 installation directory (Windows style):

\javalink\classes\com\acme\classes\modules\suppliers

com.acme.classes.modules.suppliers

Tip Gensym recommends that you specify a java-package-for-export for every KB module for which you wish to download classes. That way, the G2 module determines the Java package name that JavaLink uses when it downloads classes. This is important because developers might develop Java applications that rely on exported G2 classes being located in certain packages.

How JavaLink Finds a Previously Exported Class at Run Time

At run time, G2 might send a JavaLink client a G2 object reference or copy. JavaLink must then find a Java interface and implementation corresponding to the G2 class being sent.

JavaLink uses the following steps to ascertain the correct package name for the Java interface and implementation:

1. If the G2 class definition belongs to a module that defines a module-annotation specifying the attribute java-package-for-export, this will define the Java package name for the exported Java files. For example:

   java-package-for-export is "com.gensym.classes.modules.uilroot"

2. If no java-package-for-export is specified, JavaLink reads the class definition module name, converts it to a Java identifier, using all lowercase, and appends this name to the Java package name specified in the property *com.gensym.class.user.package*. If the class does not belong to any module, JavaLink uses the module named unspecified.

Examples

Suppose you have a class definition for a G2 class named engine-supplier that has no module and resides in a G2 running on host "*london*" on port *1111*. And suppose you are running on a Windows platform, using the default *com.gensym.class.user.package* and *com.gensym.class.user.repository* properties, described earlier.

Now, suppose a Java client needs to access an instance of the engine-supplier named mikes-engines.

JavaLink provides a convenience method, *getUniqueNamedItem*, which allows you to access a G2 object by name. Because engine-supplier has no module assignment, you would access the named item as follows:

```
import com.gensym.jgi.*;
import com.gensym.util.Symbol;

//Import the user class
import com.gensym.classes.modules.unspecified.EngineSupplier;

public class AccessEngineSupplier {

public static void main(String args[]) {

try {

    G2Connection connection =
       G2Gateway.getOrMakeConnection("london", "1111");

    EngineSupplier supplier =
        (EngineSupplier)getUniqueNamedItem
           (Symbol.intern("ENGINE-SUPPLIER"),
            Symbol.intern("MIKES-ENGINES"));
    //Do something

    } catch (G2AccessException e) {
       //Do some recover action
    }

}
```

Note For more information on the *getUniqueNamedItem* method and how to access G2, see Remote Procedure Calls.

Now, suppose the class definition for engine-supplier is defined in the module named suppliers. You would need to change the import statement in the above example to:

```
// Import the user class
import com.gensym.classes.modules.suppliers.EngineSupplier;
```

Suppose you specify the module-annotation of the suppliers module as:

java-package-for-export is “com.acme.classes.modules.suppliers”

You would need change the import statement in the above example to:

```
// Import the user class
import com.acme.classes.modules.suppliers.EngineSupplier;
```

You would also need to make sure this package directory is in your system’s Java *CLASSPATH*.

When a G2 Class Has Not Been Exported and is Passed at Run Time

JavaLink recursively attempts to select a Java interface and implementation from the G2 classes inheritance path.

For information on this process, see How JavaLink Resolves a G2 Class to a Java Class.

Note If a JavaLink client seems to hang, it might be automatically creating:

- A Java interface and implementation for a multiply inherited G2 class that has not previously been exported, or

- A Java implementation for a G2 class whose Java interface could be found but whose implementation has not previously been exported.

However, JavaLink creates an interface or implementation only once; subsequent access to the same G2 class causes little delay.

Creating a Local G2 Object in the Java Client

To create a local instance of a G2 class in Java, that is, an instance with no association to an existing G2 KB item, you use the following class:

```
com.gensym.jgi.download.G2StubResolver
```

To create the local instance, you call the following static method:

```
public static com.gensym.classes.Item
   createStubForG2Interface (Class g2InterfaceClass)
      throws G2StubCreationException;
```

For example, to create a local instance of a G2 integer-array, pass in the equivalent JavaLink interface, `com.gensym.classes.IntegerArray`, as follows:

```
import com.gensym.classes.IntegerArray;

public class Foo {
   public IntegerArray createG2LocalArray() {

    try {
    IntegerArray array = (IntegerArray)
      G2StubResolver.createStubForG2Interface
         (IntegerArray.class);
    // Setup array
    return array;

    } catch (G2StubCreationException e) {
    // Failure to create stub
    }
   }
}
```

3

Remote Procedure Calls

Describes how to make remote procedure calls between G2 JavaLink and G2.

Introduction

G2 JavaLink (JavaLink) and G2 can communicate with each other through remote procedure calls (RPCs) in one of two ways:

- JavaLink can invoke G2 procedures or methods.
- G2 can invoke Java methods.

You make remote procedure calls between JavaLink and G2 by using methods in the `G2Connection` interface in the `com.gensym.jgi` package. You can access an active `G2Connection` by calling the `G2Gateway.getOrMakeConnection` static method.

A number of the examples in this chapter require that you have the `jgidemo.kb` loaded. The `jgidemo.kb` requires certain utility KBs, located in the `\g2\kbs\utils\` directory (on Windows) and the `/g2/kbs/utils/` directory (on UNIX) of your G2 installation directory. One solution is to use the `-module-search-path` command-line argument when launching G2 and have it point to the location of the utilities that match the version of G2 that you are running.

For example, on Windows, you might launch G2 as follows, where *g2-install-dir* is your G2 installation directory:

```
g2.exe -module-search-path g2-install-dir\g2\kbs\utils\
-kb g2-install-dir\kbs\jgidemo.kb
```

Accessing G2 Procedures and Methods from Java

The `G2Connection` interface provides the following methods to support remote procedure calls from Java to G2:

- `callRPC`
- `startRPC`
- `setRemoteRPCReturnKind`

callRPC()

Calls a G2 procedure or method synchronously; this means it waits until the call has finished executing or a time-out occurs before allowing the current thread, which is waiting for the return value of the RPC, to continue.

Synopsis

```
public Object callRPC(Symbol g2_procedure_name,
                      Object args[],
                      int timeout)
   throws G2AccessException
```

Parameters

- `g2_procedure_name` is a `com.gensym.util.Symbol` that names the G2 procedure or method to be called.
- `args[]` is an `Object` array that contains the arguments to pass to the G2 procedure or method. Thus, all parameters passed to G2 must be object-wrapped. For example, if a G2 procedure defined a parameter to be of type integer, then `callRPC` would expect a `java.lang.Integer` for that parameter.
- `timeout` is the time in milliseconds to wait for the return, 0 for indefinite.

Exceptions

- `G2AccessException` is thrown when a problem occurs as a result of calling the G2 RPC.
- `TimeoutException` is thrown when the RPC takes longer than the timeout period.

Note There is a version of *callRPC* that does not take a *timeout* argument, in which case, the *defaultCommunicationTimeout* is used to create an RPC declaration, which is 20 seconds. Changes to *defaultCommunicationTimeout* do not affect the timeout on previously defined RPCs.

startRPC()

Starts a G2 procedure or method, ignoring any value that G2 attempts to return. This method is functionally identical to *callRPC* except that it does not return any value and it does not wait for the G2 procedure or method to complete before allowing another RPC call.

Synopsis

```
public void startRPC(Symbol g2_procedure_name, Object args[])
```

Parameters

- *g2_procedure_name* is a *com.gensym.util.Symbol* that names the G2 procedure or method to be started.
- *args[]* is an *Object* array that contains the arguments to pass to the G2 procedure or method. Thus, all parameters passed to G2 must be object-wrapped. For example, if a G2 procedure defined a parameter to be of type integer, then *startRPC* would expect a *java.lang.Integer* for that parameter.

Exceptions

This procedure does not throw any exceptions. Instead, if a communication problem occurs as a result of starting the G2 RPC, *startRPC* calls the user's *G2ConnectionListener.communicationError* listener, if it exists.

setRemoteRPCReturnKind()

Determines how G2 passes the return value of a remote procedure call (RPC). By default, the return value of an RPC call from Java to G2 is passed "by handle." By using this method, you can force a named G2 procedure or method to return a G2 object "by copy" such that references to the G2 object are local within Java.

For information on the difference between passing objects by handle and by copy, see Accessing G2 Objects by Handle or by Copy.

Synopsis

```
public void setRemoteRPCReturnKind (Symbol g2_procedure_name,
                                    int return_value_type)
```

Parameters

- *g2_procedure_name* is the *Symbol* that names the G2 procedure or method to be started.
- *return_value_type* is either *G2Gateway.PASS_BY_COPY* or *G2Gateway.PASS_BY_HANDLE*.

Reporting G2 Runtime Errors to Java

If during a *callRPC*, G2 fails to execute the specified procedure or method, JavaLink throws a *G2AccessException* to the waiting thread. The *G2AccessException* contains the text of the G2 error message that was reported.

Asynchronous errors are reported to Java as connection events. For more information, see Connection Events.

Setting the Maximum Number of Contexts

You can specify the maximum number of G2 Gateway (GSI) contexts allowed in the Java client, using a static method on the *G2Gateway* class. You can call the method at any time, either during startup or after connections have been made. The default value is 50, which is inherited from G2 Gateway. The syntax for the method is:

```
public void setGSIContextLimit(Int limit)
```

Calling a G2 Procedure From Java

The following Java class demonstrates how you can call a G2 procedure from Java.

The example calls the following G2 procedure:

```
A-G2-PROCEDURE(i: INTEGER, txt: TEXT, flt: FLOAT, sym: SYMBOL) =
    (FLOAT)
```

Here is the Java source for the class:

```
package com.gensym.demos.jgi;

import com.gensym.jgi.*;
import com.gensym.util.Symbol;

public class JavaCallingG2Procedure {
```

```
    // This variable holds the connection to G2
    private G2Connection g2Connection = null;

    /** Constructor */
    public JavaCallingG2Procedure(G2Connection connection) {
        this.g2Connection = connection;
    }
```

/* The method *aG2Procedure* uses *callRPC* to call a-g2-procedure across *g2Connection*. Because the argument and return types for a-g2-procedure are known, the JavaLink mapping from G2 types to Java types uses these mappings:

integer becomes *Integer*
text becomes *String*
float becomes *Double*
symbol becomes *Symbol*

callRPC accepts only object-wrapped types; however, this method conveniently accepts normal Java types, *int* and *double*, in this case, and object-wraps them internally. */

```
public double aG2Procedure(int i,
                           String txt,
                           double dbl,
                           Symbol sym)

    throws G2AccessException {
    Double ret = (Double)g2Connection.callRPC(
        Symbol.intern("A-G2-PROCEDURE"),
        new Object[] {new Integer(i),
                      txt,
                      new Double(dbl),
                      sym},
                      10000); // 10 seconds timeout

    return ret.doubleValue();
}
```

/* The example class needs to create an active *G2Connection* so it can use *callRPC*. We, therefore, define the class's *main* method, first, to connect to a G2, next, to create an instance of the example class, then, to call *aG2Procedure*. */

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

// Initiate a connection to a G2 server from
// Java by calling G2Gateway.getOrMakeConnection

try {

// Attempt to connect to G2, and wait until
// successful or timeout
G2Connection g2_connection =
    G2Gateway.getOrMakeConnection("localhost", "1111");
```

```
} catch (G2AccessException e) {
   System.out.println("Could not connect to G2, error
      was " + e.toString());
   System.exit(0);
}

// Once connected, create a new JavaCallingG2Procedure
JavaCallingG2Procedure G2procCall = new
   JavaCallingG2Procedure(g2_connection);

try {

// Call the G2 procedure
double retval =
   G2procCall.aG2Procedure(4, "Hi there from Java",
      100.34, Symbol.intern("JAVALINK"));

System.out.println("Return val is " + retval);

} catch (G2AccessException e) {
   System.out.println("Error occurred while calling G2
      Procedure, error was " + e.toString());

}
```

To compile the example, enter this command:

```
javac JavaCallingG2Procedure.java
```

To run the example, enter this command:

```
java com.gensym.demos.jgi.JavaCallingG2Procedure
```

You can find an extended version of this example in the JavaLink package *com.gensym.demos.jgi*. The example needs the G2 host and/or port to be specified. To run the example, issue the following command:

```
java com.gensym.demos.jgi.JavaCallingG2Procedure
   -g2host <hostname> [-g2port <portnumber>]
```

This example assumes you have a G2 running on *localhost* on port *1111*, which has *jgidemo.kb* loaded.

Calling G2 Methods From Java

To call a method on an existing G2 object, JavaLink must obtain a reference to a Java object that acts as a proxy for the G2 object.

Accessing G2 Objects by Handle or by Copy

By default, G2 passes the return value of an RPC call from Java to G2 "by handle," which means G2 sends a reference, or "handle" to that object to JavaLink. JavaLink then creates a Java stub class representing the original class containing the G2 object reference. Once the class exists, Java can access any attributes or methods defined on the G2 object via the property accessors and methods

available within the Java stub object. Access to the original G2 object via the Java stub object is valid as long as the *G2Connection* that created the Java stub is actively connected to G2, and the original G2 object exists and is active.

When G2 sends an object from G2 to Java "by copy," JavaLink sends the current object's attribute values from G2 to Java. JavaLink then creates a Java class representing the original G2 class and sets the current G2 attribute values set within the class. Any access to this class refers to local property values and does not affect the original G2 object. You can arrange to send objects and their return values from G2 to Java "by handle" in the remote procedure declaration in G2. By default, these objects are sent "by copy."

The method *setRemoteRPCReturnKind* on *G2Connection* enables a Java application to specify how G2 should return objects from *callRPC*. For example, the following Java source code instructs G2 to send any results from calling the G2 procedure another-g2-procedure "by copy:"

```
setRemoteRPCReturnKind(
   Symbol.intern("ANOTHER-G2-PROCEDURE"),
   G2Gateway.PASS_BY_COPY);
```

Example

The following class shows how to call a G2 procedure with this signature:

```
get-employee() = (class generic-r-employee)
```

JavaLink passes the return object to Java "by handle." The example shows how to call a method and set an attribute of the returned G2 object remotely from Java.

Because the example accesses the G2 class generic-r-employee from Java, you must first use JavaLink's G2 DownloadInterfaces utility to export a Java class that represents the definition of this class. This command creates the relevant Java class required to access this G2 class:

```
java com.gensym.jgi.download.DownloadInterfaces
-host localhost -port 1111 -class GENERIC-R-EMPLOYEE
```

This command assumes that the G2 running on the machine *localhost* on port *1111* has a KB loaded that contains a class definition for generic-r-employee.

As detailed in Mapping between G2 and Java, JavaLink generates an interface for generic-r-employee named *GenericREmployee*. Assume that generic-r-employee is defined in the jgidemo module and is, therefore, exported to the *com.gensym.classes.modules.jgidemo* package.

Here is the Java class that calls the G2 procedure:

```
package com.gensym.demos.jgi;

import com.gensym.jgi.*;
import com.gensym.util.Symbol;
```

```
import com.gensym.classes.modules.jgidemo.GenericREmployee;

public class JavaCallingG2Methods {

   // This variable holds the connection to G2
   private G2Connection g2Connection = null;

   // It`s good practice to lookup symbols that will be
   // used often
   private static Symbol getEmployeeSymbol =
      Symbol.intern("GET-EMPLOYEE");

   /** Constructor */
   public JavaCallingG2Methods(G2Connection connection) {
      this.g2Connection = connection;

   }

   /* This method calls the G2 procedure GET-EMPLOYEE across
   g2Connection. Notice that the return value of the RPC
   call is cast to a GenericREmployee. */

   public GenericREmployee getEmployee()
      throws G2AccessException {

      // By default, the retured object is passed by handle
      return(GenericREmployee)g2Connection.callRPC(
         getEmployeeSymbol,
         new Object[0],
         10000); // 10 second timeout

   }

   /* Assuming the example creates an active G2Connection,
   the class`s main method first calls getEmployee to get a
   reference to a GENERIC-R-EMPLOYEE, then calls its G2
   method RECORD-COURSE-TAKEN, then sets its Title
   attribute. */

   JavaCallingG2Methods G2ProcCaller = new
      JavaCallingG2Methods(g2_connection);

   // Get the employee
   GenericREmployee employee = G2ProcCaller.getEmployee();

   boolean ok = employee.recordCourseTaken("G2 Javalink");

   // Set the employee's new Title
   employee.setTitle("G2 Java Software Developer");
```

When JavaLink exports the Java interface *GenericREmployee*, it automatically reads any current generic-r-employee attribute and method definitions and creates the relevant Java signatures, as follows:

G2 Method and Attribute	Java Method and Attributes
RECORD-COURSE-TAKEN (course-title: TEXT) = (TRUTH-VALUE)	*public boolean recordCourseTaken (String title);*
Title as Text, initial value "Engineer"	*public void setTitle(String Txt);* *public String getTitle();*

Referring to G2 Objects by Name

For convenience, *G2Connection* also defines this method, which returns a G2 object of a given class and name:

```
public com.gensym.classes.Item getUniqueNamedItem(
   Symbol className,
   Symbol itemName)
```

For example:

```
// Get a reference to a GENERIC-R-EMPLOYEE called
// MICHAEL

   GenericREmployee employee = (GenericREmployee)
   g2_connection.getUniqueNamedItem(
      Symbol.intern("GENERIC-R-EMPLOYEE"),
      Symbol.intern("MICHAEL"));

// Set Michael's new Title
employee.setTitle("Java Software Developer");
```

You can find an extended version of the examples given in this section in the JavaLink package *com.gensym.demos.jgi*. The example needs the G2 host and/or port to be specified. To run the example, enter this command:

```
java com.gensym.demos.jgi.JavaCallingG2Methods
   -g2host <hostname> [-g2port <portnumber>]
```

This example assumes you have a G2 running on host *localhost* on port *1111*, which has *jgidemo.kb* loaded.

Using callRPC() to Call a G2 Method

If the first element of the *args[]* array passed as a parameter to *callRPC* is a Java *Object*, G2 attempts to call a method on the *Object*. The *Object* represents a G2 object exported to Java by JavaLink, such as *GenericREmployee*, in the following example.

This Java code calls the method record-course-taken on the generic-r-employee referred to by *employee*:

```
Boolean ok = g2_connection.callRPC(
   Symbol.intern("RECORD-COURSE-TAKEN"),
   new Object[] {employee, "G2 Part 1"},
   10000);
```

Calling G2 Methods with Variable Arguments

You can define G2 methods with the same name and object class (first argument), but with a differing number of arguments. When calling a G2 method by using *callRPC*, JavaLink calls the appropriate method depending on the number of arguments passed.

For example, the following method calls the version of a-method with one argument, where the argument is the employee:

```
callRPC(Symbol.intern("A-METHOD"), new Object[] {employee});
```

However, this method calls the version of a-method that takes three arguments:

```
callRPC(Symbol.intern("A-METHOD"),
   new Object[] {employee, 1, "information"});
```

Accessing Java Methods from G2

JavaLink can register any public method on a public class as callable by G2.

Registering a Java Method as Callable from G2

You use the *G2Connection* method *registerJavaMethod* to register a Java method as callable by G2 via a G2 remote procedure call. The two variants of this method allow you to register Java methods with:

- A variable number of arguments, which uses a dynamic lookup to find the registered method.
- A specific signature, which uses a pre-looked declaration.

Note For the most optimal performance, we recommend that you register specific methods by using the second variation of *registerJavaMethod*, which takes a *java.lang.reflect.Method* instead of a *java.lang.String* as the method name argument. Registering a specific method saves significant time, because JavaLink does not need to look up the registered method each time. On the other hand, registering a method by using a String allows JavaLink to dispatch a method call dynamically based on the method name and the arguments sent from G2.

When calling a Java method from G2, JavaLink passes the *Object* argument from G2 to Java according to the RPC declaration in G2. Unless the RPC declaration specifically declares to send objects as handle, by default, JavaLink sends objects and return values "by copy."

Variable Argument Declaration

In this version of *registerJavaMethod*, G2 can call any Java method of a given name found on a target object. This version of the method causes the RPC call from G2 to use a "dynamic" lookup to find a method with the registered name that best matches the RPC parameters sent by G2.

Synopsis

```
public void registerJavaMethod(Object target,
                               String method_name,
                               Symbol g2_RPD_name);
```

Parameters

- *target* is the Java *Object* instance on which the method will be called.
- *method_name* is the Java method name that is to be called on the *target*.
- *g2_RPD_name* is the value of the name-in-remote-system attribute of the G2 remote-procedure-declaration to be associated with any method named by *method_name* found on *target*.

When G2 attempts to call or start the specified remote-procedure-declaration (RPD) across a gsi-interface, JavaLink attempts to call a method with this signature:

```
[<target>].<method_name>(params_from_G2_call...)
```

JavaLink calls *method_name* with the number and type of parameters sent from G2 during the G2 RPC call. If *target* is given and does not have the corresponding *method_name* with the correct parameter signature, JavaLink raises a *java.lang.NoSuchMethodException*.

Note JavaLink dynamically dispatches methods by using the same "best method match algorithm" as used by Java, which is best described in *The Java Language Specification* by James Gosling, Bill Joy, and Guy Steele.

For example, suppose you have a Java class named *Stock* with a method named *getStockPrice*, which has multiple definitions defined with several signatures and a setup method that registers *getStockPrice* as visible to G2:

```
public class Stock {

   public double getStockPrice(Symbol stock_sym_name) {...}

   public double getStockPrice(String stock_long_name) {...}

   public double getStockPrice(int stock_number) {...}

   public double getStockPrice(StockObject stock_object) {...}

   public void setup(G2Connection g2_connection) {
      Symbol g2RPDName = Symbol.intern("GET-STOCK-PRICE");
      g2_connection.registerJavaMethod(this,
                                       "getStockPrice",
                                       g2RPDName);
   }

}
```

In G2, suppose you declare a remote-procedure-declaration as follows:

```
declare remote get-stock-price(all remaining item-or-value ) = (float)
```

You can then call the Java method from a G2 procedure, method, or action, as follows:

```
get-gensym-stock-price(javalink-interface: class
    gsi-interface) = (float)
price: float;

begin

    price = call get-stock-price(the symbol gnsm)
        across javalink-interface;
    return price;

end;
```

As a symbol is being passed to get-stock-price in the above G2 procedure, JavaLink calls this version of the *getStockPrice* method on the *Stock* class:

```
public double getStockPrice(Symbol stock_sym_name) {...}
```

This table describes which *getStockPrice* method G2 would call for different parameter types sent from G2:

This call from G2...	Calls this method in Java...
get-stock-price (the symbol gnsm)	*getStockPrice* *(Symbol stock_sym_name)*
get-stock-price ("gensym corporation")	*getStockPrice* *(String stock_long_name)*
get-stock-price (3)	*getStockPrice* *(int stock_number)*
get-stock-price (stock-object)	*getStockPrice* *(StockObject stock_object)* **Note**: *StockObject* must be exported from G2.

Pre-Looked Up Declaration

This version of *registerJavaMethod* registers a specific Java method of a target object as callable by G2. In this case, the arguments sent by G2, when converted to Java types, must match the signature of the registered Java method.

Synopsis

```
public void registerJavaMethod(Object target,
                               Method java_method,
                               Symbol g2_RPD_name,
                               boolean call_in_new_thread);
```

Parameters

- *target* is the Java *Object* instance on which the method will be called.
- *java_method* is a *java.lang.reflect.Method* object for the Java method to be registered.
- *g2_RPD_name* is the name of the G2 remote-procedure-declaration to be associated with *java_method*.
- *call_in_new_thread*, when set to *True*, instructs JavaLink to create a new Java thread especially for the execution of the Java method. By default, JavaLink queues a G2 RPC call to Java with all other method call requests over a particular *G2Connection*. If a Java method is known to take a significant amount of time to execute, you might want to ask JavaLink to create a new thread for its execution, so as not to hold up the execution of any other RPC call requests waiting on the *G2Connection* RPC call queue.

When G2 attempts to call or start the specified remote-procedure-declaration (RPD) across a gsi-interface, connected using JavaLink, JavaLink attempts to call a method with this signature:

```
[<target>].<java_method>(params...)
```

JavaLink calls *java_method* with the number and type of parameters sent from G2 during the G2 RPC call. If *java_method* does not have the correct parameter signature, JavaLink raises a *java.lang.NoSuchMethodException*.

This example shows the *Stock* class using this variant of *registerJavaMethod*:

```
public class Stock {

   public double getStockPrice(Symbol stock_sym_name) {...}

   public double getStockPrice(String stock_long_name) {...}

   public double getStockPrice(int stock_number) {...}

   public double getStockPrice(StockObject stock_object) {...}

   public void setup(G2Connection g2_connection) {
      try {

         java.lang.reflect.Method method =
            this.getClass().getDeclaredMethod(
               "getStockPrice",
               new Class[] {Symbol.class});

         g2_connection.registerJavaMethod(
            this,
            method,
            Symbol.intern("GET-STOCK-PRICE"),
            false);

   } catch (NoSuchMethodException e) {

      System.err.println("Could not find Method
         getStockPrice"); }

   }

}
```

In this case, G2 can only call *getStockPrice(Symbol)*. All other variants of *getStockPrice* are inaccessible to G2. However, because JavaLink does not need to search for *getStockPrice* at run time, JavaLink can call *getStockPrice* significantly faster than when it is registered using the variable argument method.

You can find another example of G2 calling Java in the JavaLink package *com.gensym.demos.jgi*. To run the example, enter the following command:

```
java com.gensym.demos.jgi.G2CallingJavaMethod
```

This example assumes you have a G2 running, which has *jgidemo.kb* loaded. You then connect from G2 to this Java process.

Reporting Java Exceptions to G2

Any Java exceptions that occur when a G2 procedure calls a Java method are reported to the current procedures error handler. You can trap these errors by using the G2 on error statement. The error text sent to the on error branch contains the message and trace of the Java exception generated.

At all other times, any exceptions raised as a result of G2 calling into Java are reported to G2's Operator Logbook.

Using Multiple Java Threads to Access G2

JavaLink is thread-safe, which means that multiple Java threads can access the same *G2Connection*. This means that different threads can call and start the same or different G2 procedure or method at any time.

Note In the current version of JavaLink, all network communication from Java to G2 is via a single thread. This thread also handles all data marshalling of data types and objects. Once data marshalling is completed, other threads handle the execution of G2 or Java methods and do not prevent further communication.

An execution thread is defined for each active *G2Connection*. This thread maintains a queue of all current requests from the *G2Connection* to execute Java methods. As each Java method completes execution, the queue is emptied.

You can find an example of multiple Java threads calling a G2 procedure in the JavaLink package *com.gensym.demos.jgi*. The example requires that you specify the G2 host and/or port. To run the example, enter the following command:

```
java com.gensym.demos.jgi.JavaThreadsCallingG2
   -g2host <hostname> [-g2port <portnumber>]
```

This example assumes you have a G2 running on *localhost* on port *1111*, which has *jgidemo.kb* loaded.

4

Passing Messages

Describes how G2 JavaLink and G2 can exchange messages.

Introduction

G2 JavaLink (JavaLink) enables you to pass messages through a `G2Connection`, as follows:

- Send a message to the G2 Message Board.
- Send an error message to the G2 Operator Logbook.
- Receive a message from G2 via the G2 inform statement.

Sending a Message to the G2 Message Board

To send a message to the Message Board of a G2 actively connected to a *G2Connection*, use:

```
public void returnMessage(String message);
```

Example

```
import com.gensym.jgi.*;

/**
* Simple JavaLink example to send a message to G2.
**/

public class JavaLinkExample {
   public static void main(String args[]) {
      G2Connection g2_connection = null;
      try {
         g2_connection =
            G2Gateway.getOrMakeConnection("localhost", "1111");
         } catch (G2AccessInitiationException e) {
            System.out.println("Problem connecting to G2
               exception was:" + e.toString());
         }
         g2_connection.returnMessage("Hi from JavaLink");
         System.exit(0);
   }
}
```

Sending an Error Message to the G2 Operator Logbook

To send an error message to the Operator Logbook of a G2 actively connected to a *G2Connection*, you use this method:

```
public void reportLogBookErrorMessage(Symbol error_symbol,
                                       String error_message)
```

Example

```
import com.gensym.jgi.*;
import com.gensym.util.Symbol;

/**
* Simple JavaLink example to send a Logbook error to G2.
**/

public class JavaLinkExample {
    public static void main(String args[]) {
        G2Connection g2_connection = null;
    try {
            g2_connection = G2Gateway.getOrMakeConnection
                ("localhost", "1111");
        } catch (G2AccessInitiationException e) {
            System.out.println("Problem connecting to G2 exception
                was:" + e.toString());
        }
    g2_connection.reportLogBookErrorMessage(
        Symbol.intern("JAVALINK-ERROR"),
        "JavaLink reports that an error has occurred --
            please investigate");
        System.exit(0);
    }
}
```

Receiving a Message from G2

When a *G2Connection* for a Java class implements the *G2ConnectionListener* interface, it receives a *g2MessageReceived* event from G2 when the G2 inform statement is used on a GSI message server, whose gsi-interface-name attribute names the gsi-interface used for the JavaLink connection. For more information, see Connection Events.

In addition, you must define a GSI message server in G2 whose gsi-interface-name attribute names the gsi-interface used for the JavaLink connection between G2 and the Java class that implements the *G2ConnectionListener* interface. A GSI message server is a G2 class that inherits from the G2 mixin class gsi-

message-service and at least one G2 class from which you can create instances. For more information on how to do this, see "Creating and Configuring GSI Message Servers" and "Running an Inform Action on a GSI Message Server" in the *G2 Gateway Bridge Developer's Guide*.

5

Listening for Changes in G2 Items

Describes how G2 JavaLink listens for changes in G2 items.

Introduction

The following G2 JavaLink interfaces provide notification when certain aspects of a G2 item change:

- `com.gensym.util.ItemListener` provides notification when an item's attribute has changed and when an item is deleted.
- `com.gensym.util.VariableValueListener` provides notification when there is a change in the last-recorded-value attribute of a variable-or-parameter or any subclass.

These interfaces provide a convenient and efficient way of listening for item changes. JavaLink subscribes to the specified G2 item. G2 then informs JavaLink when any changes occur. JavaLink does not, therefore, continually poll G2.

The ItemListener Interface

Any Java class that implements the *com.gensym.util.ItemListener* interface can listen for item changes by adding itself to a G2 item's listener list.

Every G2 item received from G2 by JavaLink is a subclass of *com.gensym.classes.Item*, which defines the following methods:

- *public void addItemListener (ItemListener il)*
  throws G2AccessException

  Adds an *ItemListener* that is notified of modifications to an item. The listener is notified of the initial state of the item, whenever any of the attributes of the item change, or when the item is deleted. In G2 5.x, adding yourself as an item listener downloaded a snapshot of the entire state of an item, including the current values of all attributes and virtual attributes, including histories. In G2 6.0, histories are not downloaded.

- *public void addItemListener (ItemListener il,*
  Symbol[] attributeNames)
  throws G2AccessException

  Adds an *ItemListener* that is notified of modifications to specific attributes of an item. The listener is notified of the initial state of the item, whenever any of the requested attributes of the item change, or when the item is deleted. In G2 5.x, adding yourself as an item listener downloaded a snapshot of the state of the specified attributes of the item. In G2 6.0, histories are not downloaded.

- *public void removeItemListener (ItemListener il)*
  throws G2AccessException

  Removes an *ItemListener* from the *ItemListener* list for an item. The *ItemListener* no longer receives *ItemListener* events.

The *ItemListener* interface, located in the *com.gensym.util* package, contains the following three listener methods:

- *public void receivedInitialValues (ItemEvent e)*

  Invoked when an *ItemListener* is added by using the *addItemListener* method. As soon an *ItemListener* is added to an item's listener list, it will be informed of the item's current attribute values. The current attribute values are embedded within the *ItemEvent* argument to this method. To obtain the current values of the item, call *getNewValue* on the *ItemEvent*, which can be cast to a *com.gensym.util.Structure* containing attribute-value pairs for each attribute and virtual attribute of the G2 item.

- *public void itemModified (ItemEvent e)*

  Invoked when an item's attributes are modified. The *ItemEvent* argument to this method contains a *com.gensym.util.Sequence* containing an attribute-value pair describing the attribute that has changed, which is called a **denotation**, and the attribute's new value.

  For example, assume a G2 class test-class has an attribute called an-attribute and a Java *ItemListener* is listening for changes to an instance of test-class. If the value of an-attribute is changed to the text "London Bridge," the *itemModified* method of the *ItemListener* would be called.

  The following code shows *ItemListener*'s implementation of the *itemModified* listener method:

```
public void itemModifed (ItemEvent e) {

   System.out.println("Received an itemModified Event");

   Sequence denotation = e.getDenotation();
   System.out.println("Denotation:" + denotation.toString());
   Structure newValue = (Structure)e.getNewValue();
   System.out.println("The New Value is: " +
      newValue.toString());

}
```

  When the event occurs, the following text appears in a command window:

```
Received an itemModified Event
Denotation: Structure {TYPE:ATTRIBUTE, NAME:AN-ATTRIBUTE}
The New Value is: London Bridge
```

- *public void itemDeleted (ItemEvent e)*

  Invoked when the item has been deleted in G2.

Example

The following Java class connects to a G2, obtains a G2 item, then adds an *ItemListener* as a listener to item changes:

```
package com.gensym.demos.jgi;

import com.gensym.jgi.G2Access;
import com.gensym.jgi.G2Gateway;
import com.gensym.util.ItemEvent;
import com.gensym.util.ItemListener;
import com.gensym.util.Sequence;
import com.gensym.util.Symbol;
import com.gensym.core.CmdLineArgHandler;
import com.gensym.classes.Item;
```

```
public class SimpleSubscription {

   // The correct way to make a Symbol
   public static final Symbol ITEM_ =
      Symbol.intern ("ITEM");

   public static final Symbol A_TEST_CLASS_INSTANCE_ =
      Symbol.intern ("A-TEST-CLASS-INSTANCE");

   public static void main(String[] args) {

      // Read command-line
      CmdLineArgHandler cmdLine =
         new CmdLineArgHandler (args);

      // Information for connecting to G2
      String url   = cmdLine.getOptionValue ("-url");
      String host  = cmdLine.getOptionValue ("-g2host");
      host = (host == null ? "localhost" : host);
      String port  = cmdLine.getOptionValue ("-g2port");
      port = (port == null ? "1111" : port);

      // Information to find a unique named item in G2
      String classString = cmdLine.getOptionValue
         ("-class");
      String nameString = cmdLine.getOptionValue ("-name");

      // Convert the strings to symbols
      Symbol clazz = (classString == null ? ITEM_ :
         Symbol.intern(classString.toUpperCase()));
      Symbol name = (nameString == null ?
         A_TEST_CLASS_INSTANCE_ :
         Symbol.intern(nameString.toUpperCase()));

      // Do actual work

      try {

         // Connect to G2
         G2Access connection =
            G2Gateway.getOrMakeConnection(url, host, port);

         // Obtain a Java object that represents a G2 item
         Item item =
               connection.getUniqueNamedItem(clazz, name);
         // Define a simple implementation of the
         // ItemListener interface
         ItemListener il = new ItemListener() {

            public void itemModified(ItemEvent e) {

              System.out.println("itemModified: " + e);

            }
```

```
            public void itemDeleted(ItemEvent e) {
              System.out.println("itemDeleted: " + e);
            }
            public void receivedInitialValues(ItemEvent e) {
              System.out.println("InitialValues:" + e);
            }
          }
        // Initiate the subscription
        item.addItemListener(il);
        } catch (Exception ex) {
           System.out.println(ex.getMessage());
           ex.printStackTrace();
        }
    }
}
```

To compile the code, enter this command:

```
javac SimpleSubscription.java
```

To test the example, enter the following command:

```
java com.gensym.demos.jgi.SimpleSubscription
   -name A-TEST-CLASS-INSTANCE
```

Assuming a G2 item called a-test-class-instance exists within the G2 server on *localhost* on port 1111, the Java client outputs messages to the Java console when a-test-class-instance's attributes change.

You can find this example, *SimpleSubscription.java*, in the JavaLink package *com.gensym.demos.jgi*.

The VariableValueListener Interface

Any Java class that implements the *com.gensym.util.VariableValueListener* interface is notified that there is a change in the last-recored-value attribute of a specified variable-or-parameter or any subclass.

Note An *ItemListener* is *not* notified when the attribute last-recorded-value is changed in an instance of variable-or-parameter or a subclass.

Every variable-or-parameter received from G2 by JavaLink is a subclass of *com.gensym.classes.VariableOrParameter*, which defines the following methods:

- *public void addVariableValueListener (VariableValueListener il) throws G2AccessException*

  Adds a *VariableValueListener*, which is notified when the last-recorded-value attribute of the variable-or-parameter changes.

- *public void removeVariableValueListener (VariableValueListener il) throws G2AccessException*

  Removes the *VariableValueListener* from the *VariableOrParameter*'s listener list. The *VariableValueListener* no longer receives notification of *VariableValueEvent*s.

The *VariableValueListener* interface, located in the *com.gensym.util* package contains the following two listener methods:

- *public void receivedInitialValue (VariableValueEvent e)*

  Invoked when a *VariableValueListener* is added by calling the *addVariableValueListener* method. As soon a *VariableValueListener* is added to a *VariableOrParameter*'s listener list, it is informed of the current value of the last-recorded-value attribute. You can obtain the current value of the last-recorded-value attribute by calling *getNewValue* on the *VariableValueEvent* argument to this method.

- *public void valueChanged(VariableValueEvent e)*

  Invoked when the last-recorded-value attribute of the *VariableOrParameter* changes. You can obtain the current value of the last-recorded-value attribute by calling *getNewValue* on the *VariableValueEvent* argument to this method.

You can find an example of a *VariableValueListener* in the class *SimpleVariableSubcription.java* in the JavaLink package *com.gensym.demos.jgi*.

6

Connection Events

Describes how G2 JavaLink responds to connection events.

Introduction

Any Java class can register interest in listening for predefined connection events, which can occur while using a G2 JavaLink (JavaLink) `G2Connection`. The `com.gensym.jgi.G2ConnectionListener` interface defines listener methods for each defined event.

Connection Events

The following sections describe each event and the corresponding method in `G2ConnectionListener` that the `G2Connection` calls to notify the listening Java class that the event has occurred.

Communication Error Has Occurred

This event occurs when an asynchronous error has occurred in the G2 communications interface for the *G2Connection*. The following listener method in *G2ConnectionListener* is called to inform the listener of this event.

```
public void
communicationError(G2CommunicationErrorEvent)
```

The *G2CommunicationErrorEvent* describes the error that occurred.

The Connection Between G2 and JavaLink has Closed

When Java terminates or G2 closes an active connection, the closed event occurs. The following listener method in *G2ConnectionListener* is called to inform the listener of this event.

```
public void g2ConnectionClosed(G2ConnectionEvent)
```

The Connection Has Been Made With a G2 Server

This event occurs when a connection is established between a *G2Connection* and a G2, regardless of whether G2 or Java initiated the connection. The following listener method in *G2ConnectionListener* is called to inform the listener of this event.

```
public void g2ConnectionInitialized(G2ConnectionEvent)
```

The Connected G2 Has Paused

This event occurs when the connected G2 is paused. The following listener method in *G2ConnectionListener* is called to inform the listener of this event.

```
public void g2IsPaused(G2ConnectionEvent)
```

The Connected G2 Has Resumed

This event occurs when the connected G2 resumes after a pause event. The following listener method in *G2ConnectionListener* is called to inform the listener of this event.

```
public void g2IsResumed(G2ConnectionEvent)
```

The Connected G2 Has Been Reset

This event occurs when the connected G2 has been reset. The following listener method in *G2ConnectionListener* is called to inform the listener of this event.

```
public void g2IsReset(G2ConnectionEvent)
```

The Connected G2 Has Been Started

This event occurs when the connected G2 is started. The following listener method in *G2ConnectionListener* is called to inform the listener of this event.

```
public void g2IsStarted(G2ConnectionEvent)
```

The Connected G2 Has Sent a Message

This event occurs when the connected G2 sends a message via an inform statement. The following listener method in *G2ConnectionListener* is called to inform the listener of this event.

```
public void g2MessageReceived(G2ConnectionEvent)
```

A Read Blockage Event Occurred

This event occurs when the network cannot deliver data that Java is attempting to write to G2. The following listener method in *G2ConnectionListener* is called to inform the listener of this event.

```
public void readBlockage(G2ConnectionEvent)
```

A Write Blockage Event Occurred

This event occurs when the network cannot deliver data that G2 is attempting to write to Java. The following listener method in *G2ConnectionListener* is called to inform the listener of this event.

```
public void writeBlockage(G2ConnectionEvent)
```

Registering a G2ConnectionListener

Any Java class that implements *G2ConnectionListener* can register itself to listen for these events by using the *G2Connection* method *addG2ConnectionListener*.

A *G2ConnectionListener* can remove interest by calling *removeG2ConnectionListener*.

Example

```
import com.gensym.jgi.*;
import com.gensym.util.*;

public class ConnectionEventListener implements
    G2ConnectionListener {

    // This variable holds the connection to G2
    private G2Connection g2Connection = null;

/** Constructor */
public ConnectionEventListener(G2Connection connection) {
    this.g2Connection = connection;

    // Register to listen to any connection events
    // (G2 Pause, Resume, etc...)
    connection.addG2ConnectionListener(this);
}

// G2ConnectionListener implementations
/** Called when the G2 connection has been paused.
*/
public void g2IsPaused(G2ConnectionEvent e) {
    System.out.println("G2 has been paused");
}

/** Called when the G2 connection has been resumed.
*/
public void g2IsResumed(G2ConnectionEvent e) {
    System.out.println("G2 has been resumed");
}

/** Called when the G2 connection has been reset.
*/
public void g2IsReset(G2ConnectionEvent e) {
    System.out.println("G2 has been reset");
}

/** Called when the G2 connection has been started.
*/
public void g2IsStarted(G2ConnectionEvent e) {
    System.out.println("G2 has been started");
}


/** Called when G2 sends a message to the G2Connection.
*/
public void g2MessageReceived(G2ConnectionEvent e) {
    System.out.println("Message received from G2 was: " +
        e.getMessage());
}
```

```
/** Called when the network cannot deliver data that
*   Java is attempting to write to G2.
*/
public void readBlockage(G2ConnectionEvent event) {
    System.out.println("Network read Blockage event,
                            state" + e.getState());
}

/** Called when the network cannot deliver data that
* G2 is attempting to write to Java.
*/
public void writeBlockage(G2ConnectionEvent event) {
    System.out.println("Network write Blockage event,
                            state" + e.getState());
}

/** Called when an asynchronous error has occurred in
* the G2 communications interface for the G2Connection.
/
public void communicationError(G2CommunicationErrorEvent
                                    error_event) {
    System.out.println(error_event.getMessage());
}

/** This method is called when the G2 connection has
* been shutdown.
*/
public void g2ConnectionClosed(G2ConnectionEvent e) {
    System.out.println("G2 connection has closed");
}


/** A connection has been established between this
* connection and a G2 GSI-INTERFACE.
*/
public void g2ConnectionInitialized(G2ConnectionEvent e) {
    String remote_process_string = e.getMessage();

    // Send a "hi" message to G2's Message Board
    g2Connection.returnMessage(
        "You have Connected to ConnectionEventListener
        (rpis = " + remote_process_string + ")\n
        Hi There from Java VM, how are you G2?");
}
```

```
public static void main(String args[]) {
   // A connection to a G2 server can be initiated from
   // Java using G2Gateway.getOrMakeConnection.

   try {

      // Attempt to connect to G2, and wait until
      // successful or timeout.

      G2Connection g2_connection =
         G2Gateway.getOrMakeConnection("localhost",
            "1111");
   } catch (G2AccessException e) {
      System.out.println("Could not connect to G2, error
         was " + e.toString());
   }

   // Connected OK, create a ConnectionEventListener.
   ConnectionEventListener listener = new
      ConnectionEventListener(g2_connection);

   // The main thread will terminate here, but the
   // G2Connection thread is still alive, so we will
   //receive G2ConnectionListener events.
}


}
```

7

Using Java RMI to Communicate with G2

Describes how G2 JavaLink communicates with G2, using Java RMI.

Introduction

Remote Method Invocation (RMI) enables a programmer to create distributed Java-to-Java applications, in which other Java virtual machines, possibly on different hosts, can invoke the methods of remote Java objects. Once it obtains a reference to the remote object, a Java program can make a call on a remote object. You can obtain object references by either:

- Looking up the remote object in the bootstrap naming service provided by RMI.
- Receiving the reference as an argument or a return value.

A client can call a remote object in a server, and that server can also be a client of other remote objects. RMI uses object serialization to marshal and unmarshal

parameters, and it does not truncate types, supporting true object-oriented polymorphism.

Note For more information about Java RMI, visit *http://www.javasoft.com/products/jdk/rmi/index.html*.

G2 JavaLink (JavaLink) provides a set of classes that allow access to G2 via Java RMI. These classes reside in the *com.gensym.jgi.rmi* package.

Before attempting to connect a Java client, you must set up a three-tier communication link with G2.

To set up a three-tier communication link:

1. Start an RMI registry, which is the naming service.
2. Start an RMI server, the second tier, which connects to G2(s), the first tier.
3. Start the RMI client, the third tier, which connects to the RMI server running on the second tier.

The following steps explain each of these steps in more detail.

Starting an RMI Registry

The RMI registry is a naming service. RMI servers use the registry to bind remote objects to names. Clients can look up remote objects and make remote method invocations.

The *rmiregistry* command creates and starts a remote object registry on the current host.

To start an RMI registry on your machine:

➔ Execute the following command in a command window or shell:

```
rmiregistry
```

Starting an RMI Server

JavaLink defines an RMI server interface:

com.gensym.jgi.rmi.G2RMIAccessBroker

This interface allows you to create *G2Connection*s between G2 servers and Java RMI clients. Use the default implementation supplied with JavaLink, *G2RMIAccessBrokerImpl*, to connect to G2 via RMI. After parsing the command line, this RMI server, or connection broker, exports itself and waits for connection requests. The *G2RMIAccessBroker* normally shares connections between Java

clients. It also caches Java stubs for G2 objects that have been previously passed between G2 and third-tier clients.

Note The Java stubs that JavaLink creates to represent G2 objects have been designed so you can pass them efficiently between Java clients, using RMI.

The RMI server, or middle tier, needs to be able to find the class definition for the classes that it will create. To do this, if you do not already have the *CLASSPATH* environment variable set up, use the *-classpath* argument in the command that starts the RMI server. It also needs to be able to locate the *JgiInterface* shared library by including this library on the Path.

To start the JavaLink RMI server:

➔ Execute the following command in a command window or shell:

```
java com.gensym.jgi.rmi.G2RMIAccessBrokerImpl
    -tsName <rmiservername>
```

The default name of the RMI server is its full class name, *com.gensym.jgi.rmi.G2RMIAccessBrokerImpl*. Because you must specify the RMI server when making a connection, the *-tsName* option enables you to provide a simple alias name. For example:

```
java -classpath g2-install-dir\javalink\classes\javalink.jar;
   g2-install-dir\javalink\myclasses.jar
   com.gensym.jgi.rmi.G2RMIAccessBrokerImpl
   -tsName demoserver
```

The host name of the machine that the RMI server is running on and the *rmiservername* specify the connection broker's URL. *G2RMIAccessBrokerImpl* automatically registers itself in the RMI registry on startup. If successful, a message similar to the following appears in the command window or shell:

```
rmi://boston/demoserver bound in registry
```

Note Registration of *G2RMIAccessBroker* in the RMI registry might take more than one minute on platforms running Windows, if Windows cannot find a naming service such as DNS to resolve the host name.

Connecting to G2 as an RMI Client

Once the RMI registry and *G2RMIAccessBroker* have been started, you can then create G2 connections from an RMI client.

To do this, you call a version of *G2Gateway.getOrMakeConnection* that takes three arguments:

```
public static G2Connection getOrMakeConnection(String URL,
                                               String g2Host,
                                               String g2Port)
```

The first argument can take *null* or an RMI URL that specifies a path to a *G2RMIAccessBroker* bound in an RMI registry. The URL takes the following form, where *hostname* is the host name where an RMI registry is running and *rmiservername* is the name of a *G2RMIAccessBroker* previously registered:

rmi://hostname/rmiservername

For example:

rmi://boston/demoserver

Tip This is the same URL that the *G2RMIAccessBroker* prints when it is bound to the RMI registry, as described in the previous section.

Note You can omit the *rmi:* in the command because it is assumed.

Assuming that a *G2RMIAccessBroker* has been registered in the RMI registry on machine *london* with the alias name *ukserver*, the following call to *getOrMakeConnection* attempts to return a *G2Connection* to a G2 running on machine *manchester* on port 1111 from *ukserver*'s *G2RMIAccessBroker*:

```
G2Connection connection =
   G2Gateway.getOrMakeConnection("//london/ukserver",
                                 "manchester",
                                 "1111");
```

Example of Calling a G2 Method Over RMI

You can call G2 methods by using RMI to find a *G2Connection*. For this example, assume that G2 and the RMI registry are running on the host named *boston*.

To call a G2 method over RMI:

1. In a command window or shell on host *boston*, start up the RMI registry for machine *boston*:

   ```
   rmiregistry
   ```

2. From another command window or shell on *boston*, start up the JavaLink second tier:

   ```
   java com.gensym.jgi.rmi.G2RMIAccessBrokerImpl
      -tsName mitserver
   ```

3 Wait for the *G2RMIAccessBrokerImpl* to print the following:

```
rmi://boston/mitserver bound in registry
```

Note Registration of *G2RMIAccessBroker* in the RMI registry might take more than one minute on platforms running Windows, if Windows cannot find a naming service such as DNS to resolve the host name.

4 From a third command window or shell prompt, which can be on another machine, as long as the *Javalink.jar* is on the *CLASSPATH*, enter:

```
java com.gensym.demos.jgi.JavaCallingG2Methods
   -g2host boston
   -g2port 1111
   -tsName //boston/mitserver
```

Note In steps 2 and 4, the argument is case-sensitive: *-tsName* is correct.

The Java client connects to the G2 running on *boston* via a *G2Connection* created by the *G2RMIAccessBroker* and sent to the Java client using RMI, rather than using JavaLink's native G2 interface.

The connection to the RMI server might take more than a minute on platforms running Windows, if a naming service such as DNS cannot be found to resolve the host name.

Note The method *registerJavaMethod* on *G2Connection* is not currently supported when called from an RMI Java client.

Detecting Middle Tier Connection Closed Events

A G2 JavaLink three-tier client can generate a *G2ConnectionClosed* event if the middle tier shuts down unexpectedly. By default, the client does not generate this event. To cause the event to be generated, when you start the middle tier by using *com.gensym.jgi.rmi.G2RMIAccessBrokerImpl*, provide the *-ping* command-line option followed by a poll interval, in seconds. Clients that connect through that middle tier now poll the middle tier at the specified interval and generate a connection closed event if the middle tier shuts down unexpectedly. If the *-ping* option is not specified, the appropriate Java exception is generated when any function is called across the middle tier.

In the following example, clients will poll the middle tier once a minute:

```
java com.gensym.jgi.rmi.G2RMIAccessBrokerImpl -tsName myserver -ping 60
```

Using an Applet to Connect to G2

You can create Java applets that use the JavaLink API to connect to G2, using Java RMI. The Java applet client and G2 must deliver the page contents. To ensure complete compatibility between Sun's implementation of Java and Netscape's implementation, you need to use the Java Plug-in from Sun. The Java Plug-in includes a free Java Plug-in HTML-converter that makes it easy to modify HTML pages to specify the use of the Java Plug-in, rather than using the browser's default Java runtime. The *readme-javalink.html* file lists compatibility versions between JDK and the Java Plug-in. The communication will be over RMI, not HTTP, so security restrictions apply.

Note At this time, due to Bug HQ-3186311, when using a Java applet with the Java Plug-In to connect to G2, you must use Java RMI; otherwise, the applet fails to reconnect to G2 when you go to another page and return to the applet. For an update on the status of this Bug, please query HelpLink or contact Gensym Customer Support.

8

Adding JavaBeans Events to G2 Classes

Describes how G2 JavaLink uses the JavaBeans event model with G2 classes.

Introduction

Most G2 JavaLink (JavaLink) clients require a G2 server to pass unsolicited information periodically to Java, for example, for variable updates and status changes. To implement this behavior, a JavaLink developer can register Java methods as callable by G2, as described in Remote Procedure Calls. While this method is easy to implement for small applications, it does not scale well as the application becomes more complex.

Therefore, JavaLink allows G2 developers to take advantage of the event-passing mechanism developed for Java Beans, where Java objects can register interest and listen for events generated by G2 objects.

The JavaBeans Event Model

The JavaBeans event model is based on the concept of an "event listener." An object interested in receiving events is an **event listener**, and an object that generates events is an **event source**.

Event Source

An event source maintains a list of listeners interested in being notified when events occur. An event source also provides methods that allow listeners to add and remove themselves from the list of interested objects. When the event source generates an event or when a user-input event occurs on the event source, the event source notifies all the registered event listeners that the event has occurred.

An event source notifies an event listener by invoking a method on the listener, passing an **event object** as the method's argument. The event object is an instance of a subclass of `java.util.EventObject`.

Event Listener

For an event source to invoke a method on an event listener, the listener must implement the required method. Java ensures the required method is implemented by requiring that all event listeners for a particular type of event implement the corresponding interface. For example, event listeners for `ActionEvent` objects must implement the `ActionListener` interface. All event listener interfaces themselves extend `java.util.EventListener`. This interface does not define any methods, but instead acts as a marker interface, clearly identifying all event listeners as such.

An event listener interface may define more than one method. For example, `G2ConnectionEvent` represents several different types of `G2Connection` event types, such as G2 pausing and the G2 connection being closed. These different event types cause different methods in the corresponding event listener to be invoked. By convention, the methods of an event listener are passed a single argument, which is an event object of the type that corresponds to the listener. This event object should contain all the information a program needs to respond to the event.

JavaLink uses the Java Bean event model to allow Java objects to listen for G2 connection events, as described in Connection Events. In the case of G2 connection events, `G2ConnectionListener` defines the listener interface and `G2ConnectionEvent` passes the event information to each of the event methods within the listener interface.

JavaBeans Event Model Naming Conventions

The JavaBeans specification defines strict guidelines of how to add a listener interface to a Java class:

- Any listener interface name must end in the word *Listener*, for example, *G2ConnectionListener*. The event listener's name is *G2Connection*.
- Any event objects passed to methods defined in a listener interface must end in the word *Event*, for example, *G2ConnectionEvent*.
- The Java class generating the events must supply add and remove listener interface methods for registering and removing registration for the listener, for example, *addG2ConnectionListener* and *removeG2ConnectionListener*.

The Add Listener Method

The event source must define a method with this signature for registering listeners to receive notification of events:

```
public void add<listener-name>Listener
    (<listener-name>Listener a-listener);
```

This method registers a listener with an event generator. Any registered listeners will be informed of a listener event when it occurs. The event source normally holds a reference to the registered listener.

For example, *G2Connection* defines the following method for registering *G2ConnectionListener*s:

```
public void addG2ConnectionListener
    (G2ConnectionListener a-listener);
```

The Remove Listener Method

The event source must also define a method with this signature for removing registration of listeners:

```
public void remove<listener-name>Listener
    (<listener-name>Listener a-listener);
```

This method informs the event source that a previously registered event listener should no longer receive any *<listener-name>* events. All references that the event source might have held on the listener are removed.

For example, *G2Connection* defines the following method for removing *G2ConnectionListener*s:

```
public void removeG2ConnectionListener
    (G2ConnectionListener a-listener);
```

Implementing the JavaBeans Event Model in G2

The G2 event support module (g2evliss), which you access by loading the *g2evliss.kb* that ships with JavaLink, provides basic support to allow a G2 class to implement one or more listener interfaces and corresponding event objects.

When JavaLink exports a G2 class that uses these event listener classes, JavaLink automatically generates the corresponding Java listener and event classes such that external Java classes can register themselves as interested in events fired from G2.

The G2 event support module provides the following classes:

G2 Class Name	Description
g2-event-listener	The superior of all G2 event listener classes.
g2-event-object	The superior of all G2 event classes.
g2-event-listener-support	Basic support for G2 classes that would like to implement a G2 event listener interface.

Adding a G2 Event Listener Interface to a G2 Class

To add a G2 event listener interface to a G2 class:

1. Create a G2 event listener class that inherits from g2-event-listener.

   The class name must have the form: *listener-name*-listener. This class should include methods for each event to be reported by the listener interface.

2. Create a G2 event class that inherits from g2-event-object.

   The class name must have the form: *listener-name*-event. This class should contain attributes that allow you to report event data for each possible event in the G2 event listener class created above.

3. Define the listener registration methods for the G2 class that will generate the events.

These must have the following signatures:

add-*listener-name*-listener
(self: class *event-source*, a-listener: class *listener-name*-listener) = ()

remove-*listener-name*-listener
(self: class *event-source*, a-listener: class *listener-name*-listener) = ()

Example

Suppose you want to add an event listener interface to a G2 class named company-stock. An instance of company-stock contains information about a particular company's stock and has a live feed to the stock-market so its information is constantly updated. By adding an event interface to the company-stock class, you can allow third-party systems to listen for live company-stock events.

For example, the class might generate these two events:

- stock-price-change, which informs listeners of a change in a company's stock price.
- stock-press-release, which informs listeners when a company has released a new press release.

To create this interface to the listener:

1. Decide on the listener interface name.

   For this example, the listener name is stock-update.

2. Create the listener class.

   Following the guidelines, the listener class must be called stock-update-listener and must inherit from g2-event-listener.

3. Create an event class that contains the information generated by each event.

   Following the guidelines, the event class must be called stock-update-event and must inherit from g2-event-object.

4. Define attributes on the event class to hold information about the event.

   For example, the stock-update-event might define the following attributes to hold stock information:

   message is a text, initially is "";
   price is a float, initially is 0.0

5 Add method declarations and methods to the listener class that define each possible event that might occur.

As dictated by the JavaBeans event model, the first argument to the method is self, which is an instance of the g2-event-listener class, and the second argument is the event class.

For example, the stock-update-listener might define these methods corresponding to events that might occur:

```
stock-price-change(self: class stock-update-listener,
    event: class stock-update-event) = ()
begin
end

stock-press-release(self: class stock-update-listener,
    event: class stock-update-event) = ()
begin
end
```

As you can see, these methods do not implement any code; listeners will override them to perform specific actions.

6 Finally, add listener registration methods to the event source and provide new code that informs any registered listeners when an event occurs.

The following two headings describe this last step in detail and provide examples.

Registration Methods

When adding the listener registration methods to company-stock, you can use the g2-event-listener-support class supplied with the G2 event support module KB (*g2evliss.kb*). This class defines generic add and remove methods that you can use to implement your registration methods.

First, you must add g2-event-listener-support to your event source's class inheritance path, then you can define your registration methods.

Note The registration method names must also follow the event listener naming conventions.

When a Java class adds itself as a listener to a G2 event source, G2 creates a listener stub for passing listener events from G2 to Java. Removing yourself as a listener does not delete the listener stub in G2; it simply removes the listener from the listener list in Java. Each time a new event is generated in G2, each connected listener uses the same listener stub that was created when it was added as a listener. If the G2 listener stub no longer exists and a new event is generated, JavaLink automatically creates a new listener stub in G2.

Normally, it is not necessary to delete the listener stub explicitly in G2, because JavaLink handles that for you by deleting the listener stub when a G2 event is

generated and the listener is no longer connected. However, if different clients are adding and removing themselves as listeners, G2 creates a new listener stub each time a new client adds itself as a listener, which can result in many listener stubs being created. To avoid excessive listeners being created in G2, the registration method that removes the listener can add a delete statement to delete the listener stub. However, due to the asynchronous nature of communications between G2 and JavaLink, repeatedly adding and removing listeners from a single client can cause an exception if the delete notification occurs after the add notification. To avoid this problem, insert a sleep statement after each remove statement that is followed by another add statement.

Here are the registration methods for the company-stock event source:

```
add-stock-update-listener(
    self: class company-stock,
    listener: class stock-update-listener) = ()
begin
    { call support method defined in g2-event-listener-support }
    call add-g2-event-listener(self, listener);
end

remove-stock-update-listener(
    self: class company-stock,
    listener: class stock-update-listener) = ()
begin
    { call support method defined in g2-event-listener-support }
    call remove-g2-event-listener(self, listener);
end
```

Event-Firing Code

Now you must provide code that hooks in to the company-stock's update mechanism so it can fire events. To do this, you write event-firing methods for each event that you have defined and call them from company-stock's update mechanism:

```
fire-stock-price-change-event
    (self: class company-stock, new-price: float) = ()
listener-list: class item-list;
listener: class stock-update-listener;
event: class stock-update-event;
error-sym: symbol;
error-text: text;
```

```
begin
    { use g2-event-support-listener method to get our listener list}
    listener-list = call get-event-listener-list(self);

    { use support method defined in g2-event-listener-support
    method to create an event}
    event = call g2-event-create-event(stock-update-event, self);
    conclude that the price of event = new-price;
    for listener = each stock-update-listener in listener-list do
        begin
            call stock-price-change(listener, event);
        end on error(error-sym, error-text)
        { obviously listener it is not behaving, so remove it from listener list}
        inform the operator that "problem with listener, removing..[error-text]";
        call remove-g2-event-listener(self, listener);
        end;
    end;
    delete event;
end

fire-stock-press-release-event
    (self: class company-stock, news: text) = ()
listener-list: class item-list;
listener: class stock-update-listener;
event: class stock-update-event;
error-sym: symbol;
error-text: text;

begin
    { use g2-event-support-listener method to get our listener list}
    listener-list = call get-event-listener-list(self);

    { use support method defined in g2-event-listener-support
     method to create an event}
    event = call g2-event-create-event(stock-update-event, self);
    conclude that the message of event = news;
    conclude that the price of event = the price of self;
    for listener = each stock-update-listener in listener-list do
        begin
            call stock-press-release(listener, event);
        end on error(error-sym, error-text)
        { obviously listener it is not behaving, so remove it from listener list}
        inform the operator that "problem with listener, removing..[error-text]";
        call remove-g2-event-listener(self, listener);
        end;
    end;
    delete event;
end
```

How JavaLink Exports G2 Event Listener Interfaces to Java

When JavaLink exports a G2 class to Java by using the G2 DownloadInterfaces utility as described in How to Export a G2 Class as a Java Class, it checks for any JavaBeans-compliant event interfaces. If it finds any, it exports the following classes and methods so that any Java class can listen for G2 events:

- For any G2 event listener interface found, a JavaBeans-compliant event listener interface named *G2_<listener-name>Listener* is created, which correctly implements *java.util.EventListener*.

  For example, the G2 event listener class for stock-update-listener would be *G2_StockUpdateListener*. This interface contains the signatures for each event method found in the G2 event listener interface.

  For example, the following interface would be exported for stock-update-listener:

```
public interface G2_StockUpdateListener extends
   java.util.EventListener {

public void stockPriceChange(G2_StockUpdateEvent event);
public void stockPressRelease(G2_StockUpdateEvent event);
}
```

- JavaBeans-compliant event classes required for the listener interfaces named *G2_<listener-name>Event* are also created, which correctly extend *com.gensym.classes.G2_ExternalEventObjectRoot*. For example, g2-stock-update-event is exported as *G2_StockUpdateEvent*. This exported event class contains getters for any attributes defined by the G2 class:

```
public class G2_StockUpdateEvent
   extends com.gensym.classes.G2_ExternalEventObjectRoot {
// ....

public String getMessage() {...};
public double getPrice() {...};

}
```

Using G2 Event Listener Interfaces From Java

When a Java class wishes to listen for G2 events, it simply needs to implement the interfaces exported by JavaLink and register itself as a listener on the G2 object by calling the object's add listener method from Java.

For the stock example, if you export company-stock to Java, JavaLink automatically create these Java classes:

Java Class	Description
CompanyStock	The Java interface for the G2 class company-stock.
StockUpdateListener	The Java interface for the G2 class stock-update-listener.
G2_StockUpdateListener	The JavaBeans-compliant event listener interface for stock-update-listener.
StockUpdateEvent	The Java interface for the G2 class stock-update-event.
G2_StockUpdateEvent	The JavaBeans-compliant event class for stock-update-event.

To listen for G2 events in Java:

1. The Java class should implement the G2 event listener interface, for example, *StockUpdateListener*.
2. Override the listener's methods, for example, *stockPriceChange* and *stockPressRelease*.

The following example shows how a Java class can listen for *StockUpdateEvent*s fired from a company-stock object in a connected G2:

```
public class Trader implements G2_StockUpdateListener{

    // Constructor
    public Trader(CompanyStock stock) {
        try {
            stock.addG2_StockUpdateListener(this);
        } catch (G2AccessException e) {
            // Handle problem with G2 object
        }
    }

    public void stockPriceChange(G2_StockUpdateEvent arg1)
        throws G2AccessException
    {
        // Sell or buy
    }

    public stockPressRelease(G2_StockUpdateEvent arg1)
        throws G2AccessException
    {
        // Panic or drink champagne
    }

}
```

Index

Symbols

A

B

C

D

H

I

J

L

M

N

O

P

R

S

T

U

V

W