GT.M

Technical Bulletin

GTMJI

Empowering
the Financial World

FIS

Contact Information

GT.M Group
Fidelity National Information Services, Inc.
200 Campus Drive
Collegeville, PA 19426
United States of America

GT.M Support for customers: gtmsupport@fisglobal.com
Automated attendant for 24 hour support: +1 (484) 302-3248
Switchboard: +1 (484) 302-3160
Website: http://fis-gtm.com

Legal Notice

Revision History

Revision 1.1	03 June 2019	Remove information related to the Sun SPARC Solaris platform. Update the list of Java platforms on which FIS GT.M tests the GTMJI plugin and specify that GTMJI is a source code only distributions and requires GNU Make, a C compiler, and a JDK environment for installation.
Revision 1.0	03 May 2013	First published version.

Table of Contents

GTMJI - GT.M Java Interface Plug-In 1
- Overview 1
- Installation 1

GT.M call-ins usage from Java 5
- Environment Configuration 5
- Invocations 5
- Types 6
- Return Types 7
- Input vs. Input/Output Arguments 7
- String Conversion 8
- Exceptions 8
- Multi-threading 8
- Examples 8

GT.M call-outs usage with Java 13
- Environment Configuration 13
- Invocations 13
- Types 14
- Return Types 15
- String Conversion 15
- Exceptions 15
- Examples 15

Additional Considerations 19
- Performance 19
- Numeric Conversions 19

GTMJI - GT.M Java Interface Plug-In

Overview

GTMJI provides a mechanism to call-in to GTM from Java application code, and to call out from GT.M to Java application code. GTMJI requires a minimum GT.M release of V6.0-002 and is supported on Linux on x86 and x86_64, and AIX. The following table lists the platforms and Java distributions on which FIS tested the GTMJI plug-in:

Platforms	Java
IBM System p AIX	IBM JDK 1.7
x86_64 GNU/Linux	OpenJDK 1.7 and 1.8
x86 GNU/Linux	OpenJDK 1.7 and 1.8

Although GTMJI may as well work on other combinations of platforms and Java implementations, the above are the platforms on which GTMJI is tested. Versions of each platform are those on which your GT.M is supported per the release notes for that release.

Download Examples

gtmji-demo.zip contains examples of Java programs, call-in/call-out tables, and GT.M APIs described in this technical bulletin. To download **gtmji-demo.zip**, click  or open directly from http://tinco.pair.com/bhaskar/gtm/doc/articles/gtmji-demo.zip. For instruction on running call-in examples, see the code comments of CI.java or JPiece.java. For instructions on running the call-out example, see the code comments of CO.java.

Installation

GTMJI is a source code distribution. GTMJI comes with a Makefile that you can use with GNU Make to build, test, install, and uninstall the package. You need to have GNU Make, C compiler, and a JDK environment (for the JNI [https://en.wikipedia.org/wiki/Java_Native_Interface components] components). On some platforms, GNU make may be accessed with the command gmake, not make. You can build and test GTMJI as a normal (non-root) user, and then as root install it as a GT.M plug-in. The targets in Makefile that are intended for external use are:

* **all**: creates libgtmj2m.so and libgtmm2j.so (shared libraries of C code that acts as a gateway for Java call-ins and call-outs, respectively) and gtmji.jar (a Java archive containing GTMJI type wrapper and thread management classes).

* **clean**: deletes all files created as a result of running the test and/or all target.

* **install**: executed as root, installs GTMJI as a plug-in under the GT.M installation directory.

* **install-test**: executed as root, ensures the operation of GTMJI after building and installation. Messages "GTMJI-INSTALL-SUCCESS: Call-ins test succeeded." and "GTMJI-INSTALL-SUCCESS: Call-outs test succeeded." confirm successful installation. If you are using UTF-8, make sure to set the environment; two additional GTMJI-INSTALL-SUCCESS messages should be printed.

* **test**: ensures the operation of GTMJI after building and before installation. As with the install-test target, "GTMJI-INSTALL-SUCCESS: Call-ins test succeeded." and "GTMJI-INSTALL-SUCCESS: Call-outs test succeeded." messages should appear. If you are using UTF-8, make sure to set the environment; two additional GTMJI-INSTALL-SUCCESS messages should be printed.

* **uninstall**: executed as root, removes the installed plug-in from under the GT.M installation directory.

The following targets also exist but are intended for use within the Makefile rather than for external invocation: libgtmj2m.so, libgtmm2j.so, gtmji.jar, $(PLUGINDIR)/libgtmj2m.so, $(PLUGINDIR)/libgtmm2j.so, $(PLUGINDIR)/gtmji.jar, $(UTFPLUGINDIR)/libgtmj2m.so, $(UTFPLUGINDIR)/libgtmm2j.so, and $(UTFPLUGINDIR)/gtmji.jar.

To run the Makefile, set the following environment variables:

* **gtm_dist**: Installation directory of GT.M that contains libgtmshr.so and such include files as gtm_common_defs.h and gtmxc_types.h. If you plan to install GTMJI for multiple GT.M versions, please clean the build each time, since both gtmxc_types.h and gtm_common_defs.h are included from $gtm_dist to build the shared library.

* **JAVA_HOME**: Top directory of your Java installation, such as /usr/lib/jvm/jdk1.6.0_25.

* **JAVA_SO_HOME**: Directory that contains libjava.so; typically,

  * AIX: $JAVA_HOME/jre/lib/ppc64

  * Linux: $JAVA_HOME/jre/lib/amd64, $JAVA_HOME/jre/lib/i386, or $JAVA_HOME/jre/lib/i686

* **JVM_SO_HOME**: Directory that contains libjvm.so; typically,

  * AIX: $JAVA_HOME/jre/lib/ppc64/j9vm

  * Linux: $JAVA_HOME/jre/lib/amd64/server, $JAVA_HOME/jre/lib/i386/server, or $JAVA_HOME/jre/lib/i686/server

Note that if you also have a GT.M installation with UTF-8 support (that is, GT.M executables under $gtm_dist/utf8), then you might need to configure a few additional environment variables for targets that run GTMJI tests, such as test and install-test. Setting gtm_icu_version suffices in most situations.

The steps for a typical GTMJI installation are as follows:

1. Set gtm_dist, JAVA_HOME, JAVA_SO_HOME, and JVM_SO_HOME environment variables.

2. Run make all.

3. Run make test. When make test completes, two (four if you have a UTF-8-enabled installation) "GTMJI-INSTALL-SUCCESS ..." messages get displayed.

4. Run make install.

5. Run make install-test. When make install-test completes, two (four if you have a UTF-8-enabled installation) "GTMJI-INSTALL-SUCCESS ..." messages get displayed.

6. Run make clean to remove all temporary GTMJI files created in the current directory by other make commands.

GT.M call-ins usage from Java

Environment Configuration

To call-in to GT.M application code from Java application code requires the gtmji.jar Java archive together with the libgtmj2m.so shared library. The JAR contains the definitions of special types that the call-in functions may use for arguments; it also loads the shared library, performs concurrency control, and sets up proper rundown logic on Java process termination. In addition to the usual GT.M environment variables, such as gtmroutines and gtmgbldir, the following environment variables need to be defined:

* **GTMCI**: The location of the call-in table, as described in the GT.M Programmers Guide.
* **LD_LIBRARY_PATH** (**LIBPATH** on AIX): Includes the location of libgtmj2m.so shared library, such as /usr/lib/fis-gtm/V6.0-002/plugin. Alternatively, pass the location of the shared library to the JVM via java.library.path property.

After setting these environment variables, the invocation of a Java process might look like

```
java -classpath "/path/to/classes/top/dir/:/path/to/gtmji.jar" com.callins.Test
```

or

```
java -Djava.library.path=/path/to/libgtmj2m/dir/ -classpath "/path/to/classes/top/dir/:/path/
to/gtmji.jar" com.callins.Test
```

in case you do not set LD_LIBRARY_PATH. It is also possible to define CLASSPATH environment variable instead of specifying the -classpath option.

Invocations

GTMJI provides the following methods for invoking M routines, each for a specific return type:

```
public static native void doVoidJob(String routine, Object... args);
public static native int doIntJob(String routine, Object... args);
public static native long doLongJob(String routine, Object... args);
public static native float doFloatJob(String routine, Object... args);
public static native double doDoubleJob(String routine, Object... args);
public static native String doStringJob(String routine, Object... args);
public static native byte[] doByteArrayJob(String routine, Object... args);
```

These functions are defined in the GTMCI class; so, after importing gtmji.jar, the call-in invocations have the following format:

```
GTMCI.doXXXJob(routineName, arg1, arg2, ...);
```

In many ways, these functions are similar to gtm_ci() and gtm_cip() calls from C. Note, however, that unlike with C, you do not have to call the initialization and rundown functions explicitly from Java because GTMJI handles these operations automatically. Another key difference from C concerns Java's

inability to modify primitive types by reference. To address this shortcoming, GTMJI provides the following wrapper classes to the five respective primitives:

* **GTMBoolean**
* **GTMInteger**
* **GTMLong**
* **GTMFloat**
* **GTMDouble**

Although Strings are object types in Java, they are immutable and do not require a direct call to a constructor at instantiation. As a result, GTMJI provides String-type arguments with call-in invocations, but such calls do not modify the argument's content even if that argument is declared for input-output use in the mapping table (see below). To pass strings that can be modified, the plug-in implements the GTMString wrapper. Similarly, although GT.M application code can modify the contents of a byte array argument, it cannot expand or reduce its size. Modifying a byte[] argument passed from Java does not alter the array's original capacity; furthermore, if the modifications target fewer elements than the array contains, the unmodified elements retain their values. In contrast, GTMByteArray arguments expand when a newly assigned value exceeds the original capacity of the array; still, if a newly assigned value is smaller than the original size, the unmodified bytes retain their original contents.

Types

GTMJI provides the following types for calling in to GT.M from Java:

* **GTMBoolean**
* **GTMInteger**
* **GTMLong**
* **GTMFloat**
* **GTMDouble**
* **GTMString**
* **GTMByteArray**
* **String**
* **byte[]**
* **BigDecimal**

Each GTM-prefixed class contains a 'value' field of the corresponding primitive or object (in case of GTMString) type. For simplicity and performance reasons, the field is public, and there are no mutator

or accessor methods to access it. Nevertheless, you can print the actual values without explicitly dereferencing the field because the GTM-XXX classes implement the toString() method.

Because Java always passes objects by reference, GTMJI only allows a call-in function to modify its arguments when they are marked as input-output or output-only (as opposed to input-only) in the call-in table. From a programmer's point of view, passing input-only and input-output arguments is identical. Once again, bear in mind that it is *not* possible to modify Strings even if you mark them for input-output use; if you need to modify Strings, use GTMStrings. The below table summarizes the proper usage of all types allowed with GT.M call-ins from Java:

Type in Java	Type in CI table	Use
GTMBoolean	gtm_jboolean_t	I, IO, O
GTMInteger	gtm_jint_t	I, IO, O
GTMLong	gtm_jlong_t	I, IO, O
GTMFloat	gtm_jfloat_t	I, IO, O
GTMDouble	gtm_jdouble_t	I, IO, O
GTMString	gtm_jstring_t	I, IO, O
GTMByteArray	gtm_jbyte_array_t	I, IO, O
String	gtm_jstring_t	I
byte[]	gtm_jbyte_array_t	I, IO, O
BigDecimal	gtm_jbig_decimal_t	I

Notice the special gtm_jXXX_t mapping types defined for usage with Java; be sure to employ them instead of the gtm_XXX_t and gtm_XXX_t * types designated for usage with C. To utilize an argument in input or output direction it suffices to correctly label it as 'I', 'O', or 'IO'; do not include '*' as the Java programming language does not have any close analog to the pointer types in C/C++.

Do not pass a variable of any other type than described above to a call-in function from Java. The total number of arguments should not exceed 32.

Return Types

Whenever the call-in routine is expected to return a value, use the corresponding GTMCI function. Unlike with C, do not insert additional parameter to retrieve the return value. Specify one of the available return types—void, int, long, float, double, String, or byte[]—in the call-in table.

Input vs. Input/Output Arguments

All input/output call-in arguments are passed to GT.M by reference. The ZWRITE and ZSHOW output formats of such arguments appear to have an association ("; *") with a non-existing alias variable, but under most conditions they behave just like local variables. One exception is the difference in operation

of KILL and KILL * commands. While KILL * only removes input/output arguments, KILL removes all arguments regardless of type. Arguments removed by either KILL or KILL * become undefined in GT.M, yet any modifications to the arguments made prior to KILL * reflect on the Java side, whereas KILL clears all GT.M modifications.

String Conversion

GTMJI converts all String and GTMString arguments from the UTF-16 encoding used by Java to UTF-8 before passing them to GT.M. Additionally, in case of input-output or output-only GTMStrings, the plug-in converts the UTF-8 values back to UTF-16 during transfer from M to Java. Unless you are assured that the application only deals with ASCII characters ($Char(0) through $Char(127)), GT.M must run in UTF-8 mode. Use byte[] and GTMByteArray arguments for data such as binary values, that you want to pass unmodified between Java and GT.M.

Exceptions

Whenever GT.M detects an error condition with a call-in—whether in the invocation processing or the M program execution—it returns control to Java, which throws an exception with a descriptive error message . It is the responsibility of Java application code to catch and handle the exception.

Multi-threading

Java applications normally expect to run multi-threaded, but the GT.M runtime system is single-threaded. GTMJI therefore ensures that only one thread executes GT.M code at any given time; any additional thread that attempts to enter the GT.M runtime system is blocked until the first thread returns to the control of the Java runtime system.

To avoid unpredictable results, Java application code should always protect the arguments used when calling into GT.M from unsynchronized reads and writes to prevent thread synchronization issues. If an M routine modifies one of its arguments inside a call-in while some other thread tries to read it, or if M reads an argument which might be modified in a different thread, application code needs to ensure correct synchronization between both read and write threads.

As FIS reserves the right to make GT.M runtime system multi-threaded at a future date, you should ensure that your application code does not rely on the single-threadedness of the GT.M runtime system. Also, while M local variables are shared by all Java threads that call into M, this behavior may or may not continue if and when FIS makes the GT.M runtime system multi-threaded in the future.

Examples

Consider the following example. The call-in table contains two entries:

```
f1:gtm_jdouble_t
 ci1^rtns(I:gtm_jboolean_t,I:gtm_jstring_t,I:gtm_jbyte_array_t,I:gtm_jbig_decimal_t)
f2:void
 ci2^rtns(IO:gtm_jint_t,IO:gtm_jlong_t,IO:gtm_jfloat_t,IO:gtm_jstring_t,IO:gtm_jbyte_array_t)
```

The first entry exposes the 'ci1' label from the 'rtns' routine to Java under the name 'f1'; the second entry exposes the 'ci2' label from the 'rtns' routine to Java under the name 'f2'. The first function returns a double value and expects four arguments of types GTMBoolean, String or GTMString, byte[] or GTMByteArray, and BigDecimal, respectively. The second function does not return a value and expects five arguments of types GTMInteger, GTMLong, GTMFloat, String or GTMString, and byte[] or GTMByteArray, respectively.

The 'rtns' routine contains the following:

```
ci1(bi,jsi,gbai,bdi)
  write "Inside c1:",!
  write "bi: "_bi,!,"jsi: "_jsi,!,"gbai: "_gbai,!,"bdi: "_bdi,!
  write "----------------------",!
  set bi=1,jsi=2,gbai=3,bdi=4
  quit 5.678

ci2(iio,lio,fio,gsio,jbaio)
  write "Inside c2:",!
  write "iio: "_iio,!,"lio: "_lio,!,"fio: "_fio,!,"gsio: "_gsio,!,"jbaio: "_jbaio,!
  write "----------------------",!
  set iio=123,lio=234,fio=345,gsio="567",jbaio=$char(54)_$char(55)_$char(56)
  quit
```

Finally, the Java class that invokes the call-in looks like

```
package com.fis.test;

import java.math.BigDecimal;

import com.fis.gtm.ji.GTMBoolean;
import com.fis.gtm.ji.GTMByteArray;
import com.fis.gtm.ji.GTMCI;
import com.fis.gtm.ji.GTMDouble;
import com.fis.gtm.ji.GTMFloat;
import com.fis.gtm.ji.GTMInteger;
import com.fis.gtm.ji.GTMLong;
import com.fis.gtm.ji.GTMString;

public class CI {

  public static void main(String[] args) {
    try {
      boolean booleanValue = false;
      int intValue = 3;
      long longValue = 3141L;
      float floatValue = 3.141f;
      double doubleValue = 3.1415926535;
      String gtmStringValue = "GT.M String Value";
      String javaStringValue = "Java String Value";
      String gtmByteArrayValue = new String(new byte[]{51, 49, 52, 49});
      String javaByteArrayValue = new String(new byte[]{51, 49, 52, 50});
      String bigDecimalValue = "3.14159265358979323846";
```

```
GTMBoolean gtmBoolean = new GTMBoolean(booleanValue);
GTMInteger gtmInteger = new GTMInteger(intValue);
GTMLong gtmLong = new GTMLong(longValue);
GTMFloat gtmFloat = new GTMFloat(floatValue);
GTMDouble gtmDouble = new GTMDouble(doubleValue);
GTMString gtmString = new GTMString(gtmStringValue);
GTMByteArray gtmByteArray = new GTMByteArray(gtmByteArrayValue.getBytes());
String javaString = javaStringValue;
byte[] javaByteArray = javaByteArrayValue.getBytes();
BigDecimal bigDecimal = new BigDecimal(bigDecimalValue);

System.out.println(
  "Before f1 and f2:\n" +
  "gtmBoolean: " + gtmBoolean + "\n" +
  "gtmInteger: " + gtmInteger + "\n" +
  "gtmLong: " + gtmLong + "\n" +
  "gtmFloat: " + gtmFloat + "\n" +
  "gtmDouble: " + gtmDouble + "\n" +
  "gtmString: " + gtmString + "\n" +
  "gtmByteArray: " + gtmByteArray + "\n" +
  "javaString: " + javaString + "\n" +
  "javaByteArray: " + new String(javaByteArray) + "\n" +
  "bigDecimal: " + bigDecimal + "\n" +
  "----------------------");

double doubleReturnValue = GTMCI.doDoubleJob(
  "f1",
  gtmBoolean,
  javaString,
  gtmByteArray,
  bigDecimal);

Thread.sleep(1000); // for synchronization of the output

System.out.println(
  "After f1 but before f2:\n" +
  "gtmBoolean: " + gtmBoolean + "\n" +
  "gtmInteger: " + gtmInteger + "\n" +
  "gtmLong: " + gtmLong + "\n" +
  "gtmFloat: " + gtmFloat + "\n" +
  "gtmDouble: " + gtmDouble + "\n" +
  "gtmString: " + gtmString + "\n" +
  "gtmByteArray: " + gtmByteArray + "\n" +
  "javaString: " + javaString + "\n" +
  "javaByteArray: " + new String(javaByteArray) + "\n" +
  "bigDecimal: " + bigDecimal + "\n" +
  "doubleReturnValue: " + doubleReturnValue + "\n" +
  "----------------------");

GTMCI.doVoidJob(
```

```
        "f2",
        gtmInteger,
        gtmLong,
        gtmFloat,
        gtmString,
        javaByteArray);

      Thread.sleep(1000); // for synchronization of the output

      System.out.println(
        "After f1 and f2:\n" +
        "gtmBoolean: " + gtmBoolean + "\n" +
        "gtmInteger: " + gtmInteger + "\n" +
        "gtmLong: " + gtmLong + "\n" +
        "gtmFloat: " + gtmFloat + "\n" +
        "gtmDouble: " + gtmDouble + "\n" +
        "gtmString: " + gtmString + "\n" +
        "gtmByteArray: " + gtmByteArray + "\n" +
        "javaString: " + javaString + "\n" +
        "javaByteArray: " + new String(javaByteArray) + "\n" +
        "bigDecimal: " + bigDecimal + "\n");
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

If correctly configured, the execution of the above class produces the following output:

```
Before f1 and f2:
gtmBoolean: false
gtmInteger: 3
gtmLong: 3141
gtmFloat: 3.141
gtmDouble: 3.1415926535
gtmString: GT.M String Value
gtmByteArray: 3141
javaString: Java String Value
javaByteArray: 3142
bigDecimal: 3.14159265358979323846
----------------------
Inside c1:
bi: 0
jsi: Java String Value
gbai: 3141
bdi: 3.14159265358979323846
----------------------
After f1 but before f2:
gtmBoolean: false
gtmInteger: 3
gtmLong: 3141
gtmFloat: 3.141
```

```
gtmDouble: 3.1415926535
gtmString: GT.M String Value
gtmByteArray: 3141
javaString: Java String Value
javaByteArray: 3142
bigDecimal: 3.14159265358979323846
doubleReturnValue: 5.678
----------------------
Inside c2:
iio: 3
lio: 3141
fio: 3.141
gsio: GT.M String Value
jbaio: 3142
----------------------
After f1 and f2:
gtmBoolean: false
gtmInteger: 123
gtmLong: 234
gtmFloat: 345.0
gtmDouble: 3.1415926535
gtmString: 567
gtmByteArray: 3141
javaString: Java String Value
javaByteArray: 6782
bigDecimal: 3.14159265358979323846
```

Notice that although there is an attempt to modify the arguments inside the 'ci1' function, they remain unchanged after the GTMCI.doDoubleJob("f1", ...) call. However, the values change in 'ci2', as seen after the GTMCI.doVoidJob("f2", ...) call. Note that the fourth byte of the javaByteArray argument preserved its value (because the call only changed three bytes).

GT.M call-outs usage with Java

Environment Configuration

To invoke Java programs from M code via the GT.M call-out interface, the first line of the external call table should point to the libgtmm2j.so shared library. Java code must include the gtmji.jar Java archive to provide the special types for argument passing. Besides gtmroutines and gtmgbldir, define the following environment variables:

* **GTMXC_XXX**: Includes the location of the call-out table definition, where XXX is the name of the external package as it is referenced in an M routine.
* **GTMXC_classpath**: Includes the top directory of the class file, or the JAR file with it, as well as the path to gtmji.jar. Please note that JAR paths should include the file name in the path.
* **LD_LIBRARY_PATH** (**LIBPATH** on AIX): Points to the locations of libgtmm2j.so, libjvm.so, and libjava.so shared libraries. (Use ':' as a delimeter.)
* **LD_PRELOAD** (**LDR_PRELOAD64** on AIX): Points to the location of libjsig.so shared library (required for signal chaining).

By default GTMJI starts a JVM with the following flags:

* **-Xrs**: Reduces signal usage by the virtual machine.
* **-Dgtm.callouts=1**: Indicates Java call-out context for the GTMCI class within gtmji.jar in case of embedded Java call-in.
* **-Djava.class.path=<GTMXC_classpath environment variable>**: Sets the location of Java classes referenced by the program.

To specify additional JVM options (up to 50 total), initialize GTMXC_jvm_options environment variable, using ' ' or '|' as delimiter. A sample GTMXC_jvm_options string might look like

```
-Dspecial.value1=123 -Dspecial.value2=456 -Dspecial.value3=789
```

Duplicates of the default flags (-Xrs and -Djava.class.path) and unrecognized JVM options are omitted.

Invocations

Invocation of Java classes from M is similar to that of C shared libraries, except that the first parameter specifies the full class package path (with '/' instead of '.' as package delimiter), and the second– the method name. Note that GTMJI only supports call-outs to static Java methods) For example, to invoke a static long method doSomeWork from class WorkFactory in package com.abc.factory, use the following M code:

```
set result=$&work.doSomeWork("com/abc/factoryWorkFactory","doSomeWork",firstParm,.secondParm)
```

The corresponding call-out table definition (named work.co) might look like this:

```
/usr/lib/fis-gtm/V6.0-002_x86_64/plugin/libgtmm2j.so
doSomeWork: gtm_jlong_t gtm_xcj(I:gtm_jboolean_t,IO:gtm_jdouble_t
```

Notice that the name of the method that doSomeWork maps to is gtm_xcj; always use the gtm_xcj method mapping in the call-out table when you work with Java classes. Do not list the class package and the method name in the call-out entry. The Java method referred to in the above example might be implemented as follows:

```
public static long doSomeWork(Object[] args)
{
  GTMBoolean b = (GTMBoolean)args[0];
  GTMDouble d = (GTMDouble)args[1];
  // do some work here

  d.value = 5.358979; // this arg is IO and will be propagated back to M
  return 123;
}
```

As with doSomeWork, the invoked Java method's only argument should be of Object[] type; individual arguments need to be cast to the expected types, as defined in the call-out table.

Types

With the exception of BigDecimal, the types that GT.M provides for calling out to Java are the same as for calling in from Java: GTMBoolean, GTMInteger, GTMLong, GTMFloat, GTMDouble, GTMString, String, and byte[]. When passing an input-output or output-only argument from M, a '.' in front of the name indicates passing by reference, per standard M syntax. String or byte-array arguments are passed to Java as native types (that is, String and byte[]) when used as input-only, and passed as GT.M wrappers (GTMString and GTMByteArray, accordingly) when the direction is input-output or output-only; apply appropriate casting in the target Java method. The following table summarizes the type usage for GT.M call-outs to Java:

Type in Java	Type in CO table	Use
GTMBoolean	gtm_jboolean_t	I, IO, O
GTMInteger	gtm_jint_t	I, IO, O
GTMLong	gtm_jlong_t	I, IO, O
GTMFloat	gtm_jfloat_t	I, IO, O
GTMDouble	gtm_jdouble_t	I, IO, O
GTMString	gtm_jstring_t	IO, O
GTMByteArray	gtm_jbyte_array_t	IO, O
String	gtm_jstring_t	I
byte[]	gtm_jbyte_array_t	I

The call-out interface limits the total number of arguments to 29.

Return Types

Analogous to call-outs to C, the only allowed return types from Java are void (no value returned), gtm_status_t (int is returned), and gtm_jlong_t (long is returned). Returning a non-zero value with gtm_status_t return type results in an error. For more details, refer to the 'External Calls' chapter in the GT.M Programmer's Guide.

String Conversion

The conversion between UTF-8 (used in M) and UTF-16 (used in Java) encodings described in the corresponding section of 'Call-Ins Usage' chapter applies the same way to call-outs with appropriate ordering (opposite from call-in).

Exceptions

Whenever the call out encounters an unhandled error condition—whether in the invocation processing or the Java program execution—the control returns to GT.M, which raises an error. It is the application's responsibility to handle the error.

Examples

Consider the following example. The call-out table contains two entries:

```
/usr/lib/fis-gtm/V6.0-002_x86_64/plugin/libgtmm2j.so
f1:void
 gtm_xcj(I:gtm_jint_t,I:gtm_jlong_t,I:gtm_jdouble_t,I:gtm_jstring_t,I:gtm_jbyte_array_t)
f2:gtm_jlong_t
 gtm_xcj(IO:gtm_jboolean_t,IO:gtm_jfloat_t,IO:gtm_jstring_t,IO:gtm_jbyte_array_t)
```

The first entry defines the 'f1' label, and the second entry—the 'f2' label, and "assigns" them to the package whose name is encoded in the GTMXC_<package_name> environment variable. In this example we assume it to be 'test'. The first function does not return a value and expects five arguments whose values would resolve to int, long, double, String, and byte[], respectively. The second function returns a long value and expects four arguments that would resolve to boolean, float, String, and byte[], respectively.

The Java class (which we assume to be com.fis.test.CO) contains the following:

```
package com.fis.test;

import com.fis.gtm.ji.GTMBoolean;
import com.fis.gtm.ji.GTMByteArray;
import com.fis.gtm.ji.GTMCI;
import com.fis.gtm.ji.GTMDouble;
import com.fis.gtm.ji.GTMFloat;
```

```
import com.fis.gtm.ji.GTMInteger;
import com.fis.gtm.ji.GTMLong;
import com.fis.gtm.ji.GTMString;

public class CO {
  public static void co1(Object[] args) {
    GTMInteger gtmInteger = (GTMInteger)args[0];
    GTMLong gtmLong = (GTMLong)args[1];
    GTMDouble gtmDouble = (GTMDouble)args[2];
    String javaString = (String)args[3];
    byte[] javaByteArray = (byte[])args[4];

    System.out.println(
      "In co1():\n" +
      "gtmInteger: " + gtmInteger + "\n" +
      "gtmLong: " + gtmLong + "\n" +
      "gtmDouble: " + gtmDouble + "\n" +
      "javaString: " + javaString + "\n" +
      "javaByteArray: " + new String(javaByteArray) + "\n" +
      "----------------------");

    gtmInteger.value = 3;
    gtmLong.value = 141;
    gtmDouble.value = 5.926;
    javaString = "5358979";
    javaByteArray[0] = (byte)51;
  }

  public static long co2(Object[] args) {
    GTMBoolean gtmBoolean = (GTMBoolean)args[0];
    GTMFloat gtmFloat = (GTMFloat)args[1];
    GTMString gtmString = (GTMString)args[2];
    GTMByteArray gtmByteArray = (GTMByteArray)args[3];

    System.out.println(
      "In xcallIO():\n" +
      "gtmBoolean: " + gtmBoolean + "\n" +
      "gtmFloat: " + gtmFloat + "\n" +
      "gtmString: " + gtmString + "\n" +
      "gtmByteArray: " + gtmByteArray + "\n" +
      "----------------------");

    gtmBoolean.value = true;
    gtmFloat.value = 3.141f;
    gtmString.value = "592653";
    gtmByteArray.value = new byte[]{53, 56, 57};

    return 321;
  }
}
```

Finally, the M routine (which we assume to be 'rtns') looks like

```
co
  set b=0,i=123,l=456,f=789,d=1011,s="1213",ba="1415"
  set class="com/fis/test/CO"
  write "Before f1 and f2:",!
  write "b: "_b,!,"i: "_i,!,"l: "_l,!,"f: "_f,!,"d: "_d,!,"s: "_s,!,"ba: "_ba,!
  write "----------------------",!
  do &test.f1(class,"co1",i,l,d,s,ba)
  hang 1
  write "After f1 but before f2:",!
  write "b: "_b,!,"i: "_i,!,"l: "_l,!,"f: "_f,!,"d: "_d,!,"s: "_s,!,"ba: "_ba,!
  write "----------------------",!
  set ret=$&test.f2(class,"co2",.b,.f,.s,.ba)
  hang 1
  write "After f1 and f2:",!
  write "b: "_b,!,"i: "_i,!,"l: "_l,!,"f: "_f,!,"d: "_d,!,"s: "_s,!,"ba: "_ba,!
  quit
```

If correctly configured, the execution of the above routine should yield this output:

```
Before f1 and f2:
b: 0
i: 123
l: 456
f: 789
d: 1011
s: 1213
ba: 1415
----------------------
In co1():
gtmInteger: 123
gtmLong: 456
gtmDouble: 1011.0
javaString: 1213
javaByteArray: 1415
----------------------
After f1 but before f2:
b: 0
i: 123
l: 456
f: 789
d: 1011
s: 1213
ba: 1415
----------------------
In xcallIO():
gtmBoolean: false
gtmFloat: 789.0
gtmString: 1213
gtmByteArray: 1415
----------------------
After f1 and f2:
b: 1
```

```
i: 123
l: 456
f: 3.141
d: 1011
s: 592653
ba: 589
```

Similarly to the call-ins example, notice that while we attempt to modify the arguments inside the 'f1' function, they retain their original values after the &test.f1(class,"co1",...) call. The values do change, though, in 'f2', as seen after the $&test.f2(class,"co2",...) call. It is also important that the string and byte-array arguments passed from M in the first function are cast to String and byte[], whereas the second function casts them to GTMString and GTMByteArray, accordingly.

Additional Considerations

Performance

In addition to the tasks that JVM performs automatically on behalf of the user, GTMJI introduces overheads associated with argument processing, concurrency control, job scheduling, and resource allocation for classes and methods, and field bindings. In case of call-outs, the first call to Java starts the JVM, which may also cause a corresponding delay. For performance, FIS recommends making most Java invocations in the scope of one process lifetime, and to do as much work as possible in the context of one invocation.

It is also important to carefully choose the arguments on which both M and Java operate. Unnecessarily resorting to complex types and types with higher capacity adversely affects performance. This is especially true about byte arrays and strings, which undergo special allocation and conversion procedures.

Numeric Conversions

GT.M has only two data types - canonical numbers and strings. A string call-in argument cannot exceed 1 MiB. If a numeric argument exceeds 18 significant digits (GT.M internal limit for canonical numbers), GT.M retains the most significant 18 digits before returning to Java. For performance reasons, GTMJI does not check for values of numeric arguments that fall outside the limits of the primitive Java type. It is the application's responsibility to enforce the following limits for numeric arguments for correct computation:

GTMJI Type	**GT.M->Java**		**Java->GT.M**	
	Precision	**Range - [Max, Min]**	**Precision**	**Range - [Max, Min]**
GTMString	N/A	["", 1MiB]	N/A	["", 1MiB]
GTMInteger	Full	$[-2^{31}+1, 2^{31}-1]$	Full	$[-2^{31}, 2^{31}-1]$
GTMLong	18 digits	$[-2^{63}+1, 2^{63}-1]$	18 digits	$[-2^{63}, 2^{63}-1]$
GTMFloat	6-9 digits	[1E-43, 3.4028235E38]	6 digits	[1E-43, 3.4028235E38]
GTMDouble	15-17 digits	[1E-43, 1E47]	15 digits	[1E-43, 1E47]
Precision indicates the number of decimal digits retained when passing arguments within the corresponding numeric range.				
For floating-point types (GTMFloat and GTMDouble), Range denotes absolute values.				

Of the numeric types, it is generally safe to pass GTMInteger, GTMLong, and GTMFloat arguments from Java to M as they do not exceed GT.M's numeric range; GTMDouble and BigDecimal, however,

may cause numeric overflow if you use absolute values that are too big. On the M side, to avoid overflow issues, be sure not to assign values that the corresponding Java types cannot support; for instance, do not assign a value exceeding (2^32 - 1) to an argument which is defined as gtm_jint_t in the mapping table and would be interpreted as GTMInteger on the Java side.