IBM System i™

RPG and Java working together

Barbara Morris

IBM

i want stress-free IT.
i want control.
i want an i.

Agenda

➔A brief intro to Java classes and objects

- The RPG coding to call Java methods
- How Java works (a naïve view, from RPG's point of view)
- Pitfalls
- Why call Java?

Appendixes:

- Extra coding required when you call Java
- Native methods
- Other reading

Java classes and objects

A Java *class* describes a type of Java *object*. The class acts as a sort of template for the object.

Dog: name= …
numLegs= …

```
class Dog {
    int numLegs = 4;
    String name;
    ...
}
```

d2: name="Sparky"
numLegs=3

```
Dog d1 = new Dog(…);
Dog d2 = new Dog(…);
```

d1: name="Spot"
numLegs=4

Java classes and objects

A Java object is created by calling one of the *constructor* methods of the class. A constructor is easy to spot; it has the same name as the class, and it has no return value coded (it returns an object of the class.)

```
class Dog {
   Dog(String name) {
       ...
   }
   Dog(String name, int legs) {
       ...
   }
```

In Java, the constructor is called using the "new" keyword.

```
Dog myDog = new Dog("Spot");
Dog myDog = new Dog("Sparky", 3);
```

Java classes and objects

Each Java object gets its own copies of the *instance* variables of the class. The *instance* methods of the class work on a particular object in the class; they cannot be called as standalone methods.

```
class Dog {
   int getNumLegs() {
      return numLegs;
   }
}

Dog d = ...;
  if (d.getNumLegs() < 4)
     // Awww, that must be Sparky
```

Note the Java syntax in the call to d.getNumLegs(). The call to getNumLegs is working with the d object.

Java classes and objects

Within an instance method, the special variable “this” refers to the particular instance that the method was called for.

```
void introduceMate(String name) {
   this.mate = name;
   print(this.name + ` is mated to `
        + name);
}
```

The class has an instance variable called “name” and the method has a parameter called “name”; to refer to its instance variable, it uses “this.name.

Java classes and objects

A Java class can have *class variables* that are shared by all the objects in the class.

It can also have *class methods* that apply to all the objects in the class.

The presence of the *static* keyword indicates a class variable or class method.

```
class Dog {
   static int maxLegs = 4;
   static String bestName() {
     return "Rover";
   }
}
```


Agenda

- A brief intro to Java classes and objects
- **The RPG coding to call Java methods**
- How Java works (a naïve view, from RPG's point of view)
- Pitfalls
- Why call Java?

Appendixes:
- Extra coding required when you call Java
- Native methods
- Other reading

What is important for RPG programmers?

An RPG programmer is normally interested in the **methods** of a class. (There is no easy way to get to the variables of a class; normally RPG programs call “get” or “set” methods to work with the variables.)

There are several questions that an RPG programmer must answer before coding the prototype for a Java method.

Prototype questions

The answers to the first four questions come from the Java documentation for the class.

1. What is the fully-qualified name of the class?
2. Is the method a constructor?
3. Is the method static?
4. What are the parameters and return value for the method?

The answer to the fifth question comes from the ILE RPG Programmer’s Guide.

5. How are those parameters and return values coded in RPG?

The answer to the sixth question is up to you.

6. What will be the RPG prototype name for the method?

Java documentation

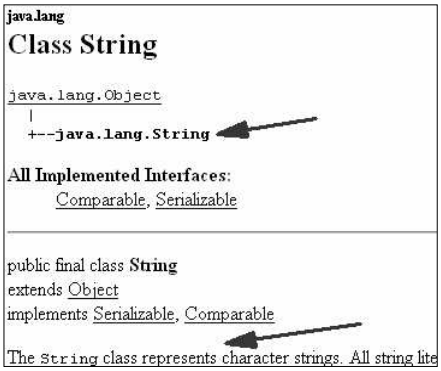
Usually, Java classes are documented in a very convenient and standard way, called “javadoc”.

Here are some excerpts of the javadoc for a Java class.

Java documentation: Intro

The introduction gives us

- the fully qualified name of the class
- a description of the class

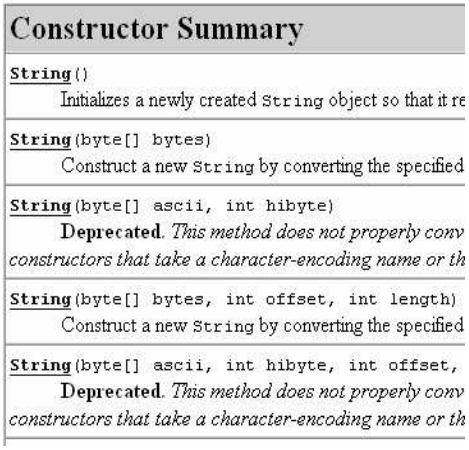


Java documentation: Constructors

The Constructor Summary gives us a list of all the constructors for the class.

A constructor method always has the same name as the class ("String" in this case).

No return value is shown because a constructor always returns an object of its class.



Constructor Summary

String()
Initializes a newly created String object so that it re

String(byte[] bytes)
Construct a new String by converting the specified

String(byte[] ascii, int hibyte)
Deprecated. *This method does not properly conv*
constructors that take a character-encoding name or th

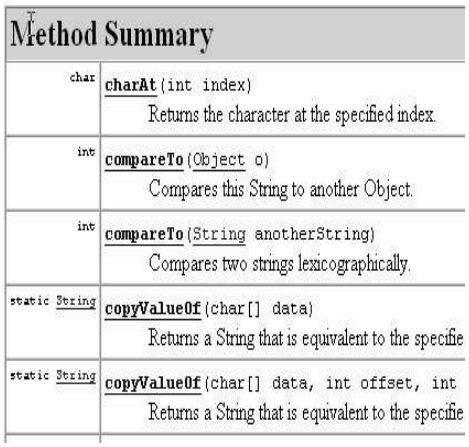
String(byte[] bytes, int offset, int length)
Construct a new String by converting the specified

String(byte[] ascii, int hibyte, int offset,
Deprecated. *This method does not properly conv*
constructors that take a character-encoding name or th

Java documentation: Methods

The Method Summary gives us a list of all the other methods for the class.

Note the two "static" methods here.

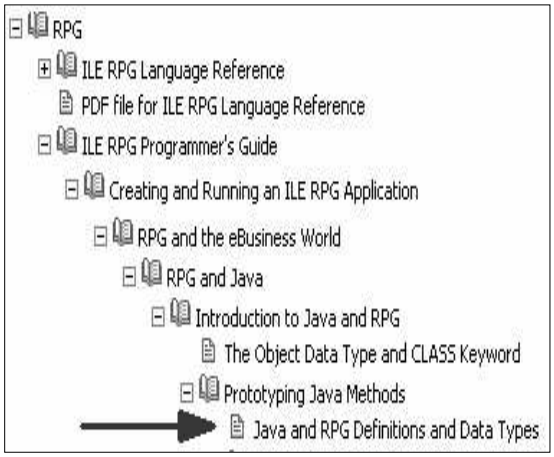


Method Summary

char	**charAt**(int index) Returns the character at the specified index.
int	**compareTo**(Object o) Compares this String to another Object.
int	**compareTo**(String anotherString) Compares two strings lexicographically.
static String	**copyValueOf**(char[] data) Returns a String that is equivalent to the specifie
static String	**copyValueOf**(char[] data, int offset, int Returns a String that is equivalent to the specifie

RPG documentation

Here is the navigation to get to the section that compares Java types to RPG types.



RPG documentation

Here is the table of type equivalence.

Java Data Type	ILE RPG Data Type	RPG Definitions
boolean	indicator	N
byte [1]	integer	3I 0
	character	1A
byte[]	character length > 1 (See 3.)	nA
	array of character length=1 (See 4.)	1A DIM(x)
	date	D
	time	T
	timestamp	Z
short	2-byte integer	5I 0
char	UCS-2 length=1	1C
char[]	UCS-2 length>1 (See 3.)	nC
	array of UCS-2 length=1 (See 4.)	1C DIM(x)
int	4-byte integer	10I 0
long	8-byte integer	20I 0
float	4-byte float	4F
double	8-byte float	8F
any object	object	O CLASS(x)
any array	array of equivalent type (See 4.)	DIM(x)

RPG coding to call Java

Let's look at coding the prototype and a sample call for three methods from the String class.

- A constructor
- An instance method
- A static method

RPG coding to call Java

A prototype for a Java method is the same as an ordinary prototype except that the EXTPROC keyword is always in this form:

```
EXTPROC(*JAVA : classname : methodname)
```

A call to a Java method is like a call to an ordinary prototype procedure (except for one difference when calling instance methods that we will see later).

```
callp someMethod(parms);

x = someOtherMethod(parms);
```

Calling a Java constructor

There are several String constructors; let's use this one:

```
String(byte[] bytes)
```

The questions:

1. Class name **java.lang.String**
2. Constructor? **Yes**
3. Static? **N/A**
4. Java parms? **Parm is byte[], returns Object**
5. RPG parms? **Parm is alpha string, returns Object**
6. RPG name? **newString** (common RPG style)

Choosing the RPG name

If we were also going to use the other common constructor that takes a char array (RPG UCS-2 type), it would not be possible to call them both "newString" in RPG.

```
String(char[] value)
```

In that case, we might call them newStringByte and newStringChar (to use the Java type names), or newStringA and newStringC (to use the RPG types).

Prototype for a Java constructor

Now that we have the answers to the questions, we can code the RPG prototype.

Let's use a named constant for the class name, since we're going to define several prototypes for that class.

```
D stringClass  C          'java.lang.String'


D newString    PR     O   EXTPROC(*JAVA
D                         : stringClass
D                         : *CONSTRUCTOR)
D    value       65535A   VARYING CONST
```


The byte array parameter

The RPG type matching the Java byte array (`byte[]`) is an alpha string, but it can be any size we want. For the String constructor, assume that 65535 is the maximum size we will want to pass.

Code VARYING for the parameter

- Java doesn't mandate the length of a String parameter

Code CONST for the parameter

- Java will not be changing the parameter
- Coding CONST will allow us to pass literals and expressions

RPG Object type

In the previous slide, the return value of the Java method is defined as type 'O'.

```
D newString    PR     O   EXTPROC(*JAVA …
```

'O' stands for 'Object'. Normally, coding the Object type requires also adding the CLASS(*JAVA) keyword to indicate the class of the object, but this is not necessary in the case of a Constructor, because a constructor always returns an Object of the same class as the Constructor method.

Defining an Object variable

In this case, we want to define a reference to an Object of class java.lang.String, to match the return type of the constructor.

```
D myString  S   O  CLASS(*JAVA
D                  : 'java.lang.String')
```

Since we already have a name constant for the class name, this is better:

```
D myString  S   O  CLASS(*JAVA
D                  : 'java.lang.String')
```

Even better is to use the LIKE keyword:

```
D myString  S      LIKE(newString)
```

Calling a Java constructor

We are ready to call the constructor. We have the prototype, and we have the variable to receive the return value.

```
/free
      myString = newString('hello');
```


Calling an instance method

There are many String methods; let's call this popular one:

```
byte[] getBytes()
```

The questions:

1. Class name **java.lang.String**
2. Constructor? **No**
3. Static? **No**
4. Java parms? **No parms, returns byte[]**
5. RPG parms? **Return type is character string**
6. RPG name? **getBytes** (same as Java name)

More about RPG prototype names

It should be easy to associate the RPG name of a method with the Java name.

Choose

- Use the exact same name: getBytes
- Prefix the name with the class name, if necessary: String_getBytes
- Suffix the name with parameter information, if necessary: String_getBytes_enc, say if you were coding the prototype for this method:

```
byte[] getBytes(String enc)
```


Prototype for an instance method

When the prototype for a Java method has keywords associated with the return type, code those keywords on the first line, and code the EXTPROC on the next line. (This is not required by RPG, but it makes it easier to understand the prototype.)

```
D getBytes   PR  65535A   VARYING
D                         EXTPROC(*JAVA
D                         : stringClass
D                         : 'getBytes')
```

Calling an instance method

Recall the syntax that Java uses when calling an instance method

```
d1.getNumLegs()
```

When calling an instance method in RPG, you have to pass the Object instance as the first parameter when you call the method.

```
getNumLegs(d1)
```

If the RPG prototype shows 3 parameters, you pass 4 parameters when you call the method: an *instance* parameter followed by the other 3 parameters.

Calling an instance method

We already created a String object earlier when we called the constructor. We'll call the getBytes() method for that object.

Recall that the getBytes prototype showed no parameters. That means we will pass exactly one parameter: the instance parameter.

```
D val        S       65535A  VARYING
 /free
      val = getBytes(myString);
```

Calling an instance method

Let's call the instance method for two different instances.

```
D string1  S      LIKE(newString)
D string2  S      LIKE(newString)
D val1     S      LIKE(getBytes)
D val2     S      LIKE(getBytes)
 /free
    string1 = newString('hello');
    string2 = newString('summit');
    val1 = getBytes(string1);
    val2 = getBytes(string2);
```

The getBytes() method returns the value of the string as a Java byte array.

Quiz: Calling an instance method

```
string1 = newString('hello');
string2 = newString('summit');
val1 = getBytes(string1);
val2 = getBytes(string2);
```

What do val1 and val2 hold?

We created two separate String instances when we called the String constructor twice. Calling the instance method getBytes() for each instance would return the value associated with that instance.

Calling a static method

Let's call this method:

```
static String valueOf(boolean b)
```

The questions:

1. Class name **java.lang.String**
2. Constructor? **No**
3. Static? **Yes**
4. Java parms? **parm is boolean, returns String**
5. RPG parms? **parm is indicator, returns Object**
6. RPG name? **valueOf_boolean** (Java name, with type)

The RPG prototype name

Here is an example where the RPG name should probably not be simply the Java name. There are eight methods in the String class with the name 'valueOf'.

Adding the Java parameter type to the RPG prototype name is a convenient way to differentiate these particular Java methods.

```
 * static String valueOf(int i)
D valueOf_int       PR


* static String valueOf(float f)
D valueOf_float     PR
```

Prototype for a static method

The prototype for a static Java method must have the STATIC keyword. If the Java method has “static”, then the RPG prototype must have “STATIC”.

```
D valueOf_boolean...
D                 PR              LIKE(newString)
D                                 EXTPROC(*JAVA
D                                  : stringClass)
D                                  : 'valueOf')
D                                 STATIC
D   b                          N  VALUE
```

Parameters matching Java primitives must be passed by value.

Call to a static method

Calling a static Java method is the same as calling an ordinary RPG procedure.

```
D string   S        LIKE(newString)
 /free
    string = valueOf(*on);
```

Java boolean variables have the values *true* or *false*.

```
    val = getBytes(string);
```

After the call to getBytes(), val has the value ‘true’.

Comments for your Java prototypes

To save space for the examples in this presentation, I didn't code any comments for the prototypes.

Besides any standard comments you code for your prototypes, you should add comments with the Java class and the Java method header; you can copy the method header from the Java documentation.

Having the exact Java information makes it much easier to debug your RPG prototypes.

```
 * Class:  java.lang.String
 * Method: String valueOf(boolean b)
D valueOf_boolean...
D            PR       LIKE(newString)
  etc
```


Try one

```
 * Java class: java.lang.StringBuffer
 * Method: int lastIndexOf(String str,
 *                         int fromIndex)
DName++++++ETDs+..L+++IDc.Keywords+++++++++
D __________  PR     _____ __________________
D                          EXTPROC(*JAVA
D                           : _______________
D                           : _______________)
D                          __________________
D   str                    LIKE(newString)
D                          __________________
D   fromIndex        _____ __________________
```

Agenda

- A brief intro to Java classes and objects
- The RPG coding to call Java methods
- **How Java works (a naïve view, from RPG's perspective)**
- Pitfalls
- Why call Java?

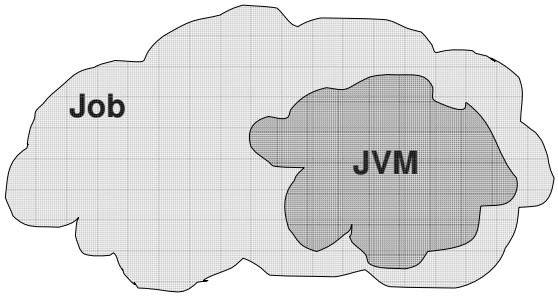
Appendixes:
- Extra coding required when you call Java
- Native methods
- Other reading

The JVM

Java runs in a **Java Virtual Machine** (JVM).

A JVM is associated with a job.

- A job may or may not have a JVM
- A JVM is associated with a single job



Java objects and object references

Consider this fragment of Java code:

```
String str = new String("abc");
```

The variable "str" is an *object reference*. It is not a String Object, it only refers to a String object.

What follows may not be exactly true, but it is convenient:

When Java creates the new object, it puts all the information about the object in an array, and it uses the array index as the object reference.

When the object reference is used later, Java works with the actual object using its array to locate it.

Java objects and object references

RPG code that uses Java objects works with object references too. When your RPG procedure calls the String constructor, Java returns a reference to the new String "abc".

```
myString = newString('abc');
```

If you were debugging your RPG program, and evaluated variable myString in the debugger, it would show a value like 13 or 5; you could not see the "abc".

The actual contents of a Java object are internal to Java.

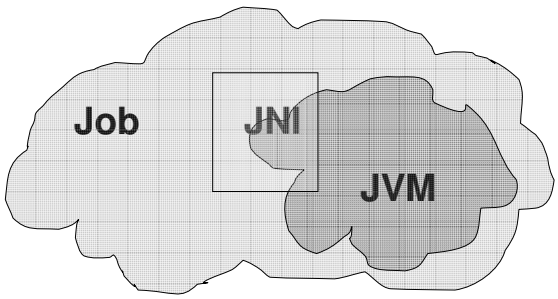
You can only work with the object by calling its instance methods.

JNI

Communication between Java and RPG is done via the Java Native Interface (JNI).

JNI is part of Java.

JNI contains a set of APIs that any native (non-Java) program can call.



What happens when RPG calls Java?

1. If it is the first Java call for the job, RPG calls a JNI API to start the JVM.
2. RPG calls the Java method using several JNI calls:
   - Transform RPG parameters to Java equivalents
   - Look up the Java class from the RPG prototype
   - Look up the Java method using
     - Method name
     - Signature (based on the return type and parameter types from the prototype)
     - Method type (constructor, instance, static)
   - Call the method
   - Transform Java return value to RPG equivalent
   - Transform non-CONST parameters to RPG equivalents

Importance of the RPG prototype

Did you notice how every bit of information from the RPG prototype is used to make the call?

If any of the following are incorrect, the Java method cannot be called:

- The fully qualified class name
- The method name
- The return value type and parameters types
- Whether the method is static or not

The class name, method name, and the class of any Object return value or parameters are all **case-sensitive**.

Garbage collection

Java periodically checks the objects that it has created.

If it finds any objects that no longer have any references, it cleans up those objects.

This is called *garbage collection*.

Garbage collection

Consider this Java code:

```
str1 = new String("abc");
str1 = new String("def");
```

The first assignment to str1 creates a reference to String “abc”. The second assignment creates a reference to “def” **and removes the reference to “abc”**.

Now the String “abc” has no references.

Garbage collection

Now consider this Java code:

```
str2 = new String("abc");
```

“abc” has one reference, string2.

```
str3 = str2;
```

“abc” has two references, string2 and string3.

```
str2 = new String("def");
```

“abc” has one reference, string3.

“def” has one reference, string2.

Garbage collection

Let's be Java for a moment.

Fill the table for these Java statements:

```
s1 = "abc";
s2 = s1;
s3 = s1;
s2 = "def";
s1 = null;
```

Now, do garbage collection; cross out the objects with no references.

Object	References

Quiz: Garbage collection vs RPG

Consider this RPG code:

```
str1 = newString('abc');
str1 = newString('def');
```

Question: How many references does Java have for string "abc"?

Answer: ?

Garbage collection vs RPG

The RPG code looks very similar to the equivalent Java code, but from Java's point of view it is very different.

When RPG calls Java, Java has no way of knowing that the same RPG variable was used to hold the object references.

Garbage collection

Let's be Java again.

Fill the table for these RPG statements.

```
s1 = newString('abc');
s2 = s1;
s3 = s1;
s2 = newString('def');
s1 = *null;
```

Now, do garbage collection; cross out the objects with no references.

Object	References

Garbage collection

```
s1 = newString('abc');
s2 = s1;
s3 = s1;
s2 = newString('def');
s1 = *null;
```

Java only knows about the calls to the String constructor.

Garbage collection cannot free "abc" or "def".

Object	References
"abc"	Native reference
"def"	Native reference

How to assist Java with garbage collection

Good news:

It's easy to let Java know you are finished with your objects.

At strategic points in your code,

1. Tell Java you are going to start creating some objects.
2. Tell Java you are finished with all the objects you created since step 1.

How to assist Java with garbage collection

```
...
beginObjGroup(env);
s1 = newString('abc');
s2 = s1;
s3 = s1;
s2 = newString('def');
s1 = *null;
endObjGroup(env);
...
```

Object	References

Agenda

- A brief intro to Java classes and objects
- The RPG coding to call Java methods
- How Java works (a naïve view, from RPG's perspective)
- **Pitfalls**
- Why call Java?

Appendixes:
- Extra coding required when you call Java
- Native methods
- Other reading

What can go wrong will go wrong

Getting started calling Java from RPG is usually very easy.

There is no particular setup required to make simple Java calls like creating String objects.

The next “real” step often involves calling Java classes that are not part of the standard Java classes.

Getting your setup right the first time

As soon as you need to call your own custom Java classes, or classes that you have obtained, you have some extra steps to take.

Make sure you have taken care of these before you try out your RPG program:

- The classpath
- The minimum Java version needed

Changes to your setup

If you change anything in your Java setup
- Environment variables
- Properties file
- Add a new Java class
- Change a Java class

Then you must start a new JVM to see the changes.

Usually this means signing off and signing back on again.

Agenda

- A brief intro to Java classes and objects
- The RPG coding to call Java methods
- How Java works (a naïve view, from RPG's perspective)
- Pitfalls
- **Why call Java?**

Appendixes:
- Extra coding required when you call Java
- Native methods
- Other reading

Why call Java?

Java can do many cool things. Most of those cool things can be done other ways, but Java already does them. By calling Java from RPG, you can tap into its abilities.

Besides Java's huge library of standard functions, there are many packages written in Java that extend its abilities.

An example of one of these extensions is the POI-HSSF API from Apache that lets you work with Excel files. Many RPG programmers are familiar with this through Scott Klement's articles and downloads on http://www.systeminetwork.com.

Why call Java?

Many shops have their own custom Java classes. Some of the function provided by those Java classes is also useful to RPG.

If you already have Java code that does a particular task that you want RPG to do you can either

- Rewrite the code in RPG
  - May be difficult or impossible to duplicate
  - If the Java class changes due to business needs, you have to change the RPG code too
- Call the Java code from RPG
  - The Java version is already debugged
  - Changes only have to be made in one place

Converse: Why have Java call RPG?

We didn't talk about Java calling RPG (see Appendix 2), but the same arguments hold.

If your shop already has working RPG code that does some function, it may be better to have Java call the RPG code than to rewrite the code in Java.

Appendix 1: Extra JNI calls that you have to do

We have just seen one example of the extra JNI coding you have to do when you are working with Java; the calls to beginObjGroup and endObjGroup are JNI calls.

Assisting with garbage collection is the most important. If you don't take care of that, Java will eventually run out of room to create more objects, and the JVM will become unusable.

Recall that a job can have only one JVM, so if its JVM is unusable, it's impossible to call Java from RPG any more in that job.

Appendix 1: How to call JNI functions

To help you call the most common JNI functions, the ILE RPG Programmer's Guide has the RPG source for a service program that you can use for your JNI needs. The RPG procedures act as wrappers for the JNI functions.

There is also a working example that uses the service program.

Be sure to use the V6R1 version; the earlier versions had bugs.

Appendix 1: Source for JNI function wrappers

Here is the URL for the page that links to the various pages containing the RPG wrappers.

http://publib.boulder.ibm.com/infocenter/systems/scope/i5os/topic/rzasc/sc092507252.htm#rpgjni

The next page shows what that page of the manual looks like.

To find all the pieces that you need, you will have to follow all the links on that page.

Don't miss the fragment of code on that main page; you will need that in your service program module.

Appendix 1: JNI wrappers in the Programmer's Guide

The module that you create to hold these JNI wrapper functions should begin with the following statements:

```
H thread(*serialize)
H nomain
H bnddir('QC2LE')
 /define OS400_JVM_12
 /copy qsysinc/qrpglesrc,jni
 /copy JAVAUTIL
```

The following RPG wrappers for JNI functions are described. See Figure 93 below for a complete working example. See Copy-file JAVAUTIL, for the copy file containing the prototypes and constants for the wrapper functions.

- Telling Java to free several objects at once
- Telling Java you are finished with a temporary object
- Telling Java you want an object to be permanent
- Telling Java you are finished with a permanent object
- Creating the Java Virtual Machine (JVM)
- Obtaining the JNI environment pointer

Appendix 1: Service program JAVAUTIL: binder source

You can use this binder source to create your service program.

```
/* Add new entries at the end.  Don't change any */
strpgmexp signature('JAVAUTIL')
   export symbol('getJniEnv')             /*   1 */
   export symbol('beginObjGroup')         /*   2 */
   export symbol('endObjGroup')           /*   3 */
   export symbol('freeLocalRef')          /*   4 */
   export symbol('getNewGlobalRef')       /*   5 */
   export symbol('freeGlobalRef')         /*   6 */
endpgmexp
```

Appendix 1: Service program JAVAUTIL: crtsrvpgm

If you created module MYLIB/JAVAUTIL and you put that into source member MYLIB/QSRVSRC JAVAUTIL, create your service program like this:

```
===> CRTSRVPGM MYLIB/JAVAUTIL
            MODULE(MYLIB/JAVAUTIL)
            SRCFILE(MYLIB/QSRVSRC)
            SRCMBR(JAVAUTIL)
```


Appendix 1: Using service program JAVAUTIL

It's convenient to put the service program in a binding directory.

```
===> CRTBNDDIR BNDDIR(MYLIB/JAVAUTIL)
===> ADDBNDDIRE BNDDIR(MYLIB/JAVAUTIL)
           OBJ((MYLIB/JAVAUTIL *SRVPGM))
```

I call my binding directory JAVAUTIL, and I have my prototypes in QRPGLESRC member JAVAUTIL.

I code this in any module where I want to call the wrapper procedures.

```
H thread(*serialize)
H bnddir('JAVAUTIL')
 /copy JAVAUTIL
```

Appendix 2: Native methods – Java coding

It is also possible to have Java call your RPG procedures using JNI, **if your Java is running on the same system as your RPG**.

It's quite simple:

- In your Java source:
  1. Code the method header, like this:

```
native int myNativeMeth(String s);
```

  2. Code a section to tell Java where to find the service program with your native methods

```
static {
  System.loadLibrary("MYSRVPGM");
}
```


Appendix 2: Native methods – RPG coding

- In the RPG source for your native service program:
  1. Code the usual *JAVA prototype for the method

```
D myNativeMeth  PR 10I 0  EXTPROC(*JAVA
D                         : 'MyClass'
D                         : 'MyNativeMeth')
D   S                O    CLASS(*JAVA
D                         : StringClass)
```

  2. Code the RPG procedure as usual:

```
P myNativeMeth  B         EXPORT
D myNativeMeth  PI 10I 0
D   S                O    CLASS(*JAVA
D                         : StringClass)
  etc
```

Appendix 2: Native methods - %THIS

Recall that a Java instance method can refer to the current instance using the special variable “this”.

You can do this in your RPG native method using the builtin function %THIS.

If myNativeMeth in the previous example wanted to call another instance method in the same class, it would pass %THIS as the instance parameter.

```
/free
    limit = getNumItems(%this);
```


Appendix 2: Native methods – garbage collection

When working with native methods, enabling garbage collection is easy because Java automatically frees any objects created while the native method was running.

The problems with native methods can arise when you call your native method again.

Appendix 2: Native methods – two problems

1. If you have static object references in your RPG module, either in global definitions or static local definitions, the object references will not get set to *NULL when your native method is called again, but the objects that they refer to will no longer exist. Trying to use those references again will cause various and sometimes unpredictable problems.

2. If you **don't** want an object to be freed between calls your native method, you can call the JNI wrapper getNewGlobalRef. You will have to explicitly free it using freeGlobalRef.

Appendix 3: Other reading

Here are some other things to read about Java in general, and RPG calls to Java in particular:

- The "RPG and Java" section of the ILE RPG Programmer's Guide in the Info Center
  http://publib.boulder.ibm.com/infocenter/systems/scope/i5os/topic/rzahg/rzahgrpgprogpdf.htm
- The Programming > Java topic in the Info Center
  http://publib.boulder.ibm.com/infocenter/systems/scope/i5os/index.jsp
- The JNI Tutorial from Sun
  http://java.sun.com/developer/onlineTraining/Programming/JDCBook/jni.html
- Scott Klement's articles about RPG using Java to work with Excel:
  http://systeminetwork.com/archivesearch/article/hssf+poi