Some Comments on using Java with OCTAVE (and MATLAB)

Martin Hepperle
May 2010

Note: this document is based on the package version `java-1.2.7`.
The java package is usually installed in `…\share\octave\packages\java-1.2.7`.

Available Functions

```
javaaddpath ( path )
```

Add `path` to the dynamic class path of the Java virtual machine. `path` can be either a directory where .class files can be found, or a .jar file containing Java classes.

See also:
`javaclasspath`

```
P = javaclasspath
```

Return the dynamic class path of the Java virtual machine in the form of a cell array of strings. If no output variable is given, the result is simply printed on the standard output.

See also:
`javaaddpath`

```
a = javaArray ( CLASS, [M,N,...] )
a = javaArray ( CLASS, M, N,... )
```

Create a Java array of size `[M, N, ...]` with elements of class `CLASS`. `CLASS` can be a Java object representing a class or a string containing the fully qualified class name.

The generated array is uninitialized, all elements are set to null if `class` is a reference type, or to a default value (usually 0) if `CLASS` is a primitive type.

Example:

```
a = javaArray ( "java.lang.String", 2, 2 );
a(1,1) = "Hello";
```

```
OBJ = java_new (NAME, ARG1, ...)
```

Create a Java object of class `NAME`, by calling the class constructor with the arguments `ARG1`, ...

Example:

```
x = java_new ("java.lang.StringBuffer", "Initial string");
```

See also:

```
java_invoke
java_get
java_set
```

```
RET = java_invoke (OBJ, NAME, ARG1, ...)
```

Invoke the method NAME on the Java object OBJ with the arguments ARG1, ... For static methods, OBJ can be a string representing the fully qualified name of the corresponding class. The function returns the result of the method invocation.

When OBJ is a regular Java object, the structure-like indexing can be used as a shortcut syntax. For instance, the two following statements are equivalent

```
ret = java_invoke (x, "method1", 1.0, "a string")
ret = x.method1 (1.0, "a string")
```

See also:

```
java_get
java_set
java_new
```

```
VAL = java_get (OBJ, NAME)
```

Get the value of the field NAME of the Java object OBJ. For static fields, OBJ can be a string representing the fully qualified name of the corresponding class.

When OBJ is a regular Java object, the structure-like indexing can be used as a shortcut syntax. For instance, the two following statements are equivalent

```
java_get (x, "field1")
x.field1
```

See also:

```
java_set
java_invoke
java_new
```

```
OBJ = java_set (OBJ, NAME, VAL)
```

Set the value of the field NAME of the Java object OBJ to VAL. For static fields, OBJ can be a string representing the fully qualified named of the corresponding Java class.

When OBJ is a regular Java object, the structure-like indexing can be used as a shortcut syntax. For instance, the two following statements are equivalent

```
java_set (x, "field1", val)
x.field1 = val
```

See also:

```
java_get
java_invoke
java_new
```

FAQ

How to write code that is compatible with OCTAVE and MATLAB?

OCTAVE and MATLAB are very similar, but handle Java slightly different. Therefore it is necessary to detect the environment and use the appropriate methods.

```
v = version;

if str2num ( v(1:3) ) < 5.0
   % assume Octave has a version number < 5
   FrameWork = 'Octave';
else
   % assume Matlab version number > 5
   FrameWork = 'Matlab';
end
```

How to make Java classes available?

Java finds classes by searching its `classpath`. While MATLAB manages classpaths globally in a file named "classpath.txt", OCTAVE manages them with the `javaaddpath` method in your script. MATLAB comes with an `import` statement, which directly makes class names available to your script.
If you want to run the script example in OCTAVE, you must also prepare a dummy function file `import.m` to satisfy the OCTAVE parser.

See also "How to create an instance of a Java class?"

```
if FrameWork == 'Matlab'
   % Note that in Matlab you must 'edit classpath.txt'.
   % Octave needs a dummy function file import.m because it
   % does not know the "import" statement
   import ( 'package.FirstClass' );
   import ( 'another.package.TouristClass' );
else
   % adapt to your installation
   base_path = 'C:/octave/java_files';
   javaaddpath ( [base_path, '/someclasses.jar'] );
   javaaddpath ( [base_path, '/moreclasses.jar'] );
end
```

The contents of the file import.m for OCTAVE is:

```
function import ( path )
      % dummy for Octave to mimic Matlab statement
end
```

How to create an instance of a Java class?

If you code shall work under OCTAVE as well as MATLAB you should implement two different ways to create an object instance.
Of course you could wrap the creator into a generic platform independent function, possibly with a variable number of arguments.

```
% Depending on environment creation is slightly different.
% Therefore it may be useful to place instantiation into
% a specific function

function [Passenger] = createPassenger ( FrameWork, row, seat )
```

```
   if FrameWork == 'Matlab'
      Passenger = FirstClass ( row, seat );
   else
      Passenger = java_new ('package.FirstClass', row, seat );
   end

end
```

How can I handle memory limitations?

The Java Virtual Machine (JVM) is usually created with a fixed amount of initial memory and also a fixed limit of maximum memory, independent of how much main memory your computer has. This is obviously a relic from times when Java was intended to be mainly used for applets inside web browsers. When the maximum memory limit is hit, Java code may fail or throw errors.

In OCTAVE as well as in MATLAB, you can specify options for the creation of the JVM in a options file named `java.opts`. Here you can specify memory limits.

For example you can increase the maximum amount of memory available to the Java machine to 256 Megabytes by adding the following line to the `java.opts` file. Additionally the example defines a system property which could be used by Java programs.

```
-Xmx256m
-DMyProperty=12.34
```

In OCTAVE, the options file must be located in the directory where `javaclasspath.m` resides, i.e. the package installation directory, usually something like `…\share\octave\packages\java-1.2.6`.
In MATLAB, the options file goes into the MATLABROOT/bin/ARCH directory or in your personal MATLAB startup directory (can be determined by a `pwd` command). MATLABROOT is the MATLAB root directory and ARCH is your system architecture, which you find by issuing the commands `matlabroot` respectively `computer('arch')`.

How to compile and install the java package in OCTAVE on Windows?

Most OCTAVE installations come with the `java` package preinstalled. In case you want to replace this package with a more recent version, you must perform the following steps:

1) Check and if necessary uninstall the currently installed package

```
pkg list
```

If the `java` package shows up in the list you can uninstall it by issuing the command

```
pkg uninstall java
```

2) Make sure that the build environment is configured properly

The installation process requires that the environment variable **JAVA_HOME** points to the Java Development Kit (JDK) on your computer.

- JDK is not equal to JRE (Java Runtime Environment). The JDK home directory contains subdirectories with include, library and executable files which are required to compile the **java** package. These files are not part of the JRE.
- Do not use backslashes but ordinary slashes in the path.

```
Octave> setenv("JAVA_HOME","C:/Java/jdk1.6.0_18");
```

3) Compile and install the package in OCTAVE:

If you have for example saved the package archive on your Z: drive the command would be

```
Octave> pkg install z:/java-1.2.7.tar.gz
```

or if you are curious and want to see many warnings (but hopefully no errors)

```
Octave> pkg install -verbose z:/java-1.2.7.tar.gz
Octave> pkg list
```

The java package should not be listed anymore. If you have used the **java** package during the session of OCTAVE, you may have to exit and restart Octave to be able to uninstall the package.

4) Afterwards, test the java package.

```
Octave > s = java_new ( 'java.lang.String', 'Hello OctaveString' )
s = Hello OctaveString
```

Note that the java packages automatically transforms the Java String object to an Octave string. This means that you cannot apply Java String methods to the result.

This “auto boxing” scheme seems to be implemented for the following Java classes:

- **java.lang.Integer**
- **java.lang.Double**
- **java.lang.Boolean**
- **java.lang.String**

If you instead create an object for which no “auto-boxing” is implemented, you receive the genuine Java object:

```
Octave > v = java_new ( 'java.util.Vector')
v =
<Java object: java.util.Vector>
Octave > v.add(12);
Octave > v.get(0)
ans = 12
```

In this case you can apply all methods of the Java object.

Note also that for some objects you have to specify an initializer:

```
% oh no:
Octave > d = java_new ( 'java.lang.Double')
error: [java] java.lang.NoSuchMethodException: java.lang.Double
% but:
Octave > d = java_new ( 'java.lang.Double',12.34)
d =  12.340
```

Additional note:

If you want to get rid of many compiler warnings, you can change all calls to the function `find_octave_class()` from

```
find_octave_class (jni_env, "org/octave/ClassHelper")
```

by adding a type cast into

```
find_octave_class (jni_env, (char *)"org/octave/ClassHelper")
```