javacall — Call a Java plugin

Description

`javacall` calls a Java plugin by invoking a static method. The *method* to be called must be implemented with a specific Java signature in the following form:

```
static int java_method_name(String[] args)
```

`javacall` requires *class* to be a fully qualified name that includes the class's package specification. For example, to call a method named `method1` from class `Class1`, which was part of package `com.mydomain` and packaged in `myjarfile.jar`, the following command would be used:

```
. javacall com.mydomain.Class1 method1, jars(myjarfile.jar)
```

`javacall` can parse a varlist, along with `if` and `in` qualifiers. The Data class in the Java-Stata API Specification has methods for interpreting those parsed values.

Syntax

`javacall` *class method* [*varlist*] [*if*] [*in*] , {`jars(`*jar_files*`)` | `classpath(`*classpath*`)`}
 [`args(`*arg_list*`)`]

Options

`jars(`*jar_files*`)` specifies the JAR files to be added to the class path. *jar_files* may be one JAR file or a list of JAR files separated either by spaces or by semicolons. Stata will search along the ado-path for the specified JAR files and add them to the Java class path for the plugin. Either `jars()` or `classpath()` must be specified.

`classpath(`*classpath*`)` specifies the class path to use. *classpath* may be a single class path or multiple paths specified using a platform-specific Java class path. On Windows, multiple paths are separated by semicolons. On Mac and Unix, multiple paths are separated by colons. Either `jars()` or `classpath()` must be specified.

This option is provided as a convenience for use during the development process. For example, a developer might use this option to set the class path to the directory where their compiler is generating `.class` files, allowing newly compiled code to be tested quickly without the need to build a JAR file. After the development process is complete, a JAR file should be created, and the `jars()` option should be used instead.

`args(`*args_list*`)` specifies the *args_list* that will be passed to the Java method as a string array. If `args()` is not specified, the array will be empty.

Remarks and examples

Each Java plugin uses its own instance of the class loader, allowing the currently loaded plugin to be discarded and a new version of the plugin to be loaded. Because each plugin uses a separate instance of the class loader, dependencies are not shared globally. A plugin developer can bundle their plugin with any third-party dependencies using a single JAR file, or dependencies may be distributed in multiple JAR files. Plugin isolation occurs because the `jars()` option allows each plugin to use a unique set of JAR files.

▷ Example 1

Consider two variables needing to store integers too large to be held accurately in a double or a long, so instead they are stored as strings. If we needed to subtract the values in one variable from another, we could write a plugin using Java's BigInteger class. The following code shows how we could perform the task:

```
/* Java class begins here */
import java.math.BigInteger;
import com.stata.sfi.*;
public class MyClass {
   /* Define the static method with the correct signature */
   public static int sub_string_vals(String[] args) {
      long nobs1 = Data.getObsParsedIn1() ;
      long nobs2 = Data.getObsParsedIn2() ;
      BigInteger b1, b2 ;
      if (Data.getParsedVarCount() != 2) {
          SFIToolkit.error("Exactly two variables must be specified\n") ;
          return(198) ;
      }
      if (args.length != 1) {
          SFIToolkit.error("New variable name not specified\n") ;
          return(198) ;
      }
      if (Data.addVarStr(args[0], 10)!=0) {
          SFIToolkit.errorln("Unable to create new variable " + args[0]) ;
          return(198) ;
      }
      // get the real indexes of the varlist
      int mapv1 = Data.mapParsedVarIndex(1) ;
      int mapv2 = Data.mapParsedVarIndex(2) ;
      int resv  = Data.getVarIndex(args[0]) ;
      if (!Data.isVarTypeStr(mapv1) || !Data.isVarTypeStr(mapv2)) {
          SFIToolkit.error("Both variables must be strings\n") ;
          return(198) ;
      }
      for(long obs=nobs1; obs<=nobs2; obs++) {
          // Loop over the observations
          if (!Data.isParsedIfTrue(obs)) continue ;
          // skip any observations omitted from an [if] condition
          try {
              b1 = new BigInteger(Data.getStr(mapv1, obs)) ;
              b2 = new BigInteger(Data.getStr(mapv2, obs)) ;
              Data.storeStr(resv, obs, b1.subtract(b2).toString()) ;
          }
          catch (NumberFormatException e) { }
      }
      return(0) ;
   }
}
/* Java class ends here */
```

Consider the following data, containing two string variables with four observations:

```
. input str20 big1 str20 big2
     29811231010193176  29811231010193168
     42981123101023696  42981123101023669
    -98121437010116560 -98121437010116589
     1000               999
  end
. list
```

```
        ┌─────────────────────────────────────────┐
        │              big1                  big2 │
        ├─────────────────────────────────────────┤
    1.  │   29811231010193176   29811231010193168 │
    2.  │   42981123101023696   42981123101023669 │
    3.  │  -98121437010116560  -98121437010116589 │
    4.  │                1000                 999 │
        └─────────────────────────────────────────┘
```

Next we call the Java method using `javacall`. The two variables to subtract are passed in as a varlist, and the name of the new variable is passed in as a single argument using the `args()` option.

```
. javacall MyClass sub_string_vals big1 big2, args(result1) jars(test.jar)
. list
```

```
        ┌───────────────────────────────────────────────────┐
        │              big1                  big2   result1 │
        ├───────────────────────────────────────────────────┤
    1.  │   29811231010193176   29811231010193168         8 │
    2.  │   42981123101023696   42981123101023669        27 │
    3.  │  -98121437010116560  -98121437010116589        29 │
    4.  │                1000                 999         1 │
        └───────────────────────────────────────────────────┘
```

Normally, a program should be used as a wrapper for `javacall`; see [U] **18 Programming Stata**. For example,

```
program subtract_str
    version 18.0            // (or version 18.5 for StataNow)
    syntax varlist [if] [in], result(string)
    confirm new variable `result'
    javacall MyClass sub_string_vals `varlist' `if' `in',   ///
            args(`result') jars(test.jar)
end
. subtract_str big1 big2, result(bigres)
. list
```

```
        ┌──────────────────────────────────────────────────┐
        │              big1                  big2   bigres │
        ├──────────────────────────────────────────────────┤
    1.  │   29811231010193176   29811231010193168        8 │
    2.  │   42981123101023696   42981123101023669       27 │
    3.  │  -98121437010116560  -98121437010116589       29 │
    4.  │                1000                 999        1 │
        └──────────────────────────────────────────────────┘
```

◁

Also see

[P] **Java intro** — Introduction to Java in Stata

[P] **Java integration** — Java integration for Stata

[P] **Java plugin** — Introduction to Java plugins

[P] **Java utilities** — Java utilities