Title

stata.com

java — Java plugins

Description Usage Remarks and examples Also see

Description

Java plugins are Java programs that you can call from Stata. When called from Stata, a Java plugin has the ability to interact with Stata's dataset, matrices, macros, scalars, and more. Programmers familiar with Java can leverage Java's extensive language features. There are also many third-party libraries available. If you are not an experienced Java programmer and you intend to implement a Java plugin, you should start by learning to program Java. Once you become a proficient Java programmer, writing a Java plugin for Stata should be relatively easy.

Usage

Java programs are compiled into class files and optionally bundled into Java Archive (JAR) files. Class files and JAR files must be placed in the correct location for the Java Runtime Environment (JRE) to find them. The JRE relies on the Java classpath for this task. When Stata initially loads the JRE, the classpath is set to reflect Stata's ado-path. All class and JAR files must be located within Stata's ado-path or in a directory named `jar` within the ado-path. For example, if your personal directory is `C:\ado\personal\`, then you would need to place your compiled Java files in `C:\ado\personal\` or `C:\ado\personal\jar\`. Similarly, all other ado-path directories and `jar` directories along the ado-path are added to the Java classpath when the JRE is initially loaded.

A typical Java stand-alone program has an entry point through a `main()` method, which looks like this:

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

To call a Java plugin from Stata, you must define a similar entry point. However, there are two important distinctions. First, you may name your method whatever you like as long as it conforms to standard Java naming requirements. Second, your method must have a return type of `int` instead of `void`. Here is an example of a valid method signature that Stata can call:

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

Let's assume that `mymethod()` really exists and the compiled Java files have been placed in an appropriate location. To call `mymethod()`, we use Stata's `javacall` command. `javacall` allows you to invoke any static method in the classpath if that method follows the correct signature as described above.

For a Java plugin to be useful, it needs to have access to certain functionality in Stata. We provide Java packages to address those needs. Refer to Java-Stata API Specification for details.

Remarks and examples

stata.com

When a programmer is developing and testing a Java program, it is important to understand when the JRE is loaded and its effect.

The JRE loads the first time that it is needed. That can happen if internal Stata functionality requires Java or if Java is needed for some user-written command. Java's classpath is set when the JRE is loaded, and it cannot be modified afterward (that is, modifying the ado-path after the JRE has loaded will not change the classpath). For the end user who is consuming a completed Java plugin, the process of how Java plugins are loaded is not important because it happens transparently. However, for the programmer who is modifying and testing code, it is very important to understand the process.

Assume you are implementing a Java method named `mymethod()`. You have compiled it, placed the class or JAR file in the correct location, and call it for the first time using `javacall`. Perhaps it executes correctly, but you want to make some modification. You edit the source code, compile it, and copy it to the correct location. If you are using the same Stata session, your changes will not be reflected when you call it again. To reload a Java plugin, Stata must be restarted.

If you intend to redistribute your Java plugin through Stata's `net` (see [R] **net**) command, you must always bundle your compiled program into a JAR file. This is important because `net` copies `.class` files as text instead of binary because of text-based Stata `class` files.

▷ Example 1

Consider two variables that store integers too large to be stored 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 utilizing 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.*;
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.error("Unable to create new variable " + args[0] + "\n") ;
          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:

```
. list
```

	big1	big2
1.	29811231010193176	29811231010193168
2.	42981231010023696	42981231010023669
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 `args(`*argument_list*`)`.

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

	big1	big2	result1
1.	29811231010193176	29811231010193168	8
2.	42981123101023696	42981123101023669	27
3.	-98121437010116560	-98121437010116589	29
4.	1000	999	1

Also see

[P] **javacall** — Call a static Java method