Title

stata.com

Java integration — Java integration for Stata

Description Syntax Option Remarks and examples Also see

Description

`java` creates an instance of a Java environment for executing Java code within Stata. In this environment, Java code does not need to be compiled or bundled into a Java Archive (JAR) file. This allows Java code to be executed interactively, in do-files, and in ado-files. Stata's datasets, matrices, macros, scalars, and more can be accessed using the Java-Stata API Specification.

`java[:]` creates a Java environment in which Java code can be executed in a Read-Evaluate-Print-Loop environment, similar to JShell in Java 9 and later versions.

`java:` *istmt* executes one Java simple statement or several simple statements separated by semicolons.

`java clear` clears all instances of the Java environment. This means that the global environment and all environments associated with ado-files will be destroyed.

Syntax

Syntax is presented under the following headings:

Calling Java from Stata

Enter Java environment

`java` [*varlist*] [*if*] [*in*] [`, shared(`*keyname*`)`] [`:`]

Execute Java simple statements

`java` [*varlist*] [*if*] [*in*] [`, shared(`*keyname*`)`] `:` *istmt*

Clear all instances of the Java environment

```
java clear
```

A colon (`:`) tells the Java instances to exit the interactive mode if an error is encountered.

istmt is either one Java simple statement or several simple statements separated by semicolons.

Instance commands

The following commands can be issued inside the Java environment:

Exit the Java session

```
end
```

Show help information about the rest of the Java instance commands

```
/help
```

Set or display the class-path for the environment. When called without an argument, the current class-path will be displayed. The class-path must be set before calling anything depending on it; otherwise, you must call `/reset`.

`/cp` [*jar_file* | *path*]

Read a Java file, and execute the source in Stata's Java environment

`/open` *file* | *path*

Show all imported packages

```
/imports
```

Reset the instance as if it were completely new

```
/reset
```

Show all active and inactive variables

```
/vars
```

Show all method declarations and unresolved references if they exist

```
/methods
```

Show all type declarations and unresolved references if they exist

```
/types
```

Show all source snippets given in the current Java environment

```
/list
```

Option

`shared(`*keyname*`)` specifies that a shareable instance of Java, named *keyname*, be invoked. This allows you to share an instance across ado-files. *keyname* must be a valid Stata name.

Remarks and examples

stata.com

Remarks are presented under the following headings:

How the environment works

`java` provides utilities for integrating Java with Stata. `java` creates an instance of the Java environment that allows you to execute Java code interactively or in do-files and ado-files.

The `java` environment has different behavior based on how it is used. When used interactively or in do-files, class definitions and instance variables share a global instance of the environment. So a class defined in a do-file can also be referenced interactively or from another do-file. On the other hand, class definitions and instance variables that are defined in ado-files get their own unique instance of the environment by default. The `shared()` option can be used to override that default behavior. By limiting the scope of the environment associated with ado-files, you can make each ado-file behave autonomously without worry of class definitions and instance variables colliding in other ado-files.

Each `java` environment automatically imports `java.util.*`, `java.io.*`, `com.stata.sfi.*`, and `com.stata.sfi.util*` when initialized. Other packages can be imported in the usual way by using `import` statements in your code.

For information on Java versions supported by this integration, see [P] **Java utilities**.

Invoking Java interactively

To invoke Java interactively, you must type either `java` or `java:`. Including a colon tells the Java instances to exit the interactive mode if an error is encountered.

When you execute single statements, a semicolon at the end of the statement is optional. When you execute multiple or complex statements, semicolons are required to delimit the statements.

Below, we demonstrate the two syntaxes:

```
. java
------------------------------ java (type end to exit and /help for help) ------------
java> int x = 1
x ==> 1
java> int y = 2; x + y;
y ==> 2
$1 ==> 3
java> end
--------------------------------------------------------------------------------
```

You may have noticed `$1 ==> 3` in the output. When you execute a statement that returns some value without assigning it to a result, it will store the value in a temporary variable for you. You can access those variables by their names, for example, `int z = $1 + 2`.

To exit your interactive session, type `end`. This will exit your session; however, it will not get rid of your work. If you go back into java, you will be able to access your work. Let's try going back into our environment and looking at the variables we have set.

```
. java
------------------------------ java (type end to exit and /help for help) ------------
java> /vars
|  int x = 1
|  int y = 2
|  int $1 = 3
java> end
--------------------------------------------------------------------------------
```

You can also enter interactive mode for a single statement with the syntax `java:` *istmt*, for example, `java: /vars`.

If you wish to reset your environment, you can type `java: /reset` to reset that instance. Alternatively, you can type `java clear` to clear all Java instances you have, including the ones in ado-files you may have loaded.

Executing Java in a do-file

Java code and Stata code can be executed in the same do-file. To do this, wrap your Java code in `java[:]` and `end`, similar to Python and Mata.

For example, we have the following do-file that calculates the mean of two Stata macros:

```
------------------------------------------------------------------ begin java_ex1.do -
local x = 10
local y = 2

java:
    double mean = (‘x’ + ‘y’) / 2;
    Macro.setLocal("mean", String.valueOf(mean));
end

di ‘mean’
-------------------------------------------------------------------- end java_ex1.do -
```

First, we define two local macros in Stata, `x` and `y`. Inside the Java block, we do basic arithmetic to compute the mean of the two local macros. Then, we use the Stata Function Interface package to set the value of the new `mean` macro in Stata. Macro substitution is a convenient way to pass values from Stata to Java.

Below, we run this do-file:

```
. do java_ex1
. local x = 10
. local y = 2
. java:
------------------------------ java (type end to exit and /help for help) ------------
java> double mean = (‘x’ + ‘y’)/2;
mean ==> 6.0
java> Macro.setLocal("mean", String.valueOf(mean));
$2 ==> 0
java> end
--------------------------------------------------------------------------------

. di ‘mean’
6
.
end of do-file
```

Executing Java in do-files uses the same Java instance as the Command window. We call this the global instance. That means anything you do in this do-file will carry over to the Command window and other do-files.

Executing Java in an ado-file

Unlike do-files, ado-files will get their own instance of Java. This means that anything you do with Java in an ado-file is bound to it by default. However, if you use the `shared()` option, you will be able to access the same instance across multiple ado-files.

Java blocks may be placed in an ado-file but must be placed outside the ado program itself. Functions defined in the `java` block may be called from the ado-file using the `java:` *istmt* syntax.

For example, we have the following ado-file that prints the value of x:

begin `java_program.ado`

```
program java_program
    version 17
    java: printX();
end

java:
    int x = 123;
    void printX() {
        System.out.println("x: " + x);
}
end
```

end `java_program.ado`

To run this program in Stata, we simply type

```
. java_program
x: 123
```

After running `java_program.ado`, if we type `java: x` in the Command window, we will not see a value of 123. This is because `x` is defined only in the context of the ado-file it was defined in. If you ran the example shown in *Invoking Java interactively*, then `x` would be 1; otherwise, it will not be defined.

Executing Java files

Executing Java files in Stata is a little bit different from the traditional way, in which you would normally include dependencies and have a single entry point. With the Java integration, we allow you to run any Java file as if it were passed in line by line into the environment; Stata will search along the ado-path for the specified file. This could mean you simply define classes to use, or you could even set up a dependency in your class-path and do real work in your Java file.

Let's take this example that defines a class called `Addition`, which takes two arguments in its constructor and can return the sum of the two.

begin Addition.java

```
class Addition {

    int x, y;

    public Addition(int x, int y) {
            this.x = x;
            this.y = y;
    }

    public int result() {
            return x + y;
    }

    @Override
    public String toString() {
        return "Addition{" +
                "x=" + x +
                ", y=" + y +
                '}';
    }

}
```

end Addition.java

Below, we will open and use our new class:

```
. java:
------------------------------ java (type end to exit and /help for help) ------------
java> /open Addition.java
java> Addition addition = new Addition(4, 6);
addition ==> Addition{x=4, y=6}
java> int sum = addition.result();
sum ==> 10
java> end
-------------------------------------------------------------------------------
```

Notice that the `Addition` class was declared in the file, but by running this file with `/open`, we declare it in whatever scope calls it. In our case, running `/open` in the Command window results in the `Addition` class being defined in the global instance.

Stata Function Interface examples

Integrating Java code with Stata requires use of the Java-Stata API Specification. This package provides tools to interact with Stata's datasets, matrices, macros, scalars, and more.

For example, if we want to print a list of all the variables in Stata in `auto.dta`, we can type

```
. sysuse auto, clear
(1978 Automobile Data)

. java:
------------------------------ java (type end to exit and /help for help) ------------
java> int parsedVariables = Data.getParsedVarCount();
parsedVariables ==> 12

java> for (int v = 1; v <= parsedVariables; v++) {
 ...> /* Get the real index of parsed vars for varlist support */
 ...> int varIndex = Data.mapParsedVarIndex(v);
 ...> System.out.println(Data.getVarName(varIndex));
 ...> }
make
price
mpg
rep78
headroom
trunk
weight
length
turn
displacement
gear_ratio
foreign

java> end
-------------------------------------------------------------------------------
```

To interpret *varlist*, `if`, and `in` qualifiers, we can make use of a few notable functions in the `com.stata.sfi.Data` class.

To interpret *varlist*, we must first get a count of the variables set to be used in the environment. For this, we use `Data.getParsedVarCount()`. From there, we create an association between variables 1 through N in the environment and their location in the dataset as a whole. We can use `Data.mapParsedVarIndex(v)`, with `v` being the 1-based index starting with the first variable you passed into the environment with *varlist*. For example, if you call `java mpg price:`, `Data.mapParsedVarIndex(1)` will return the index in the dataset where the `mpg` variable is located, which would be 3. Alternatively, `Data.mapParsedVarIndex(2)` will return the index in the dataset where the price variable is located, which would be 2. We need this function because any of the functions in `com.stata.sfi.Data` that take an index as an argument refer to the entire dataset. For example:

```
. java mpg price:
------------------------------ java (type end to exit and /help for help) ------------
java> int parsedVariables = Data.getParsedVarCount();
parsedVariables ==> 2

java> for (int v = 1; v <= parsedVariables; v++) {
 ...> int varIndex = Data.mapParsedVarIndex(v);
 ...> SFIToolkit.displayln(Data.getVarName(varIndex));
 ...> }
mpg
price

java> end
-------------------------------------------------------------------------------
```

To interpret `if`, use the `Data.isParsedIfTrue(int obs)` method. If it returns false, you should not process the observation.

To interpret `in`, use the `Data.getObsParsedIn1()` and `Data.getObsParsedIn2()` methods. For example, if you type `java in 10/50:`, then the return values of `Data.getObsParsedIn1()` and `Data.getObsParsedIn2()` will be 10 and 50, respectively. From there, you can set up a loop to iterate over only those observations, like so:

```
. sysuse auto, clear
. java in 1/50:
java> long obsStart = Data.getObsParsedIn1();
java> long obsEnd = Data.getObsParsedIn2();
java> for (long i = obsStart; i <= obsEnd; i++)
 ...>   ...
 ...>
java> end
```

Using JAR dependencies

To set up dependencies in the environment's class-path, you will use the `/cp` instance command. Say you have a JAR file named `myjar.jar` in your ado-path. You can run the instance command `/cp myjar.jar` to include it in the class-path. After you include it, you may run code that uses that dependency. There is one caveat. If you try to run code that uses the dependency before adding it to the class-path, the class loader will try to load your nonexistent dependency and will require a `/reset` to reload it. Alternatively, you may provide an absolute path or a path relative to your current Stata working directory to search for dependencies.

❑ Technical note

Note that the Stata `version` statement affects only the Stata command interpreter and does not affect the execution or behavior of the Java Virtual Machine.

❑

Also see

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

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

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

[P] **javacall** — Call a Java plugin