NAME

jdb − Java debugger

SYNOPSIS

jdb [*options*] [*class*] [*arguments*]

PARAMETERS

options	Command-line options.
class	Name of the class to begin debugging.
arguments	Arguments passed to the **main( )** method of *class*.

DESCRIPTION

The Java debugger, **jdb**, is a simple command-line debugger for Java classes. It is a demonstration of the **Java Platform Debugger Architecture** that provides inspection and debugging of a local or remote Java Virtual Machine.

Starting a jdb Session

There are many ways to start a jdb session. The most frequently used way is to have **jdb** launch a new Java Virtual Machine (VM) with the main class of the application to be debugged. This is done by substituting the command **jdb** for **java**(1) in the command line. For example, if your application's main class is **MyClass**, you use the following command to debug it under **jdb**:

example% jdb MyClass

When started this way, **jdb** invokes a second Java VM with any specified parameters, loads the specified class, and stops the VM before executing that class's first instruction.

Another way to use **jdb** is by attaching it to a Java VM that is already running. A VM that is to be debugged with **jdb** must be started with the following options:

option	purpose
−Xdebug	Enables debugging support in the VM.
−Xrunjdwp:transport=dt_socket, server=y,suspend=n	Loads in-process debugging libraries and specifies the kind of connection to be made.

For example, the following command will run the **MyClass** application and allow **jdb** to connect to it at a later time:

**example% java −Xdebug **
**−Xrunjdwp:transport=dt_socket,address=8000,server=y,suspend=n **
MyClass

You can then attach **jdb** to the VM with the following commmand:

example% jdb −attach 8000

Note that **MyClass** is not specified in the **jdb** command line in this case because **jdb** is connecting to an existing VM instead of launching a new one.

There are many other ways to connect the debugger to a VM, and all of them are supported by **jdb**. The Java Platform Debugger Architecture has additional documentation on these connection options.

Basic jdb Commands

The following is a list of the basic **jdb** commands. The Java debugger supports other commands listed with the **help** command.

Notice that to display local (stack) variables, the class must have been compiled with the **javac −g** option.

cont Continues execution of the debugged application after a breakpoint, exception, or step.

dump For primitive values, this command is identical to **print**. For objects, it prints the current value of each field defined in the object. Static and instance fields are included.

The **dump** command supports the same set of expressions as the **print** command.

help, or **?** As the most important **jdb** command, **help** displays the list of recognized commands with a brief description.

print Displays Java objects and primitive values. For variables or fields of primitive types, the actual value is printed. For objects, a short description is printed. See the **dump** command for getting more information about an object.

print supports many simple Java expressions including those with method invocations. For example:

- **print MyClass.myStaticField**
- **print myObj.myInstanceField**
- **print** *i* + *j* + *k* ... where *i*, *j*, and *k* are primitives and either fields or local variables.
- **print myObj.myMethod( )** ... if **myMethod** returns a non-null.
- **print new java.lang.String("Hello").length( )**

thread Selects a thread to be the current thread. Many **jdb** commands are based on the setting of the current thread. The thread is specified with the thread index described in the **threads command.**

threads Lists the threads that are currently running. For each thread, its name and current status are printed, as well as an index that can be used for other commands. For example:

4. (java.lang.Thread)0x1 main running

In this example, the thread index is **4**, the thread is an instance of **java.lang.Thread**, the thread name is **main**, and it is currently running

run After starting **jdb**, and setting any necessary breakpoints, use this command to start the execution of the debugged application. This command is available only when **jdb** launches the debugged application (as opposed to attaching to an existing VM).

where The **where** subcommand with no arguments dumps the stack of the current thread (which is set with the **thread** command). Using **where all** dumps the stack of all threads in the current thread group. Using **where** *threadindex* dumps the stack of the specified thread. If the current thread is suspended (either through an event such as a breakpoint or through the **suspend** command), local variables and fields can be displayed with the **print** and **dump** commands. The **up** and **down** commands select which stack frame is current.

Breakpoint Commands

Breakpoints are set in **jdb** at line numbers or at the first instruction of a method. For example:

stop at MyClass:22 Sets a breakpoint at the first instruction for line **22** of the source file containing **MyClass**.

stop in java.lang.String.length Sets a breakpoint at the beginning of the method **java.lang.String.length**.

stop in MyClass.*init* *init* identifies the **MyClass** constructor.

stop in MyClass.*clinit* *clinit* identifies the static initialization code for **MyClass**.

If a method is overloaded, you must also specify its argument types so that the proper method can be selected for a breakpoint. For example,

MyClass.myMethod(int,java.lang.String)

or

MyClass.myMethod()

The **clear** command removes breakpoints using a syntax as in **clear**MyClass:45**.** Using the **clear** command with no argument displays a list of all breakpoints currently set. The **cont** command continues execution.

Stepping Commands

The **step** command advances execution to the next line, whether it is in the current stack frame or a called method. The **next** command advances execution to the next line in the current stack frame.

Exception Commands

When an exception occurs for which there is no catch statement anywhere in the throwing thread's call stack, the VM normally prints an exception trace and exits. When running under **jdb**, however, control returns to **jdb** at the offending throw. Use **jdb** to determine the cause of the exception.

catch Causes the debugged application to stop at other thrown exceptions. For example:

catch java.io.FileNotFoundException

or

catch mypackage.BigTroubleException

Any exception which is an instance of the specified class (or of a subclass) will stop the application at the point where it is thrown.

ignore Negates the effect of a previous **catch** command. Notice that the **ignore** command does not cause the debugged VM to ignore specific exceptions, only the debugger.

OPTIONS

When using **jdb** in place of the Java application launcher on the command line, **jdb** accepts many of the same options as the **java**(1) command, including **−D**, **−classpath**, and **−X***option*.

The following additional options are accepted by **jdb**:

−sourcepath *dir1*:*dir2*:...
Uses the given path in searching for source files in the specified path. If this option is not specified, the default path of "." is used.

−attach *address* Attaches the debugger to previously running VM using the default connection mechanism.

−launch Launches the debugged application immediately upon startup of **jdb**. This option removes the need for using the **run** command. The debuged application is launched and then stopped just before the initial application class is loaded. At that point, you can set any necessary breakpoints and use the **cont** command to continue execution.

−J option Pass *option* to the Java virtual machine, where *option* is one of the options described on the man page for the java application launcher, java(1). For example, *−J-Xms48m* sets the startup memory to 48 megabytes. It is a common convention for **−J** to pass options to the underlying virtual machine.

Other options are supported for alternate mechanisms for connecting the debugger and the VM it is to debug. The Java Platform Debugger Architecture has additional documentation on these connection alternatives.

SEE ALSO

java(1), **javac**(1), **javadoc**(1), **javah**(1), **javap**(1)