NAME

jdb – Finds and fixes bugs in Java platform programs.

SYNOPSIS

jdb [*options*] [*classname*] [*arguments*]

options Command-line options. See Options.

*class*name
Name of the main class to debug.

arguments
Arguments passed to the **main()** method of the class.

DESCRIPTION

The Java Debugger (JDB) is a simple command-line debugger for Java classes. The **jdb** command and its options call the JDB. The **jdb** command demonstrates the Java Platform Debugger Architecture (JDBA) and provides inspection and debugging of a local or remote Java Virtual Machine (JVM). See Java Platform Debugger Architecture (JDBA) at http://docs.oracle.com/javase/8/docs/technotes/guides/jpda/index.html

START A JDB SESSION

There are many ways to start a JDB session. The most frequently used way is to have JDB launch a new JVM with the main class of the application to be debugged. Do this by substituting the **jdb** command for the **java** command in the command line. For example, if your application's main class is **MyClass**, then use the following command to debug it under JDB:

jdb MyClass

When started this way, the **jdb** command calls a second JVM with the specified parameters, loads the specified class, and stops the JVM before executing that class's first instruction.

Another way to use the **jdb** command is by attaching it to a JVM that is already running. Syntax for starting a JVM to which the **jdb** command attaches when the JVM is running is as follows. This loads in-process debugging libraries and specifies the kind of connection to be made.

java –agentlib:jdwp=transport=dt_socket,server=y,suspend=n MyClass

You can then attach the **jdb** command to the JVM with the following command:

jdb –attach 8000

The **MyClass** argument is not specified in the **jdb** command line in this case because the **jdb** command is connecting to an existing JVM instead of launching a new JVM.

There are many other ways to connect the debugger to a JVM, and all of them are supported by the **jdb** command. 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 JDB supports other commands that you can list with the **-help** option.

help or ?
The **help** or **?** commands display the list of recognized commands with a brief description.

run After you start JDB and set breakpoints, you can use the **run** command to execute the debugged application. The **run** command is available only when the **jdb** command starts the debugged application as opposed to attaching to an existing JVM.

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

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 to find out how to get more information about an object.

Note: To display local variables, the containing class must have been compiled with the **javac -g** option.

The **print** command supports many simple Java expressions including those with method invocations, for example:

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

dump For primitive values, the **dump** command is identical to the **print** command. For objects, the **dump** command 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.

threads List the threads that are currently running. For each thread, its name and current status are printed and an index that can be used in other commands. 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.

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

thread Select 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.

where The **where** command with no arguments dumps the stack of the current thread. The **whereall** command dumps the stack of all threads in the current thread group. The **wherethreadindex** command 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, then local variables and fields can be displayed with the **print** and **dump** commands. The **up** and **down** commands select which stack frame is the current stack frame.

BREAKPOINTS

Breakpoints can be set in JDB at line numbers or at the first instruction of a method, for example:

- The command **stop at MyClass:22** sets a breakpoint at the first instruction for line 22 of the source file containing **MyClass**.
- The command **stop in java.lang.String.length** sets a breakpoint at the beginning of the method **java.lang.String.length**.
- The command **stop in MyClass.<clinit>** uses **<clinit>** to identify the static initialization code for **MyClass**.

When 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 the following syntax: **clear MyClass:45**. Using the **clear** or **stop** command with no argument displays a list of all breakpoints currently set. The **cont** command

continues execution.

STEPPING

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.

EXCEPTIONS

When an exception occurs for which there is not a **catch** statement anywhere in the throwing thread's call stack, the JVM typically prints an exception trace and exits. When running under JDB, however, control returns to JDB at the offending throw. You can then use the **jdb** command to diagnose the cause of the exception.

Use the **catch** command to cause the debugged application to stop at other thrown exceptions, for example: **catch java.io.FileNotFoundException** or **catchmypackage.BigTroubleException**. Any exception that is an instance of the specified class or subclass stops the application at the point where it is thrown.

The **ignore** command negates the effect of an earlier **catch** command. The **ignore** command does not cause the debugged JVM to ignore specific exceptions, but only to ignore the debugger.

OPTIONS

When you use the **jdb** command instead of the **java** command on the command line, the **jdb** command accepts many of the same options as the **java** command, including **-D**, **-classpath**, and **-X** options. The following list contains additional options that are accepted by the **jdb** command.

Other options are supported to provide alternate mechanisms for connecting the debugger to the JVM it is to debug. For additional documentation about these connection alternatives, see Java Platform Debugger Architecture (JPDA) at http://docs.oracle.com/javase/8/docs/technotes/guides/jpda/index.html

-help
: Displays a help message.

-sourcepath *dir1:dir2: . . .*
: Uses the specified path to search for source files in the specified path. If this option is not specified, then use the default path of dot (.).

-attach *address*
: Attaches the debugger to a running JVM with the default connection mechanism.

-listen *address*
: Waits for a running JVM to connect to the specified address with a standard connector.

-launch
: Starts the debugged application immediately upon startup of JDB. The **-launch** option removes the need for the **run** command. The debugged 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.

-listconnectors
: List the connectors available in this JVM.

-connect connector-name:*name1=value1*
: Connects to the target JVM with the named connector and listed argument values.

-dbgtrace [*flags*]
: Prints information for debugging the **jdb** command.

-tclient
: Runs the application in the Java HotSpot VM client.

-tserver
: Runs the application in the Java HotSpot VM server.

-J*option*
: Passes **option** to the JVM, where option is one of the options described on the reference page for the Java application launcher. For example, **-J-Xms48m** sets the startup memory to 48 MB. See

java(1).

OPTIONS FORWARDED TO THE DEBUGGER PROCESS

-v -verbose[:*class*|gc|jni]
: Turns on verbose mode.

-D*name*=*value*
: Sets a system property.

-classpath *dir*
: Lists directories separated by colons in which to look for classes.

-X*option*
: Nonstandard target JVM option.

SEE ALSO

- javac(1)
- java(1)
- javap(1)