jdb: Java Debugger

The `jdb` tool helps you to find and fix errors in Java programs. For more information on `jdb` tool, see the **Oracle Java documentation for jdb**.

Synopsis

```
jdb [ options ] [ class ] [ arguments ]
```

`options`

See **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 an example of the use of the **Java Platform Debugger Architecture** that provides inspection and debugging of a local or remote Java virtual machine (VM).

Starting a `jdb` Session

There are many ways to start a `jdb` session. The most frequent way is to have `jdb` launch a new Java VM with the main class of application to be debugged. Perform this by substituting the command `jdb` for the command `java` in the command line. For example, if your application's main class is named `MyClass`, you use the following command to debug it under `JDB`:

```
jdb MyClass
```

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

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

Option	Purpose
`-Xdebug`	Enables debugging support in the Java VM.
`-agentlib:jdwp=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 runs the `MyClass` application and allows `jdb` to connect to the application at a later time.

```
java -Xdebug -agentlib:jdwp=transport=dt_socket,\
address=8000,server=y,suspend=n MyClass
```

You can then attach `jdb` to the Java VM with the following command:

```
jdb -attach 8000
```

NOTE:

`MyClass` is not specified in the `jdb` command line in this case because `jdb` connects to an existing Java VM instead of launching a new one.

There are many other ways to connect the debugger to a Java VM, and all of them are supported by jdb, as specified in **Connecting for Remote Debugging**.

Basic `jdb` Commands

The following is a list of the basic `jdb` commands. The Java debugger supports other commands, which you can list by using the `jdb help` command.

`{help | ?}`

Displays the list of recognized commands with a brief description.

`run`

After starting jdb and setting any necessary breakpoints, you can 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 Java VM).

`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 below for getting more information about an object.

NOTE:

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

`print` 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 primitives and either fields or local variables)`
- `print myObj.myMethod() (to print the value if myMethod() returns a non-null)`
- `print new java.lang.String("Hello").length()`

`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.

`thread`

List 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:

```
(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.

`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`

`where` with no arguments dumps the stack of the current thread. `where all` dumps the stack of all threads in the current thread group. `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.

Breakpoints

Breakpoints can be 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 by 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

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 Java VM normally prints an exception trace and exits. When running under `jdb`, however, control returns to `jdb` at the offending throw. You can then use `jdb` 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 `catch mypackage.BigTroubleException`. Any exception that is an instance of the specified class (or of a subclass) stops the application at the point where it is thrown.

The `ignore` command negates the effect of a previous catch command.

NOTE:

The `ignore` does not cause the debugged VM to ignore specific exceptions, only the debugger.

Command Line Options

When you use `jdb` in place of the Java application launcher on the command line, `jdb` accepts many of the same options as the **java: Java Application Launcher** command, including `-D`, `-classpath`, and `-Xoption`.

The following additional options are accepted by jdb:

`-help`

Displays a help message.

`-sourcepath directory1 [:directory2]...`

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 the previously running Java VM by using the default connection mechanism.

`-listen address`

Waits for a running VM to connect to the specified address through a standard connector.

`-listenany`

Waits for a running VM to connect to any available address through a standard connector.

`-launch`

Launches the debugged application immediately upon startup of jdb. This option removes the need for using 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` to continue execution.

`-connect connector-name:name1=value1,...`

Connects to the target VM through a named connector that uses the listed argument values.

`-dbgtrace [flags]`

Prints information for debugging jdb.

`-Joption`

Pass `option` to the Java virtual machine, 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 megabytes.

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

Deviations from Standard Java

`-tclient`

Runs the application in the Java HotSpot client VM.

NOTE:

The `-tclient` option is not valid with NonStop Server for Java 8.0.

`-tserv`

Runs the application in the Java HotSpot server VM.

NOTE:

`-tserv` is the default option for NonStop Server for Java 8.0; therefore, specifying `-tserv` is optional.

Options Forwarded to the Process Being Debugged

`-v -verbose[:class|gc|nji]`

Turns on verbose mode.

`-D name=value`

Sets a system property.

`-classpath directory1 [:directory2]...`

Lists directories in which to look for classes.

`-X option`

Sets a nonstandard target VM option.

Connecting for Remote Debugging

1. The Debugger launches the target Java VM.

   `-launch`

   `jdb -launch ClassName`
2. The Debugger attaches to a previously running Java VM.

   `-attach`

   `jdb -attach hostname:portnum`

   For this command, the JVM must already be running as a server at `[<hostname>:]<portnum>|<start port>-<end port>`

   To start the server, use the following command :

   ```
   java -Xnoagent -Xdebug -Djava.complier=NONE \ -
   agentlib:jdwp=transport=dt_socket,\address=[<hostname>:]<portnum>|<start
   port>-<end port>,server=y \ ClassName
   ```

```
If address option is not given, the server will start on any available port
on the local host and print portnum. This portnum should be used by the jdb
to attach.
```

NOTE:

In NonStop, there is an additional option to specify the port range, where, <start port> and <end port> are the starting and ending port numbers for a range of ports.

3. The target JVM attaches to previously running debugger.

   `-listen`

   `jdb -listen hostname:portnum`

   To attach a target JVM, use the following command:

   ```
   java -Xnoagent -Xdebug -Djava.complier=NONE \ -
   agentlib:jdwp=transport=dt_socket, address=hostname:portnum \ ClassName
   ```
4. The Debugger selects a connector.

   `-connect`

   `jdb -connect option`

   NOTE:

   Only the `com.sun.jdi.SocketListen` option is supported.

   The target Java VM can then attach as:

   ```
   java -Xnoagent -Xdebug -Djava.compiler=NONE \ -
   agentlib:jdwp=transport=dt_socket, address=hostname:portnum \ ClassName
   ```

Transports

A Java Platform Debugger Architecture (JPDA) transport is a form of inter-process communication used by a debugger application and the debuggee. NonStop Server for Java 8.0 provides a socket transport that uses the standard TCP/IP sockets to communicate between debugger and the debuggee.

NonStop Server for Java 8.0 defaults to socket transport. NonStop Server for Java 8.0 does not support shared memory transport.

See Also:

- **javac**
- **java**
- **javadoc**
- **javah**
- **javap**