Technote 1154

Debugging Java Code With MacsBug

CONTENTS

MacsBug, the low-level debugger for the Mac OS, seems unlikely to be useful for debugging a very high-level language like Java. *Au contraire!* The MRJ plug-in `'dcmd'` for MacsBug adds a number of commands that can help you debug everything from deadlocks to memory leaks.

MacsBug In a Nutshell

Before you can start using MacsBug to debug Java, you need to install MacsBug and learn the basics of how to use it. If you've already been developing Mac software, this is a non-issue since you're almost certainly already familiar with MacsBug, and you can skip to the next section.

However, there are a lot of people developing or testing Java on the Macintosh who are not otherwise Mac developers and don't know an A-trap from "Take the A Train." This section is for them, so it presumes a lot less Mac technical knowledge than most technotes do.

What's MacsBug?

MacsBug is Apple's assembly-level 680x0 and PowerPC debugger for Mac OS. It can be used to debug code running in most execution environments, from applications to drivers, and everything in between. It's often used as a bug-reporting tool by many third-party developers, as well as Mac OS system software developers.

MacsBug knows nothing about source code, only assembly-language instructions, and its support for high-level data structures is primitive. But it's great for examining the exact machine state.

Unlike debuggers on most other platforms, MacsBug is always resident once installed, and can take over instantly when a crash occurs or when you press a hot-key, even if the machine is otherwise frozen.

Download & Installation

MacsBug is available from Apple's developer Web site. As befits a tweaky developer tool, it spends most of its time in one prerelease state or another, and the latest and greatest version is nearly always marked as alpha or beta. Nonetheless, it's still usually best to install the latest prerelease; they've proven to be pretty stable. (Note in particular that as of this writing, only prerelease versions are compatible with OS

8.5.)

There are a lot of files in the download, but the only one you need is MacsBug itself, which you'll find in the "into System folder" folder. Drag it into your System folder (*not* the Extensions folder!) and restart. You should see the message "Debugger Installed" appear below "Welcome To Mac OS" in the Mac OS splash screen as the system begins to boot.

A Very Brief Overview

The MacsBug manual is also available online, but it's large and intimidating (not to mention slightly obsolete) and actually overkill for the Java-only developer. Here's a very brief introduction:

MacsBug loads itself into memory very early in the boot process and hides out invisibly until it's needed. (It does consume a bit of RAM.) Three different circumstances will cause MacsBug to appear:

1. A CPU exception or a system error occurs -- both are usually referred to as "crashes." If these occur and MacsBug is not installed, then depending on the severity, you get either an "Unexpectedly quit" alert or a dread bomb box.
2. The system routines `Debugger` or `DebugStr` are called. MacsBug will report a "user break," usually with a message. This lets software report messages to the user -- usually warnings of dangerous or unexpected situations. Only special debug versions of software should contain user breaks.
3. You explicitly invoke MacsBug by holding down the Command ("cloverleaf") key and pressing the Power key (the one with the triangle that you use to turn your system on). You can do this even if the computer is otherwise hung or frozen; if it doesn't work, things are in really bad shape and your only option is to force a restart by pressing Command-Control-Power, the dread "three-finger salute."

When MacsBug appears, it completely takes over the screen and the CPU. No other application or OS software can run while MacsBug is visible. This explains why MacsBug's user interface is so completely non-Mac-like -- it can't use any of the Toolbox.

MacsBug is a command-line environment like DOS or a Unix shell. It shows one fixed-size window in the middle of the screen, with garbage pixels outside the window. There's an input line at the bottom, a few lines of machine code disassembly above it, and a large scrolling output area above that. On the left side is a register and stack display.

You type commands into the input line at the bottom and press Return to run them. You can't use the mouse to select text, but most standard editing keystrokes work (arrow keys, delete and forward-delete, option- and Command- arrow, etc.) You can also press Cmd-V to copy the previous command into the input line.

Some essential commands

The most vital MacsBug commands are:

- **`help`** -- Displays a list of help topics. You can get more info by entering "help" followed by the name of a topic or command.
- **`g`** -- "Go." Attempts to resume normal operation. This will only work if you entered MacsBug on purpose via Command-Power or after a user break: if the system is crashed, you can't continue normally.
- **`es`** -- "Exit to Shell." Attempts to force the currently active application to quit. This is just like the Command-Option-Escape panic button that non-MacsBug-users use. This may or may not succeed, depending on how damaged the system is and exactly where you were at the point you entered MacsBug. If it succeeds, restart ASAP. If not, use...
- **`rs`** -- "ReStart." Attempts to restart/reboot the system after some clean-up (for instance, flushing the disk cache). This is friendlier than Command-Control-Power or pulling out the power cord, and less likely to cause disk damage.
- **`stdlog`** -- dumps a text file to your desktop, containing a lot of information about the current

machine state. We ask you to submit one of these when reporting any crash or user-break involving MRJ.

- **`log`** -- If you put a filename after the log command, all subsequent MacsBug output will be written to that file. (The file usually appears in the same folder as the current application.) "log" with no filename turns off logging.

Back to top

The 'mrj' dcmd

We've written a plug-in (called a "dcmd") for MacsBug that adds a new command, **`mrj`**. Actually it adds many new commands, invoked by name on the command line after the "`mrj`" part. For instance, "`mrj sc`" dumps a Java stack crawl.

The `dcmd` is available as part of the MRJ SDK, in the folder "`Tools: Other Tools: MRJ dcmd for PPC:`". Just put the file "`MRJ dcmd`" into the `MacsBug Preferences` folder of the `Preferences` folder of your System folder, and restart.

To use the `dcmd`, you must have a Java application or applet running. Most of the `dcmd`'s commands are not available, and make no sense, otherwise.

Important:
Almost all of the commands described below work only with MRJ 2.1. MRJ 2.0's SDK included an earlier version of the `dcmd` with only two or three commands. Make sure you install the `dcmd` from the MRJ 2.1 SDK.

Disclaimer:
This technote describes MRJ 2.1. It is quite likely that the `dcmd`'s feature set and details of its commands may change in the future as the JVM is improved. Some commands or features of commands may not be available depending on how the corresponding area of the JVM is reimplemented.

Back to top

Basic Commands

Here's information about the most commonly used commands. (Many of the other commands are for querying internal data structures and are only of use for debugging MRJ itself.)

If you invoke the `dcmd` without any command (by typing just "`mrj`") it will display a brief list of commands:

Java log (`mrjlog`)

This is the Java equivalent of `stdlog` -- it writes a text file to your desktop that contains a lot of information about the current Java state. **Note** that there is no space in "`mrjlog`"! This is really a macro that opens a log file and runs several `mrj dcmd` commands that write to it.

Stack crawl (`mrj sc`)

Displays information about the current thread and its Java stack, with the current stack frame listed first. The details are described below.

Thread dump (`mrj threads`)

Displays information about all active threads, including their stacks. Each thread also tells what monitors it's acquired (what object it's currently synchronized against, in Java parlance) and at the end of the

listing is a dump of the cache of the most recently used monitors and who owns them or is waiting for them.

Deadlock detection (`mrj dl`)

Looks for a classic deadlock -- two threads, each one blocked waiting for a monitor owned by the other thread. If it finds one, it will list information about the threads and the monitors.

Synchronization check (`mrj sync`)

Looks for other possible synchronization problems besides deadlocks, and displays information about them if it finds any. These include:

- Thread holding any monitor (synchronized against anything) while blocked in `Object.wait`
- Thread blocked on a monitor while holding another monitor
- Thread suspended (via `Thread.suspend`) while holding a monitor

These are all technically legal situations but can often lead to deadlock and are thus suspicious if they turn up.

Redirect output (`mrj redirect` *`filename`*)

Redirects `System.out` and `System.err` to the given filename, which should be a full path. If no filename is given, output is disabled entirely.

The redirection does not take effect immediately -- some Java code needs to be called, and the JVM is usually not in the right state to do this at the moment that MacsBug was invoked, so the request is queued until the next time the main thread runs.

Java heap usage (`mrj chunks`)

Displays the total, used and free space in the Java object heap, and in the handle heap which is associated with it. Also displays info about each memory “chunk” in the heap, which is generally not of interest.

MRJ non-Java heap usage (`mrj alloch`)

Displays the amount of memory used by MRJ for things other than objects. These include data loaded from `.class` files and internal data structures associated with classes, threads and other things.

Find monitor (`mrj fm` *`monitor`*)

Given a monitor ID (as reported as the “`mon:`” value in an `mrj sc` or `mrj threads` dump) this locates the object the monitor is associated with, giving its classname and handle. You can then use the handle with object inspection commands like `mrj do` to get more information about the object.

Execute a method (`mrj exec` *`classname methodname`*)

Waits for the MRJ main thread to get control, then calls a single parameterless static void method. The classname needs to be fully qualified with package names separated by “`/`”s. The method name is separated by a space, not a “`.`”!

This isn’t too useful out of the box, but you can turn it into a powerful debugging tool by adding such methods to your app. For instance, you could add a `Debug` class in the default/root package, and give it static void methods like `dumpState` or `startLogging`. Then at any moment you can press Command-Power to enter MacsBug, type “`mrj exec Debug dumpState`”, and get all kinds of useful info printed to the console.

Find references (`mrj fr` *`handle`*)

Searches moderately hard for things that point to the given object. This can be useful if you're trying to figure out why objects aren't being garbage collected. I say "moderately hard" because this command searches the object heap and JNI global references, but not thread stacks, so it may not find all references.

The references will be listed one per line and described as "instance field," "array element," etc. The handle of the object containing the reference will be given, which you can use as the argument to another `mrj fr` command if you want to trace further.

A more sophisticated reference-finder is `mrj graphrefs`, but it's only available in the debug build.

Instruction listing (`mrj il` *`methodname`*)

Disassembles the bytecodes of a method. The method has to be named in the usual internal format:

- Fully qualified classname with packages separated by "`/`"s
- "`.`",
- method name
- "`.`"
- Encoded parameter types in parentheses
- Encoded return type

For example:

```
il java/lang/Thread.join.()V
      disassembly from $659ed84 java.lang.Thread.join(Thread.java:873)
          [0]  2A aload_0
          [1]  9 lconst_0
          [2]  E2 invokevirtual_quick_w Method: "java/lang/Thread.join(J)V"
          [5]  B1 return
```

Back to top

Commands Available Only In the Debug Build

The debug build of MRJ enables extra commands in the MRJ `dcmd`. (They're not in the regular build because supporting them makes MRJ slower or use more memory or both.)

The debug build also has a limited form of deadlock-checking built into the thread scheduler: in the case of a classic two-thread deadlock it will automatically drop into MacsBug with a user break warning about a deadlock. You should immediately use "`mrj dl`" to get more information.

Another handy feature of the debug build is that it will display a cursor shaped like a little bulldozer while it garbage-collects. This can help you tell whether long pauses in your app are actually caused by garbage collection (as can the `mrj tracegc` command described below.)

Count class instances (`mrj extant`)

Lists every class currently loaded, in alphabetical order, plus the current and maximum number of instances. If you add a numeric argument to this command, it displays only classes with at least that many current instances.

Trace object allocation (`mrj tracealloc` *`value`*)

A *value* of *1* turns on tracing of object allocations, *0* turns it off. This writes a line of output to the console whenever an object is allocated, giving the name of the object's class. This can be very verbose (it's scary to see just how many `String` and `StringBuffer` objects the most seemingly simple code can generate!) but can also be very useful for checking how efficient your code is at using objects.

Trace garbage collection (`mrj tracegc` *`value`*)

A *value* of *1* turns on tracing of garbage collection, *0* turns it off. This will write a bunch of detailed info about garbage collection whenever it occurs. Almost none of this information will be of any use to you, but it can be helpful to see visual evidence that garbage collection is occurring.

Method tracing (`mrj tracem` *`value`*)

A *value* of *1* turns on method tracing, *0* turns it off. Method tracing writes a line of output to the console whenever any method is entered or exited. This results in reams of output -- you should first use `mrj redirect` to write to a file, not the console window! -- but can be quite useful for examining the flow of execution without stopping the program or when no high-level debugger is available.

Instruction tracing (`mrj trace` *`value`*)

A *value* of *1* turns on instruction tracing, *0* turns it off. Instruction tracing writes a line of output to the console for every Java bytecode instruction that's executed. This is very rarely useful, and produces staggering amounts of output -- use `mrj redirect` first to write to a file. It also has no effect when running JITted code, so you probably want to disable the JIT before launching MRJ if you plan on using this.

In MRJ 2.1 not all bytecodes executed get displayed. We plan to fix this in the next release.

Graph references (`mrj graphrefs` *`handle`*)

A more involved reference tracker than `mrj fr`, this transitively searches for chains of references from roots (like static variables) that point to the given object and thus keep it from being garbage collected.

Warning:
This command was written in a hurry (Steve Zellers needed to squash some memory leaks) and has been only minimally tested in MRJ 2.1. It does not work in Apple Applet Runner, and (ironically) can leak memory into the application heap.

This command waits for the main MRJ thread to get control, does its work,
then writes the results to `System.out`. Here's the beginning of some typical output:

```
mrj graphrefs $6b11f18
          recursively searching for references to $6b11f18
          References to: $6b11f18
          instance field: $6b11aa8 java/lang/
  Thread.target(Ljava/lang/Runnable;)
          instance field: $6b126c8 com/apple/mrj/console/Console$ConsoleArea.this$0(Lcom/
  apple/mrj/console/Console;)
          instance field: $6b13628 com/apple/mrj/
  console/Console$1.this$0(Lcom/apple/mrj/console/Console;)
          java thread var ref $6b11f18 at $x (tid $68ef584, )

        References to: $6b11aa8
        array element: $6b11ea8 [1]
        c thread found $6b11aa8 at $680f2fc (tid $6adda7c, ConsoleThread)
        c thread found $6b11aa8 at $680f34c (tid $6adda7c, ConsoleThread)
        c thread found $6b11aa8 at $680f370 (tid $6adda7c, ConsoleThread)
        c thread found $6b11aa8 at $680f3e8 (tid $6adda7c, ConsoleThread)
        java thread var ref $6b11aa8 at $x (tid $68ef558, )
```

Traces are separated by blank lines. Each trace starts with "`References to:`" followed by the handle of the object it's tracing, and then lists all references to that object, such as instance variables in other objects, static variables, and local variables of current stack frames.

The first trace is for the object you requested. Subsequent traces are for objects found in previous traces. The result, if you follow from one trace to another, lets you find out exactly what chains of references are keeping an object from being garbage collected.

This output is pretty hard to read and cries out for a nice tool to interpret it. For now all we can suggest is pasting the output into a good programmer's editor and using the Find command to find matching hex values.

Back to top

Inspecting Objects

You can examine the fields of Java objects and the elements of arrays if you know the object/array's *handle* . This is a 32-bit object ID. There are three ways to find a handle:

1. Many `dcmd` commands display object handles. For instance, `mrj sc` displays the handle of the `Thread` and of the receiver ("`this`") of every stack frame.
2. The method `Object.hashCode` happens to return the object's handle shifted right 3 bits. So to print foo's handle to the console you can use:
   `System.out.println(Integer.toHexString(foo.hashCode()<<3));`
   This will not work if the object's class overrides `hashCode` to return a different value! So it's useless on `Strings` and `Points`, for instance. But for most classes you can use this in your logging code to dump the handles of useful objects to the console.
   This behavior may obviously change in the future if we re-implement the JVM.

Display Object (`mrj do` *`handle`*)

Displays the object with the given *handle* . The results might look like:

```
java.lang.ThreadGroup@5D590C0/5D0A668
     SuperName: java/lang/Object
     # ClassName: java/lang/ThreadGroup
         parent.(Ljava/lang/ThreadGroup;) = $05d540a0
         name.(Ljava/lang/String;) = $05d590d0
         maxPriority.(I) = 10
         destroyed.(Z) = false
         daemon.(Z) = false
         vmAllowSuspension.(Z) = false
         nthreads.(I) = 3
         threads.([Ljava/lang/Thread;) = $05d59090
         ngroups.(I) = 1
         groups.([Ljava/lang/ThreadGroup;) = $05d5a948
     # ClassName: java/lang/Object
```

The first line shows the object's class-name and handle. (The second number after the "`/`" is not useful.) The second line shows the name of the superclass.

After that, follow blocks for the object's class and each superclass. Each block starts with the classname and then shows all variables declared in that class and their values for that object. (Static variables have "`[static]`" at the end.)

Each variable entry shows its name, then its type in parentheses. The type follows the typical encoding scheme used by the JVM: Single letters for primitive types (`I` for int, `Z` for boolean, etc.) and for object types, "`L`" followed by the classname followed by "`;`".

If a variable's type is an object, the value shown is the object's handle, so you can use a further `mrj do` command to inspect *that* object.

Display array (`mrj da` *`handle`*)

Displays the contents of the array with the given *handle* . The first line shows the type of the array elements and the length of the array; then, each element is listed on a separate line.

To list only a portion of an array, you can provide two extra parameters that specify the first item to display and the item *after* the last one to display; for example:

```
mrj da 05d5a948 1 3
                          java.lang.ThreadGroup[4]
         1: $00000000 -> NULL
         2: $00000000 -> NULL
```

Display string (`mrj ds` *`handle`*)

A convenient way to display the contents of a String object. (You could use `do` and `da` to find and dump the `char[]` array in the String, but it's awkward.)

This command doesn't do very well with non-ASCII characters, since that would require higher level translation services that aren't available from within MacsBug.

Find a class (`mrj fc` *`name`*)

Locates a loaded class with the given *name* . You need to specify the complete name including packages, and package names need to be separated with "`/`" instead of "`.`"; for historical reasons, that's the way classnames are represented internally.

If the class is found, the command will return the handle of its `Class` object. That object usually isn't very useful, but this command can still be handy to determine whether or not a particular class has been

loaded yet.

Dump class methods (`mrj dcm classname`)

Displays all the methods of a given class (including inherited ones.) The classname has to be fully specified, with "/"s, as described above for `mrj fc`. The output format is similar to that of `mrj do`:

There's a section for the class and each superclass. Each section shows the classname, then each method introduced by that class. Each method is shown on a line containing:

1. The method's name.
2. The method's signature. The parameter types are shown in parentheses, then the return type. The types follow the typical encoding scheme used by the JVM: Single letters for primitive types (`I` for `int`, `V` for void, `Z` for `boolean`, etc.) and for object types, "`L`" followed by the classname followed by "`;`".
3. Modifiers like `static` and `synchronized`.

Dump object methods (`mrj dom handle`)

Similar to `mrj dcm`, but dumps the methods of the class of the object whose *handle* is given.

Back to top

Interpreting Stack Crawls and Thread Dumps

The **`mrj sc`** and **`mrj threads`** commands both display stack crawls, and there's a lot of cryptic but useful information packed into them. A typical stack crawl looks like:

```
"QDPipeline"
    TID: $60a1bf8, prio: 5
    sys_thread: $5fe3200, priority: 5, saved_sp: $5fc5980
    state: WCV, mon: $727ec24, cq: $727ec30
 $60a1c20 -> java.lang.Object.wait(Object.java:315)
 $60a1c20 -> com.apple.mrj.internal.awt.QDPipeline.run(QDPipeline.java:289)
 $60a1bf8 -> java.lang.Thread.run(Thread.java:474)
```

The thread header

The first four lines display information about the thread:

"The thread's name" In quotes on the first line. This is the String parameter passed to the thread's constructor. If you don't provide one, you get a default name like "Thread-7." Giving your threads meaningful names is quite useful when debugging.

`TID:` This is the handle of the Thread object. You can use this with object-inspection commands like `mrj do`.

`prio:` The regular Java thread priority, from 1 to 10. You might notice that the "main" thread has priority 11 -- this is impossible to do programmatically, but we can set it that way because we wrote the JVM. The main thread needs to be able to pre-empt any other thread when a `JManager` call comes in.

`sys_thread:` Points to the native thread structure. You can use the `mrj thd` command to display lots of cryptic information about it.

`saved_sp:` or `*current thread*:` The `saved_sp` points to the thread's *native* stack; use it as the argument to MacsBug's regular stack crawl command `sc7` to display the stack. (But `sc7` stack crawls aren't very accurate and tend to display a lot of junk.) If you see `*current thread*` instead, that means

that this is the thread currently running; use `sc` or `sc7` with no arguments to see its native stack.

`priority:` For historical reasons, this is a duplicate of the `prio:` field on the previous line.

`state:` The thread's current state. **`RDY`** means "ready": the thread is runnable, and is either running now or will run when the scheduler gives it a chance. (It might still never get a chance to run if higher priority threads are always busy.) **`WMON`** means "waiting on monitor": this usually means that the thread is blocked entering a `synchronized` method or statement because another thread is already synchronized against that object. (Unfortunately, due to a bug in the `dcmd`, a runnable thread is sometimes incorrectly listed as `WMON`.) **`WCV`** means "waiting on condition variable": the thread is blocked in the `Object.wait` method and hasn't yet been woken up by an `Object.notify` or `notifyAll` call. **`SUSP`** means the thread has been suspended via `Thread.suspend`.

`mon:` If the thread's state is `WMON`, this field shows the ID of the monitor it's blocked on. Monitors are usually associated with objects, but a monitor ID is *not* an object handle, and there are monitors that don't correspond to objects. You can use the `mrj fm` command to find which Java object owns that monitor, or possibly the `mrj mon` command to display cryptic information about the monitor itself.

`cq:` If the thread's state is `WCV`, this field shows the ID of the condition queue it's waiting on. This is an internal data structure with no user-serviceable parts inside.

The stack crawl itself

After the thread header comes the Java stack crawl. This is mostly identical to the kind of stack crawl you're used to seeing when an exception is dumped to the console.

The stack frames are listed in reverse chronological order, so the current method is at the top.

There's an additional hex number at the beginning of each line, which is the object handle of the "`this`" variable (or receiver) of the method. You can use this in conjunction with object inspection commands like `mrj do`.

After the object handle comes the name of the method. After that in parentheses is the name of the source file and the exact line number.

Note:
The source file and line number will not be displayed if the line-number mapping table is not found in the class file. (If you compiled your code with Metrowerks CodeWarrior, make sure the debugging dot is turned on next to the source file in the project window.)

The source/line information will also be replaced with "(Compiled Code)" if the method has been translated into native code by the JIT: the JIT isn't able to perform the (very difficult) reverse mapping of native instructions to Java bytecodes. You can prevent this by disabling the JIT: remove the "MRJ PPC JITC" library from the MRJ Libraries folder.

The Monitor Cache Dump

Monitors are the primitives used to implement synchronization on objects. Monitors are not objects, but an object is assigned a monitor when a thread synchronizes against it. (There are also special internal monitors that are not associated with objects.)

As described above, the header of a stack crawl tells whether the thread is blocked on a monitor and, if so, which one. The thing you probably want to know next is what object that monitor corresponds to. Usually (not always) you can determine this by looking in the Monitor Cache Dump section in the "`mrj threads`" dump. This section comes right after the last thread's stack crawl, and lists the objects that have most recently acquired monitors. A typical entry looks like this:

```
"com.apple.mrj.JManager.AVDispatcherThread@60A09C8/618AD48"
   <unowned>
  Waiting to be notified:
     "AVGrp-com.apple.mrj.JManager.JMAWTContextImpl@c6ce6f-Disp" prio 4
```

The first line shows the object's class. The hex number between the "`@`" and the "`/`" is the object's handle, which lets you inspect the object via commands like `mrj do`.

The second line shows which thread owns (is currently holding) that monitor, or `<unowned>` if no thread owns it.

If one or more other threads are blocked on that monitor, they will be listed after a line reading "`Waiting to be notified:`". Each thread is listed by name, followed by "`@`", followed by the handle of the thread object and its priority. You can of course find more info about the thread in its stack crawl above.

It's worth pointing out two Java objects that often play a prominent role in synchronization problems. `com.apple.mrj.macos.toolbox.Toolbox` is the *Toolbox lock*, which is used by the AWT peers and other native or `JDirect` code to synchronize access to the Mac Toolbox. (It's described in much more detail in Technote 1153, *Thread-Safe Toolbox Access From MRJ*. And if you see a `java.lang.Object` in the monitor cache, it's probably the `treeLock` used by the public AWT classes (it's declared in `java.awt.Component`) to synchronize access to the component hierarchy. AWT-related deadlocks often involve one or both of these.

The Registered Monitor Dump

The last section of the thread dump shows the list of registered monitors. These are monitors that are not associated with objects but which are known to the JVM. These are used internally by things like the JIT, the class loader, and the finalizer thread. Normally you don't need to pay attention to these, but very rarely you may encounter a deadlock that involves one or them (for instance, we once had a nasty bug that could cause the JIT and the class loader to deadlock). If you encounter any problems involving these, it's almost certainly a bug in MRJ, which you should report at once, including a `stdlog` and an `mrjlog`.

Back to top

Switching Contexts

If only a single instance of MRJ is running, the `mrj dcmd` will automatically target it; it doesn't matter which application is active at the moment MacsBug was entered. But if you have two Java apps running at the same time, you'll need to disambiguate them.

To target a particular instance of MRJ, you need to know its CFM *context ID*. You can find this by using the MacsBug "`frags`" command. The output shows a list of applications, and for each app the list of libraries it's instantiated. The header line for an app shows its context ID (with a "#" sign prepended to show that it's in decimal -- don't forget to include the "#" sign when typing in the value.)

Switch contexts (`mrj prf` *`context`*)

Targets the instance of MRJ running in the given *context*. Subsequent commands will apply to this instance of MRJ until you target a different one.

Tell context (`mrj pr`)

Indicates which CFM context is targeted, if any.

Back to top

Further References

- Apple Computer. *MacsBug Reference And Debugging Guide* . 1995. The complete guide to MacsBug -- not very tutorial-like, unfortunately, and certainly overkill unless you plan on debugging or testing a lot of native code.
- Lindholm, Tim. *The Java Virtual Machine Specification.* Addison-Wesley, 1996. The exact specification for how the JVM operates. If you want to know the exact details of how thread behavior is specified, look here. (It does not discuss implementation specific details of object layout or thread scheduling, though.)
- The MRJ 2.1.1 Debug release.

Downloadables

Acrobat version of this Technote (K).

Back to top

To contact us, please use the Contact Us page.
Updated: 07-April-99