UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS61B **P. N. Hilfinger**
Spring 1998

The GJDB Debugger

Sun Microsystems, Inc. distributes a rather pitiful text-based debugger, called JDB, with its Java Developer's Kit (JDK)[1]. I have modified JDB to make its commands look pretty much like GDB, and (frankly) to make it work at all. Here, I will assume that you have read *Simple Use of GDB* in this reader. This present document will describe differences between GDB and our Java debugger, which is called GJDB. As with GDB, I really suggest that you run the debugger only under Emacs (with the `M-x gjdb` command).

Starting GJDB

When compiling your Java programs, use the `-g` switch to `javac` to indicate that you want debugging information included, as in

```
javac -g Hello.java Greeting.java
```

Compiling a Java program produces a set of `.class` files, one of which contains your main function as a static member. If the class `Hello` contains this function, then the command

```
gjdb Hello
```

starts the debugger and prepares it to execute `Hello.main`. As with GDB, however, the program will not immediately start executing. Instead, you will see the prompt

```
(gjdb)
```

This provides a clumsy but effective text interface to the debugger. I don't actually recommend that you do this; it's much better to use the Emacs facilities described below. However, the text interface will do for describing the commands.

[1]Given that the JDK is a Java interpreter (for which it is usually pretty easy to build a debugger), Sun's inattention to this tool is particularly odd. I can only presume that they are trying not to undermine the market for tools that they sell for money.

GJDB commands

Whenever the command prompt appears, you have available the following commands. Actually, you can abbreviate most of them with a sufficiently long prefix. For example, `p` is short for `print`, and `b` is short for `break`.

`help` *command*
: Provide a brief description of a GJDB command or topic. Plain `help` lists the commands.

`run` *command-line-arguments*
: Starts your program as if you had typed

```
java Hello {\it command-line-arguments}
```

to a Unix shell. In contrast to GDB, you can only execute this command once per session.

`where`
: Produce a backtrace—the chain of function calls that brought the program to its current place.

`up`
: Move the current frame that GDB is examining to the caller of that frame. Emacs (see below) provides the shorthand `C-c<` (Control-C followed by less-than).

`down`
: Undoes the effect of `up`. Emacs provides the shorthand `C-c>`.

`print` E
: prints the value of E in the current frame in the program, where E is a simple variable name. For objects (that is, variables of reference type), this prints the string returned by the `.toString()` member function, which may or may not be particularly informative.

`dump` E
: also prints the value of simple variable E in the current frame. If E is an object reference, however, it prints the fields of the referenced object.

info locals
: Print all local variables in the current frame.

quit
: Leave GJDB.

halt!!!
: This command may be entered at any time, even when a program is running. It causes the program to stop. This serves the purpose of Control-C in ordinary GDB.

`break` *place*
: Establishes a breakpoint; the program will halt when it gets there. The easiest breakpoints to set are at the beginnings of functions, as in

```
(gjdb) break Greeting.greet
Breakpoint set in Greeting.greet
```

You may also set breakpoints at particular lines in a source file:

```
(gdb) break Greeting.java:55
Breakpoint set at Greeting.java:55
```

This command will sometimes not work if the class that contains the line at which you wish to stop has not yet been "loaded." The `load` command (see below) will fix that.

When you run your program and it hits a breakpoint, you'll get a message and prompt like this.

```
Breakpoint hit: Greeting.greet at Greeting.java:55
(jgdb main[1])
```

The extra `main[1]` in the prompt tells you that you are looking at the top (first level) of the stack in the *thread* called `main`. At least at first, you will seldom see any other thread; we will get into what a "thread" rather late in the course. The number in square brackets changes with `up` and `down` commands. In Emacs, you may also use `C-c C-b` to set a breakpoint at the current point in the program (the line you have stepped to, for example) or you may move the point to the line at which you wish to set a breakpoint, and type `C-x SPC` (Control-X followed by a space) or `C-x C-a C-b`.

`delete` *place*
: Removes breakpoint number at *place*, or all breakpoints if you leave off *place*.

`cont` or `continue`
: Continues regular execution of the program. In Emacs, you may use `C-c C-r`.

`step`
: Executes the current line of the program and stops on the next statement to be executed. In Emacs, you may use `C-c C-s`. This command has some bugs, last time I checked, and sometimes does not stop soon enough. In general, you should use `next` until you are actually on a line that contains a function call you wish to step into.

`next`
: Like `step`, however if the current line of the program contains a function call (so that `step` would stop at the beginning of that function), does not stop in that function. In Emacs, you may use `C-c C-n`.

`finish`
: Keeps doing `next`s, without stopping, until reaching the end of the current function. In Emacs, you may use `C-c C-f`. (At the moment, this command is rather fragile, and may, I fear, disappoint.)

`load` *class* You must sometimes inform `gjdb` of the existence of certain classes (see `break`, above). Use this command to do so.