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

CS61B **P. N. Hilfinger**
Fall 2015

The GJDB Debugger

A *debugger* is a program that runs other programs, allowing its user to exercise some degree of control over these programs, and to examine them when things go amiss. Sun Microsystems, Inc. distributes a text-based debugger, called JDB, with its Java Developer's Kit (JDK). I have modified JDB to make its commands look pretty much like GDB, the GNU Debugger[1], which handles C, C++, Pascal, Ada, and a number of other languages. The result is called GJDB (g'jay dee bee). Perhaps the most convenient way to use it is through the interface supplied with Emacs.

GJDB is dauntingly chock-full of useful stuff, but for our purposes, a small set of its features will suffice. This document describes them.

1 Basic functions of a debugger

When you are executing a program containing errors that manifest themselves during execution, there are several things you might want to do or know.

- What statement or expression was the program executing at the time of a fatal error?
- If a fatal error occurs while executing a function, what line of the program contains the call to that function?
- What are the values of program variables (including parameters) at a particular point during execution of the program?
- What is the result of evaluating a particular expression at some point in the program?
- What is the sequence of statements actually executed in a program?
- When does the value of a particular variable change?

These functions require that the user of a debugger be able to *examine* program data, to obtain a *traceback* (a list of function calls that are currently executing sorted by who called whom), to set *breakpoints* where execution of the program is suspended to allow its data to be examined, and to *step* through the statements of a program to see what actually happens. GJDB provides all these functions. It is a *symbolic* or *source-level* debugger, creating the fiction that you are executing the Java statements in your source program rather than the machine code they have actually been translated into.

[1]The recursive acronym GNU means "GNU's Not Unix" and refers to a larger project to provide free software tools.

2 Preparation

In this course, we use a system that compiles (translates) Java programs into executable files containing *bytecode,* a sort of machine language for an idealized virtual machine that is considerably easier to execute than the original source text. This translation process generally loses information about the original Java statements that were translated. A single Java statement usually translates to several machine statements, and most local variable names are simply eliminated. Information about actual variable names and about the original Java statements in your source program is unnecessary for simply executing your program. Therefore, for a source-level debugger to work properly, the compiler must retain some of this superfluous information (superfluous, that is, for execution).

To indicate to our compiler (`javac`) that you intend to debug your program, and therefore need this extra information, add the `-g` switch during both compilation. For example, if you are compiling an application whose main class is called `Main`, you might compile with

```
javac -g Main.java
```

This sample command sequence produces a *class file* `Main.class` containing the translation of the class `Main`, and possibly some other class files.

3 Starting GJDB

To run this under control of `gjdb`, you can type

```
gjdb Main
```

in a shell. You will be rewarded with the initial command prompt:

```
[-]
```

This provides an effective, but unfrilly 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.

4 Threads and Frames

When GJDB starts, your program has not started; it won't until you tell GJDB to run it (you tell the program is not started from GJDB's prompt, which will be `[-]`). After the program has started and before it exits, GJDB will see a set of *threads,* each one of which is essentially a semi-independent program. If you haven't encountered Java threads before, the part of your program that you usually think of as "the program" will be the *main thread,* appropriately named `main`. However, there will also be a bunch of *system threads* (running various support activities), that GJDB will tell you about if asked, but which will generally not be of interest. GJDB can examine one thread at a time; which one being indicated by the prompt:

[-] Means there are no threads; the program has not been started.

[?] Means the program is started, but GJDB is not looking at any particular thread. You'll often see this if you interrupt your program.

name[*n*] Means that GJDB is looking at thread *name*, and at frame #*n* (see below) within that thread.

```
class Example {
  public static void main (String[] args) {
    for (int i = 0; i < args.length; i += 1)
      process (args[i]);           // (A)
  }

  static void process (String x) {
    ilog (Integer.parseInt(x), 0); // (B)
  }

  static void ilog (int x, int a) {
    if (x <= 1)
       report (a);                 // (C)
    else
       ilog (x/2, a+1);            // (D)
  }

  static int report (int x) {
    System.out.println (x);        // (E)
  }
}
```

At any given time, a particular thread is in the process of executing some statement inside a function (method)[2]. To arrive inside that method, the program had to execute a method call in a statement of some other method (or possibly the same, in the case of recursion), and so on back to the mysterious system magic that started it all. In other words, in each thread, there is a sequence of currently active method calls, each of which is executing a particular statement, and each of which also has a bunch of other associated information: parameter values, local variable values and so forth. We refer to each of these active calls as *frames,* or sometimes *stack frames,* because they come and go in last-in-first-out order, like a stack data structure. Each has a *current location,* which is a statement or piece of a statement that is currently being executed in that call (sometimes called a *program counter* or, confusingly, *PC*). The most recent, or *top* frame is the one that is executing "the next statement in the program," while each of the other frames is executing a (so-far incomplete) method call.

For example, consider the simple class `Example` on page 3. Suppose we start the program with command-line argument 5, and are stopped at statement (E). Then (for the main thread) GJDB sees frames #0–#5, as follows:

Frame#	Method	Location	Variables
0.	`report`	(E)	`x: 2`
1.	`ilog`	(C)	`x: 1, a: 2`
2.	`ilog`	(D)	`x: 2, a: 1`
3.	`ilog`	(D)	`x: 5, a: 0`
4.	`process`	(B)	`x: "5"`
5.	`main`	(A)	`args: { "5" }`

[2]Even when your program is initializing a field in a record, which doesn't *look* as if it's inside a method, it is actually executing a part of either a constructor or a special "static initializer" method (which you'll see in certain listings under the name `<clinit>`).

5 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 possible topics.

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

    `java` *Main command-line-arguments*

  to a Unix shell. GJDB remembers the arguments you pass, and plain `run` thereafter will restart your program from the top with those arguments. By default, the standard input to your program will come from the terminal (which causes some conflict with entering debugging commands: see below). However, you may take the standard input from an arbitrary file by using input redirection: adding `<` *filename* to the end of the *command-line-arguments* uses the contents of the named file as the standard input (as it does for the shell). Likewise, adding `>` *filename* causes the standard output from your program to go to the named file rather than to the terminal, and `>&` *filename* causes both the standard output and the standard error output to go to the named file.

`where`
: Produce a backtrace—the chain of function calls that brought the program to its current place. The commands `bt` and `backtrace` are synonyms.

`up`
: Move the current frame that GJDB is examining to the caller of that frame. Very often, your program will blow up in a library function—one for which there is no source code available, such as one of the I/O routines. You will need to do several `up`s to get to the last point in your program that was actually executing. Emacs (see below) provides the shorthand `C-c<` (Control-C followed by less-than), or the function key `f3`.

`up` n Perform n `up` commands (n a positive number).

`down`
: Undoes the effect of one `up`. Emacs provides the shorthands `C-c>` and function key `f4`.

`down` n Perform n `down` commands (n a positive number).

`frame` n Perform `up`s or `down`ss as needed to make frame #n the current frame.

`thread` n Make thread #n (as reported by `info threads`, below) the current thread that GJDB is examining.

`print` E
: prints the value of E in the current frame in the program, where E is a Java expression (often just a variable). For example

```
main[0] print A[i]
$1 = -14
main[0] print A[i]+x
$2 = 17
```

This tells us that the value of `A[i]` in the current frame is -14 and that when this value is added to `Main.x`, it gives 17. The notations '*$n*' to the left of the equal signs are *history variables.* You can reference them in subsequent `print` commands, and can also use '$' without a number to refer to the most recent (highest-numbered) history variable. For example:

```
main[0] print $ + $1
$3 = 3
```

Here, '$' is a synonym for '$2', since that was the last expression printed in our running example.

Printing a reference value is less informative:

```
main[0] p args
$4 = instance of java.lang.String[3] (id=172)
```

This tells you that `args` contains a pointer to a 3-element array of strings, but not what these strings are.

`print/`*n* *E* also prints the value of expression *E* in the current frame. If *E* is a reference value, however, it also prints the subcomponents (fields or array elements) of the referenced object to *n* levels. Plain `print` without this specification is equivalent to `print/0`, and does not print subcomponents. *Printing subcomponents to one level* means printing each subcomponent of *E*'s value as if by `print/0`. Printing to two levels prints means printing each subcomponent as if by `print/1`, and so forth recursively. For example,

```
main[0] print/1 args
$5 = instance of java.lang.String[3] (id=172) {
"A", "B", "C"
}
main[0] p T
$6 = instance of Tree(id=176)
main[0] p/1 T
$7 = instance of Tree(id=176) {
  label: "A"
  left: null
  right: instance of Tree(id=178)
}
main[0] p/2 T
$8 = instance of Tree(id=176) {
  label: "A"
  left: null
  right: instance of Tree(id=178) {
    label: "B"
    left: null
    right: instance of Tree(id=180)
  }
}
```

`dump` *E*
: Equivalent to `print/1` *E*.

`dump/`*n* *E*
: Equivalent to `print/`*n* *E*.

`info locals` Print the values of all parameters and local variables in the current frame.

`info threads` List all current threads.

quit
Leave GJDB.

The commands to this point give you enough to pinpoint where your program blows up, and usually to find the offending bad pointer or array index that is the immediate cause of the problem (of course, the actual error probably occurred much earlier in the program; that's why debugging is not completely automatic.) Personally, I usually don't need more than this; once I know where my program goes wrong, I often have enough clues to narrow down my search for the error. You should *at least* establish the place of a catastrophic error before seeking someone else's assistance.

The next bunch of commands allow you to actively stop a program during normal operation.

`suspend` **and** `C-f`
When a program is run from a Unix shell, `C-c` will terminate its execution (usually). At the moment, unfortunately, it will also do this to GJDB itself. When debugging, you usually want instead to simply stop the debugged program temporarily in order to examine it. When the standard input is redirected from a file (using '`<`'; see the `run` command), you can simply use `suspend` to stop the program (and then use `continue` or `resume` to restart). When the program is running and standard input comes from the terminal, things get complicated: how does GJDB know a command from program input. If you are using GJDB mode (see §7), then `C-c C-c` will do the trick in this case. Otherwise, if you are running in an ordinary shell, use `C-f` following by return. And finally, if you are running in a shell under Emacs, use `C-qC-f` followed by return.

`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

```
[-] break Example.process
Set breakpoint request Example:8
```

(using the class `Example` from §4). Use the full method name (complete with class and package qualification), as shown. You will either get a confirming message as above (saying that the system set a breakpoint at line 8 of the file containing class `Example`), or something like

```
Deferring BP RatioCalc.main [unresolved].
It will be set after the class is loaded.
```

when you set a breakpoint before the class in question has been loaded. Breakpoints in anonymous classes are a bit tricky; their names generally have the form "$C\$n$" where C is the name of the outermost class enclosing them, and n is some integer. The problem is that you don't generally know the value of n. GJDB therefore allows "C`.0`" as a class name, meaning "any anonymous class inside C."

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

```
Breakpoint hit: thread="main", Example.main(), line=4, bci=22
main[0]
```

(Here, "bci" indicates a position within the bytecode translation of the method; it is not generally very useful). Emacs allows you to set breakpoints with the mouse (see §7).

`condition` *N cond* Make breakpoint number *N* conditional, so that the program only stops if *cond,* which must be a boolean expression, evaluates to true.

`condition` *N* Make breakpoint number *N* unconditional.

`delete`
Removes breakpoints. This form of the command gives you a choice of breakpoints to delete, and is generally most convenient.

`cont` or `continue`
Continues regular execution of the program from a breakpoint or other stop.

`step`
Executes the current line of the program and stops on the next statement to be executed.

`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.

`finish`
Does `next`s, without stopping, until the current method (frame) exits.

6 Common Problems

Name unknown. When you see responses like this:

```
main[0] print x
Name unknown: x
main[0] print f(3)
Name unknown: f
```

check to see if the variable or method in question is static. A current limitation of the debugger is that you must fully qualify such names with the class that defines them, as in

```
main[0] print Example.f(3)
```

Make sure that fully qualified names include the package name.

Ignoring breakpoints. For a variety of reasons, it is possible for a program to miss a breakpoint that you thought you had set. Unfortunately, GJDB is not terribly good at the moment at catching certain errors. In particular, it will tell you that a breakpoint has been deferred, when in fact it will never be hit due to a class name being misspelled.

7 GJDB use in Emacs

While one *can* use `gjdb` from a shell, nobody in his right mind would want to do so. `Emacs` provides a much better interface that saves an enormous amount of typing, mouse-moving, and general confusion. Executing the Emacs command `M-x gjdb` starts up a new window running `gjdb`, and enables a number of Emacs shortcuts, as well as providing a **Debug** menu for issuing many GJDB commands.

This command prompts for a command string (typically `gjdb` *classname*) and (for certain historical reasons) creates a buffer named `*gud-`*classname*`*`. Emacs intercepts output from `gjdb` and interprets it for you. When you stop at a breakpoint, Emacs will take the file and line number reported by `gjdb`, and display the file contents, with the point of the breakpoint (or error) marked. As you step through a program, likewise, Emacs will follow your progress in the source file. Other commands allow you to set or delete breakpoints at positions indicated by the mouse.

The following table describes the available commands. On the left, you'll find the text command line, as described in §5. Next comes the **Debug** menu button (if any) that invokes the command. This menu applies both to the GJDB buffer and to buffers containing `.java` files. Next come the Emacs shortcuts: sequences of keys that run the commands. The shortcuts are slightly different in the GJDB buffer as opposed to buffers containing source (`.java`) files, so there are two columns of shortcuts. The last column contains further description. Finally, here are a few reminders about Emacs terminology:

1. In shortcuts, `C-`x means "control-x," `S-`x means "shift-x," `f`n refers to one of the function keys (typically above the keyboard), and `SPC` is the space character.

2. The *point,* in Emacs, refers to the location of the cursor; there is one for each buffer. You can set the point using the usual motion commands when in the buffer, or by simply clicking the mouse at the desired spot.

3. The *region* in any given buffer is a section of text (usually shadowed or highlighted so that you can tell where it is). One convenient way to set it is by dragging the mouse over the text you want included while holding down the left mouse button.

Table 1: Summary of Commands for Program Control

Command Line	Emacs Menu	Emacs GJDB buffer	Emacs `.java` buffer	Description
`next`	**Step Over**	`f6` or `C-c C-n`	`f6`	Execute to the next statement of the program; if this statement contains function calls, execute them completely before stopping. [See Note 3, below]
`step`	**Step Into**	`f5`, or `C-c C-s`	`f5`	Execute to the next statement of the program; if this statement calls a function, stop at its first line. [See Note 3, below]
`finish`	**Finish Function**	`f7` or `C-c C-f`	`f7`	Execute until the current function call returns.
`continue`	**Continue**	`f8` or `C-c C-r`	`f8`	Continue execution of stopped program.
`suspend`	**Interrupt**	`C-c C-c`		Interrupt execution of program and suspend its threads.
`C-f`	**Interrupt**	`C-c C-c`		Same as `suspend`, but works in cases where the debugged program is running and GJDB is passing input to it from the terminal.
`break` *file:line#*	**Set Breakpoint**		`C-x SPC`	Set a breakpoint at the point (applies only to the source buffer).
`delete` *file:line#*	**Clear Breakpoint**			Remove a breakpoint at the point (applies only to the source buffer).
`run`	**Run**			(Re)start the program, using the last set of command-line arguments. Only available in the GJDB buffer.
`quit`	**Quit**			Leave GJDB. Only available in the GJDB buffer.
-	**Refresh**			Re-arrange Emacs' windows as needed to display the current source line that GJDB is looking at.
-	**Start Debugger**			Run `gjdb` on the class in this (source) buffer.

Table 2: Summary of Commands for Examining a Program

Command Line	Emacs			Description
	Menu	GJDB buffer	`.java` buffer	
`print` *expr*	**Print**	`f9`	`f9`	Evaluate *expr* and print, without showing any subcomponents of the value. Emacs commands apply either to the contents of the region, or if it is inactive, to the variable, field selection, or function call at or after the point.
`dump` *expr*	**Print Details**	`S-f9`	`S-f9`	Evaluate *expr* and print, also printing any components (array elements or fields). With Emacs, gets the expression to print as for `print`.
`info locals`				Print (as for the `print` command) the values of all local variables in the current frame.
`up`	**View Caller**	`f3` or `C-c <`	`f3`	Move the debugger's current focus of attention up one frame; if looking at frame n at the moment, we switch to frame $n-1$.
`down`	**View Callee**	`f4` or `C-c >`	`f4`	Move the debugger's current focus of attention down one frame (from frame n to frame $n+1$). Opposite of `up`.
`where`				Print a backtrace, showing all active subprogram calls.
`info threads`				List all threads in the program.
`thread` N				Make thread #N be the one that GJDB is currently examining.