

CS1Bh Lecture Note 17

Debugging Java Code

This note has three aims:

1. To demonstrate debugging as a way of locating errors in programs.
2. To introduce **jdb**, the Java command-line debugger supplied as part of the Java SDK.
3. To give a small example of the use of **jdb** in debugging programs.

17.1 A command-line debugger

The Java debugger, **jdb** is a command-line utility that enables you to debug Java programs. The Java debugger provides debugging support by interacting with the Java runtime interpreter. This allows you to interact with the interpreter to interrupt the normal execution of your Java program to look at the values of variables as execution progresses. The debugger is a command-line tool. This means it has only rather primitive ways of getting information from the Java interpreter but it is capable of providing all the information available in Java debuggers with more elaborate user interfaces.

In this note we introduce some new terminology:

- A *breakpoint* is a line of code specified by the user using the debugger that the interpreter will halt at each time it reaches that point in the program.
- *Single-stepping* is the process of executing your code one line at a time (in single steps).

Before you can use **jdb**, you must compile your code so that it includes debugging information. The compiler switch `-g` for Sun's Java compiler causes the compiler to generate debugging tables containing information about line numbers and variables. So if you want to debug the `PartA.java` file from a practical exercise, you should re-compile the program with:

```
javac -g PartA.java
```

The process of debugging is one of locating the error, redesigning the relevant part of the code and then implementing it. Debuggers are designed to help only with the

first stage of the process. When you first see a bug you may have no idea where the bug is located in your program. The debugger is a tool to narrow this search to a small number of lines that can be inspected closely for errors.

This note concentrates on the use of the Java debugger and the basic facilities you need to debug simple applications and applets. Here we concentrate on debugging applications. Applets are a little more complex because applets will usually have more than one thread of execution to consider. This note considers basic use of the debugger for the kinds of programs you are likely to write during your coursework.

How to use jdb

Using **jdb** is just like using the Java interpreter. In fact **jdb** uses the java interpreter to run the programs being debugged. You invoke the debugger by typing the command:

```
jdb [options] <Classname>
```

The `Classname` argument is optional and specifies the name of the class you want to execute. The fact that `Classname` is optional brings up an interesting point regarding the usage of the debugger: there are two different ways to go about using the debugger, depending on whether you are debugging an application or an applet. For applications, you simply execute **jdb** directly and provide the name of the main class in the `Classname` argument, as the previous syntax shows. If you are debugging an applet, however, you must execute the debugger within the applet viewer, like this:

```
appletviewer -debug URL
```

In this case, `URL` refers to a document URL containing an HTML page with the applet to be debugged. Instead of directly executing the class, the applet viewer launches the debugger and allows you to debug the applet. Technically, there are three ways to use the Java debugger. The third technique involves attaching the debugger to an application that is already running in the interpreter. In this case the options to **jdb** specify which interpreter we should attach to. This method of using **jdb** will not concern us here so we do not consider the `[options]` further.

Commands

When the debugger is up and running, you control it through commands that are entered at a command-line prompt. The debugger command-line prompt is a `>` prompt by default, similar to DOS or UNIX shell prompts. This prompt specifies that there is no default thread running. The thread that is currently executing in the debugger is displayed in the command prompt itself, so the `>` prompt signifies that no thread is currently being debugged. When you are debugging a thread, the command prompt changes to a thread name followed by the current position of the stack frame, which is enclosed in square brackets. An example of a thread prompt is `main[1]`, which signifies that the main thread is running and you are at the topmost position (1) in the stack frame.

Here is a list of some of the most useful debugging commands:

- `help` prints out a listing of all the available commands and a brief explanation of what they do.
- `locals` displays the current value of all the objects in the current scope (stack frame).
- `list` lists the code round about the line at which the debugger has stopped execution of the program.
- `print Object` prints both entire objects and individual member variables; you simply specify the name of the object or member variable in the `Object` argument.
- `dump Object` prints objects or member variables, but it prints more detailed information such as an object's inheritance.
- `methods Class` lists the methods in a class.
- `run` runs the main thread.
- `classes` list all the classes in the application.
- `stop in Classname.Methodname` sets a breakpoint when the named method is invoked.
- `stop at Classname:LineNumber` this is the same as the previous command but it specifies the breakpoint by giving a line number.
- `step` steps on the execution of the program by one line.
- `cont` continue running after stopping at a breakpoint.
- `clear Classname:LineNumber` clears the named breakpoint.

Now we have an idea of how to look at the values of different things in the debugger, let's move on to some commands that are a little more exciting. The `stop in` and `stop at` commands are used to set breakpoints in methods and at specific lines of source code, respectively. For example, to set a breakpoint in the `mouseDown` method of an applet called `Dynamic`, you would type the following command at the debugger command line:

```
stop in Dynamic.mouseDown
```

When you click the mouse button in the applet window, the debugger will halt the applet at the beginning of the `mouseDown` method. To begin single-stepping through the method, we use the `step` command. The debugger executes one line of code for each `step` command issued. When we find out the information we need and are ready to get things running at full speed again, we use the `cont` command, which continues the normal execution of the program until execution hits the next breakpoint. Likewise, it is possible clear any breakpoints with the `clear` command.

That sums up the basics of using the Java debugger. Like any powerful tool, the best way to gain confidence with the debugger by simply tinkering with it. Try running the debugger on a simple program and getting acquainted with some of the commands before trying to take on a serious debugging project.

17.2 An example session with jdb

Suppose we are debugging the program written for the CS1Ah Calculator practical exercise.¹ In developing the code we may very well have omitted some simple action that needs to be taken to ensure correct operation. For example, suppose in developing the add method we forget to include the statement in which correctly sets the op field to its correct value. To simulate that here, we comment out the line:

```

38     public int add(int right)
39     {
40         apply(right);
41         //op = '+';
42         return left;
43     }

```

We now execute the JUnit test case `PartAAddTest` as described in CS1Bh Lecture Note 9. There is a reminder of the test program on the final page, and here are the commands used to compile and run it:²

```

[tarbhaidh]da: export CLASSPATH=./usr/java/lib/junit/junit.jar
[tarbhaidh]da: javac -g *.java
[tarbhaidh]da: java PartAAddTest

```

Running our test suite on the resulting code uncovers four errors in the six test cases included in the test harness:

```

...F.F.F.F
Time: 0.158
There were 4 failures:
...
FAILURES!!!

```

Unfortunately the output produced by the test case failures does little to illuminate the source of the problem (why should some additions work but others fail?). So let's run **jdb** on the erroneous code in an attempt to locate the bug so we can fix it. In the following transcripts of a debugging session each line of user input is preceded by a prompt, either `>` or `main[n]`.

```

[tarbhaidh]da: jdb PartAAddTestInitializing jdb ...

```

```

\prompt{> stop in PartA.add}
Deferring breakpoint PartA.add.
It will be set after the class is loaded.

```

The input above sets a breakpoint and then starts off the execution of the main thread in the program. In the next part of the transcript we run the program, and then use `cont` to force the interpreter to continue after stopping at each call of the add method, until we see the `F` first test failure indicator from JUnit. Then we inspect the code.

¹Of course, it seems rather late to be thinking of debugging that program now! You may imagine that the Java debugger would have been useful to you earlier in the course, but to use a debugger effectively it is useful to first understand the behaviour of the machine.

²Invoke Junit's GUI with: `java junit.swingui.TestRunner PartAAddTest`.

```

> run
run PartAAddTest
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: .Set deferred breakpoint PartA.add

Breakpoint hit: "thread=main", PartA.add(), line=40 bci=0
40      apply(right);

main[1] cont
.>
Breakpoint hit: "thread=main", PartA.add(), line=40 bci=0
40      apply(right);
main[1] cont
.>
Breakpoint hit: "thread=main", PartA.add(), line=40 bci=0
40      apply(right);
main[1] cont
> F.
Breakpoint hit: "thread=main", PartA.add(), line=40 bci=0
40      apply(right);
main[1] list
36      }
37
38      public int add(int right)
39      {
40 =>      apply(right);
41          //op = '+';
42          return left;
43      }
44
45      public int subtract(int right)
main[1] print left
left = 0
main[1] print right
right = 1
main[1] step
Step completed: "thread=main", PartA.apply(), line=20 bci=0
20      if (op == '+')
main[1] step
Step completed: "thread=main", PartA.apply(), line=22 bci=22
22      else if (op == '-')
main[1] step
Step completed: "thread=main", PartA.apply(), line=24 bci=44
24      else if (op == '/')
main[1] step

```

```

Step completed: "thread=main", PartA.apply(), line=26 bci=66
26      else if (op == '*')
main[1] step
Step completed: "thread=main", PartA.apply(), line=28 bci=88
28      else left = right;
main[1] step
Step completed: "thread=main", PartA.apply(), line=29 bci=93
29      }
main[1] step
Step completed: "thread=main", PartA.add(), line=42 bci=5
42      return left;
main[1] print left
    left = 1
main[1] print right
    right = 1
main[1] print op
    op =

```

Here we have progressed the application to the point where we are adding 1 to zero. As we step through we can use list to see where we have reached in the code³. At this point we have detected the problem by looking at the value of `op`. We can see that it is not set to `''+''` as it should be after calling `add`. The remainder of the transcript just records the behaviour of the program on the final set of tests.

```

main[1] cont
F.>
Breakpoint hit: "thread=main", PartA.add(), line=40 bci=0
40      apply(right);
main[1] cont
F.>
Breakpoint hit: "thread=main", PartA.add(), line=40 bci=0
40      apply(right);
main[1] print left
    left = 0
main[1] cont
Breakpoint hit: "thread=main", PartA.add(), line=40 bci=0
40      apply(right);
main[1] cont
F
> Time: 640.504
There were 4 failures:
....
FAILURES!!!
Tests run: 6,  Failures: 4,  Errors: 0

```

The application exited

³Notice that on the second use of list we seem to have moved backwards, this is because step moves us to the end of the if statement and this starts at line 37.

Summary

- Debuggers are used to locate faults in programs by finding errors in the state of the system arising from the faults.
- The debugger in the Java SDK is called **jdb**, it is a simple command line utility with a number of commands to control the system. More sophisticated graphical debuggers are provided in Integrated Development Environments. The **Bluej** environment provides a simple graphical debugger.
- The main facilities provided by debuggers are the ability to interrupt the program and the ability to inspect the variables of the program.

Questions

1. Choose one of the programs you have written in the last year, write some tests for it and find an error to introduce into the program that causes some tests to fail. Try using **jdb** to track down your error.
2. When we are debugging recursive programs we may want to inspect the stack. Experiment with doing this in **jdb**, to understand the structure of recursive invocation of methods.
3. Experiment with **jdb**— how do you think it could be improved?
4. Consider how **jdb** works. What special support do you think may have to be provided for debugging by the Java Virtual Machine? If the JVM did not provide specialised support, how might you implement a debugger otherwise?

*Note first prepared by Stuart Anderson.
Some text based on Debugging chapter from “Java Unleashed”.
Updated and extended by David Aspinall..
David Aspinall, 2003/03/05 23:39:37.*

Excerpts of Calculator PartA

```
38     public int add(int right)
39     {
40         apply(right);
41         //op = '+';
42         return left;
43     }

19     private void apply(int right) {
20         if (op == '+')
21             left += right;
22         else if (op == '-')
23             left -= right;
24         else if (op == '/')
25             left /= right;
26         else if (op == '*')
27             left *= right;
28         else left = right;
29     }
```

Excerpt of PartAAddTest

```
27     public void add0to0() {additionTest(0,0,0);}
28     public void add0to1() {additionTest(0,1,1);}
29     public void add1to0() {additionTest(1,0,1);}
30     public void add1to1() {additionTest(1,1,2);}
31     public void add9to42() {additionTest(9,42,51);}
32     public void compositeAdd123() {
33         partA.add(1);
34         partA.add(2);
35         complete(3,6);
36     }
37
38     public static Test suite() {
39         TestSuite suite = new TestSuite();
40         suite.addTest(new PartAAddTest("add0to0"));
41         suite.addTest(new PartAAddTest("add0to1"));
42         suite.addTest(new PartAAddTest("add1to0"));
43         suite.addTest(new PartAAddTest("add1to1"));
44         suite.addTest(new PartAAddTest("add9to42"));
45         suite.addTest(new PartAAddTest("compositeAdd123"));
46         return suite;
47     }
```