Java Q&A

Debug with jdb

How do you use this crazy thing?

By Tony Sintes

Q: How do you use `jdb` (included in the JDK 1.2 package) effectively to debug Java programs?

I've tried many times, but I am successful only in loading a class file to `jdb`; I can't debug it. The `help` command isn't much use.

A: You ask an interesting question. To be honest, I've *never* used `jdb`. I have always used the debugger provided by my IDE environment. So to answer your question I had to do a little research of my own.

This month's Java Q&As

- Study hall

Want more? See the **Java Q&A** index for the full Q&A catalog.

Do you have a burning Java question? One whose answer would benefit not just you but other *JavaWorld* readers too? We've got the experts to help. Submit your **Java Q&A** questions to javaqa@javaworld.com.

It turns out that Sun considers `jdb` a proof of concept for the Java Debugger API. The Java Debugger API allows us to actually peek into the runtime and debug our code. The `jdb` is just one implementation of a debugger that uses the API. Compared to the visual debuggers with which I'm familiar (yes, I guess I'm a wimp), it's not the easiest debugger to use -- though it is similar to other command-line debuggers, such as `gdb`.

Anyhow, on to your question. Before attempting to debug your code, be sure to use the `-g` option while compiling your classes. This option tells the compiler to include debugging information in your class file.

Let's define a contrived class for testing:

```
public class TestMe {

  private int int_value;
  private String string_value;

  public static void main(String[] args)
  {
```

```
    TestMe testMe = new TestMe();
    testMe.setInt_value(1);
    testMe.setString_value("test");

    int integer = testMe.getInt_value();
    String string = testMe.getString_value();

    String toString = testMe.toString();
  }

  public TestMe()
  {
  }

  public int getInt_value()
  {
    return int_value;
  }

  public String getString_value()
  {
    return string_value;
  }

  public void setInt_value(int value)
  {
    int_value = value;
  }

  public void setString_value(String value)
  {
    string_value = value;
  }

  public String toString()
  {
    return "String value: " + string_value + " int value: " + int_value;
  }

}
```

Start the debugger:

```
> jdb TestMe
```

You should see:

```
> Initializing jdb...
> 0xaa:class<TestMe>
```

Let's take a look at some basic commands. In order to set breakpoints, we need to know the line numbers or the method names of the places where we would like to break. To obtain a list of methods, simply use the `methods` command:

```
> methods TestMe
   void main(java.lang.String[])
   void <init>()
   int getInt_value()
   java.lang.String getString_value()
   void setInt_value(int)
   void setString_value(java.lang.String)
   java.lang.String toString()
```

Setting a breakpoint is simple. Use the following syntax:

```
stop in <class id>.<method>[<argument_type,...>]
```

Or:

```
stop at <class id>:<line>
```

We should start debugging at the beginning of the main method:

```
> stop in TestMe.main
   Breakpoint set in javaworld.TestMe.main
```

Now that we have a breakpoint, we can begin execution. To run up to the breakpoint, simply use the `run` command:

```
> run
run javaworld.TestMe
running ...
main[1]
Breakpoint hit: javaworld.TestMe.main (TestMe:10)
```

At this point, the debugger halts execution at the first line of the main method. Notice that the cursor has changed to reflect the method that we are currently in.

The `list` command will display the code at the breakpoint. An arrow indicates the spot where the debugger has halted execution.

```
main[1] list
6 private String string_value;
7
8 public static void main(String[] args)
9 {
10 => TestMe testMe = new TestMe();
11 testMe.setInt_value(1);
12 testMe.setString_value("test");
13
14 int integer = testMe.getInt_value();
main[1]
```

Next, we'll want to `step` through a few lines of code and see what's changed:

```
main[1] step
main[1]
Breakpoint hit: javaworld.TestMe.<init> (TestMe:20)
main[1] locals
Method arguments:
Local variables:
  this = String value: null int value: 0
main[1] list
16
17 String toString = testMe.toString();
18 }
19
20 => public TestMe()
21 {
22 }
23
24 public int getInt_value()
main[1] step
main[1]
Breakpoint hit: java.lang.Object.<init> (Object:27)
main[1] list
Unable to find Object.java
main[1] step
main[1]
Breakpoint hit: javaworld.TestMe.<init> (TestMe:22)
```

```
main[1] list
18 }
19
20 public TestMe()
21 {
22 => }
23
24 public int getInt_value()
25 {
26 return int_value;
main[1] step
main[1]
Breakpoint hit: javaworld.TestMe.main (TestMe:10)
main[1] list
6 private String string_value;
7
8 public static void main(String[] args)
9 {
10 => TestMe testMe = new TestMe();
11 testMe.setInt_value(1);
12 testMe.setString_value("test");
13
14 int integer = testMe.getInt_value();
main[1] step
main[1]
Breakpoint hit: javaworld.TestMe.main (TestMe:11)
main[1] list
7
8 public static void main(String[] args)
9 {
10 TestMe testMe = new TestMe();
11 => testMe.setInt_value(1);
12 testMe.setString_value("test");
13
14 int integer = testMe.getInt_value();
15 String string = testMe.getString_value();
main[1] locals
Method arguments:
Local variables:
  args =
  testMe = String value: null int value: 0
```

After each `step`, I called the `list` command to see where I was in the code. The return value from the command listed the line number, but somehow that didn't really help me very much.

As we `step`, we see that the main method is constructing a `TestMe` instance. Each step takes us through the constructor and finally back into the main method. The `locals` command lists all of the local variables visible in the current stack. We see that at this point in the main method there are only two local variables: `args` and `testMe`.

By using `step`, we can get inside any of the methods to see what is going on. When we combine `step` with the `locals` command we can see our variables:

```
main[1] step
main[1]
Breakpoint hit: javaworld.TestMe.setInt_value (TestMe:36)
main[1] list
32 }
33
34 public void setInt_value(int value)
35 {
36 => int_value = value;
37 }
38
39 public void setString_value(String value)
40 {
```

```
main[1] locals
Method arguments:
Local variables:
  value = 1
  this = String value: null int value: 0
```

If we `step` one more time, we end up in the `setInt_value()` method. If we `step` two more times, the method will set the `int_value` member to `1` and return. (To check to see that the method set the value, use the `locals` command.)

Of course, when we `step`, we won't always want to trace into each method we encounter. Some method calls can nest very deeply. If we were forced to trace through an entire hierarchy, we might never finish. Luckily, `jdb` has a way to execute a method *without* tracing into that method: the `next` command.

`jdb` also provides a few other `step` commands. The `stepi` command executes the current instruction. In other words, the code at the `=>` will execute but the current line will not advance to the next instruction. You can call `stepi` a million times, but the `=>` displayed from the `list` command will not move.

`jdb` also provides the `step up` command. The `step up` call executes until the current method returns to its caller. Simply put, this stepper executes a method and nothing else. Take the following code segment as an example:

```
int integer = testMe.getInt_value();
```

If this is our current line and we run `step up`, the `getInt_value()` method will execute. However, that's all that will happen. The return value will not get set to `integer`.

`jdb` also allows us to set multiple breakpoints. To go from one breakpoint directly to the next, `jdb` provides the `cont` command.

Finally, there are times when we want to look at all the members of an instance or class. Luckily, `jdb` provides the `dump` and `print` commands:

```
main[1] dump TestMe
TestMe = 0xa9:class(javaworld.TestMe) {
    superclass = 0x2:class(java.lang.Object)
    loader = (sun.misc.Launcher$AppClassLoader)0xaa
}
main[1] print TestMe
TestMe = 0xa9:class(javaworld.TestMe)
main[1] dump testMe
testMe = (javaworld.TestMe)0xec {
    private java.lang.String string_value = test
    private int int_value = 1
}
main[1] print testMe
testMe = String value: test int value: 1
```

When you run `dump` or `print` on a class, you get class information, which includes superclass and loader information. When you run `dump` and `print` on an instance, you get instance information, such as data members and their current values.

`jdb` also provides commands for getting down and dirty in the threads and stacks. However, these commands are really beyond the scope of a `jdb` intro.

One final point: you may ask, "How do you effectively use `jdb`?" The effectiveness of use will depend

on your comfort level with `jdb`. When you first use `jdb`, the most important command is `help`. The `help` command lists each command and provides some basic information to help you get started. Once you have the `help` command mastered, you'll find yourself using the commands that set breakpoints, along with `step` and `list`. Any combination of those commands will allow you to get started using `jdb`. `step`, `list`, `step`, `list`... should help you quickly locate code that is bombing out on you.

About the author
Tony Sintes is a principal consultant at BroadVision. Tony, a Sun-certified Java 1.1 programmer and Java 2 developer, has worked with Java since 1997.

Home | Mail this Story | Resources and Related Links

Resources

- "Java Language Debugging," from the Postech ME Website: http://mech.postech.ac.kr/Java/java.sun.com/products/JDK/debugging/
- "`jdb`: The Java Debugger," from *Java Developer's Reference*, Mike Cohen, et al. (Sams.net Publishing, 1996): http://docs.online.bg/PROGRAMMING/JAVA_Developers_Reference/ch15.htm

Feedback: jweditors@javaworld.com
Technical difficulties: webmaster@javaworld.com
URL: http://www.javaworld.com/javaqa/2000-06/04-qa-0623-jdb.html
Last modified: Wednesday, February 21, 2001