CS2: Debugging in Java

Jon Cook (LFCS) April 2003

1. General Advice

Debugging is not always easy. Some bugs can take a long time to find. Debugging concurrent code can be particularly difficult and time consuming.

It helps to understand the language and its libraries. A common source of errors in Java is to use libraries without properly understanding what methods do and how they work. For example, one might forget to call a method to initialise an object.

It is a good idea to add toString methods to your own classes. Then when debugging you can simply print out an object of that class and see some of its data. You will see this in the example below.

It is also a good idea to add a main method to your class, which does some testing of that class. Then the class can be executed directly and you can test your classes separately. Again, you will see this at work in the example below.

2. Debugging by Adding Print Statements

One easy way to debug a program is to add print statements at various points reporting the status of the program: for example the values of variables. This is easy to do in Java as if `o` is an object, the `System.out.println( o )` will call the toString() method of `o` and print out the result. (This assumes an appropriate toString() method has been defined of course.)

It is important to make enough information visible but not too much. It can help to highlight important information in various ways so that it stands out from the output. e.g.

```
System.out.println( "****** Got to start of foo method, arguments "
   + arg1 + ", " + arg2 );
```

or

```
System.out.println( "------------------------" );
System.out.println( "a = " + a + ", b = " + b );
System.out.println( "------------------------" );
```

It is a bad idea to use print statements of the form:

```
System.out.println( "Got to here!" );
```

since when you run the program the next day you may have forgotten where "here" actually is.

A Debug Flag

If you add a class like the following class to your project:

```
public class Debug {
    static boolean debugOn = true;
    static int debugLevel = 3;
}
```

You can then, in your project, include code such as:

```
if( Debug.debugOn && Debug.debugLevel > 2 )
    System.out.println( "a = " + a + ", b = " + b );
```

This makes it easy to switch debugging output on and off and to set the amount of output you wish to see when you run the program.

Adding assert Statements

A new feature of Java 1.4 is the ability to add assertions to your code. These are statements of the form: `assert <boolean>;`. If the program is run with assertions switched off, this statement is just ignored. If assertions are on, though, and the boolean evaluates to false, then an Error will be thrown.

Assertions state that you expect some condition to be true at some point in your program. If that condition is not true at that point, then you want to be told about it.

Consider the following program:

```
public class MyClass {
    public static void main( String[] args ) {
        int a = Integer.parseInt( args[ 0 ] );
        int b = Integer.parseInt( args[ 1 ] );
        int sum = a + b + 1; // bug, + 1 is wrong.
        assert sum == a + b;
    }
}
```

This must be compiled with the command:

```
javac -source 1.4 MyClass.java
```

It can then be run as usual, or to switch on the assertions use one of the two following commands:

```
java -enableassertions MyClass 1 2
java -ea MyClass 1 2
```

In this case the assertion will fail and an error message will be printed indicating the line of the program which failed.

3. Common Errors and Bad Style

Bug: The Wrong Kind of Equals

The == operator, when applied to Objects, tests that the two objects are the same object, not whether they contain the same data. To test that two Strings, for example, contain the same text, use the `equals()` method.

For example:

```
String s = "Hello";
String t = s + "";
System.out.println( s == t ); // prints false
System.out.println( s.equals( t ) ); // prints true
```

Bug: Break Statements in Switches

If a case of a switch statement doesn't end in a break, then execution continues with the next case statement:

```
int i = 3;
switch( i ) {
    case 1: System.out.println( 1 ); break;
    case 3: System.out.println( 3 );
    case 5: System.out.println( 5 ); break;
}
```

will print out 3 then 5.

Bad Style: Catching All Exceptions

It is a bad idea to write code that looks like this:

```
try {
     ...some code....
} catch( Exception e ) { }
```

This catches all exception and if one is caught, it does nothing. This is very easy to do, but what if something goes wrong in the try block: you may not even find out that something went wrong, if it does, it will be hard to find out what.

It is better to catch more specific exceptions, or at the very least to print an error message.

4. Debuggers

jdb

After you have watched me debugging this code in the lecture, you could have a go at it yourself using jdb (the command line Java debugger).

Here are the commands for jdb:

```
Initializing jdb ...
> ** command list **
run [class [args]]        -- start execution of application's main class

threads [threadgroup]     -- list threads
thread <thread id>        -- set default thread
suspend [thread id(s)]    -- suspend threads (default: all)
resume [thread id(s)]     -- resume threads (default: all)
where [thread id] | all   -- dump a thread's stack
wherei [thread id] | all  -- dump a thread's stack, with pc info
up [n frames]             -- move up a thread's stack
down [n frames]           -- move down a thread's stack
kill <thread> <expr>      -- kill a thread with the given exception object
interrupt <thread>        -- interrupt a thread

print <expr>              -- print value of expression
dump <expr>               -- print all object information
eval <expr>               -- evaluate expression (same as print)
set <lvalue> = <expr>  -- assign new value to field/variable/array element
locals                 -- print all local variables in current stack frame

classes                   -- list currently known classes
class <class id>          -- show details of named class
methods <class id>        -- list a class's methods
fields <class id>         -- list a class's fields

threadgroups              -- list threadgroups
threadgroup <name>        -- set current threadgroup

stop in <class id>.<method>[(argument_type,...)]
                          -- set a breakpoint in a method
stop at <class id>:<line> -- set a breakpoint at a line
clear <class id>.<method>[(argument_type,...)]
                          -- clear a breakpoint in a method
clear <class id>:<line>   -- clear a breakpoint at a line
clear                     -- list breakpoints
catch [uncaught|caught|all] <exception-class id>
```

```
                          -- break when specified exception occurs
ignore [uncaught|caught|all] <exception-class id>
                          -- cancel 'catch' for the specified exception
watch [access|all] <class id>.<field name>
                          -- watch access/modifications to a field
unwatch [access|all] <class id>.<field name>
   -- discontinue watching access/modifications to a field
trace methods [thread]    -- trace method entry and exit
untrace methods [thread]  -- stop tracing method entry and exit
step                      -- execute current line
step up
   -- execute until the current method returns to its caller
stepi                     -- execute current instruction
next                      -- step one line (step OVER calls)
cont                      -- continue execution from breakpoint

list [line number|method] -- print source code
use (or sourcepath) [source file path]
                          -- display or change the source path
exclude [class id ... | "none"]
   -- do not report step or method events for specified classes
classpath                 -- print classpath info from target VM

monitor <command>         -- execute command each time the program stops
monitor                   -- list monitors
unmonitor <monitor#>      -- delete a monitor
read <filename>           -- read and execute a command file

lock <expr>               -- print lock info for an object
threadlocks [thread id]   -- print lock info for a thread

pop
   -- pop the stack through and including the current frame
reenter               -- same as pop, but current frame is reentered
redefine <class id> <class file name>
                          -- redefine the code for a class

disablegc <expr>          -- prevent garbage collection of an object
enablegc <expr>           -- permit garbage collection of an object

!!                        -- repeat last command
<n> <command>             -- repeat command n times
help (or ?)               -- list commands
version                   -- print version information
exit (or quit)            -- exit debugger
```

```
<class id> or <exception-class id>: full class name with package
qualifiers or a pattern with a leading or trailing wildcard ('*')
NOTE: any wildcard pattern will be replaced by at most one full class
name matching the pattern.
<thread id>: thread number as reported in the 'threads' command
<expr>: a Java(tm) Programming Language expression.
Most common syntax is supported.

Startup commands can be placed in either "jdb.ini" or ".jdbrc"
in user.home or user.dir
```

Example jdb Session

```
[bashed]s0090668: jdb
Initializing jdb ...
> stop in ArrayDictionary.insertItem
Deferring breakpoint ArrayDictionary.insertItem.
It will be set after the class is loaded.
> run ArrayDictionary
run  ArrayDictionary
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: Set deferred breakpoint ArrayDictionary.insertItem

Breakpoint hit: "thread=main", ArrayDictionary.insertItem(),
   line=38 bci=0
38              int index = recFind(key,0,size-1);

main[1] step
>
Step completed: "thread=main", ArrayDictionary.recFind(), line=16 bci=0
16              if ( i1 > i2 )

main[1] cont
>
Exception occurred: java.lang.NullPointerException (uncaught)
   "thread=main", ArrayDictionary.insertItem(), line=39 bci=18
39              size = items.length + 1;

main[1] eval items
 items = null
main[1] quit
Exception in thread "main" java.lang.NullPointerException
```

bdbj

If you want to try out bdbj, my reverse execution debugger, then go to

```
http://www.dcs.ed.ac.uk/home/jjc/bdbj/bdbj-1.2.1/dl.html
```

The installation takes up about 5MB. You can get round problems with the source being too large for your quota by downloading bdbj into /tmp and then using the command

```
installbdbj-1.2.1dir /home/<your matric no.>/bdbj
```

Note that bdbj supports an older version of Java than the version which should be used for the CS2 course (see http://www.kaffe.org for details), and supports Swing 1.1.1 but not later versions.