7 Debugging

This section introduces the most important aspects of the debugging functionality in BlueJ. In talking to computing teachers, we have very often heard the comment that using a debugger in first year teaching would be nice, but there is just no time to introduce it. Students struggle with the editor, compiler and execution; there is no time left to introduce another complicated tool.

That's why we have decided to make the debugger as simple as possible. The goal is to have a debugger that you can explain in 15 minutes, and that students can just use from then on without further instruction. Let's see whether we have succeeded.

First of all, we have reduced the functionality of traditional debuggers to three tasks:

- setting breakpoints
- stepping through the code
- inspecting variables

In return, each of the three tasks is very simple. We will now try out each one of them.

To get started, open the project *debugdemo*, which is included in the *examples* directory in the distribution. This project contains a few classes for the sole purpose of demonstrating the debugger functionality – they don't make a lot of sense otherwise.

7.1 Setting breakpoints

Summary: To set a breakpoint, click in the breakpoint area to the left of the text in the editor.

Setting a breakpoint lets you interrupt the execution at a certain point in the code. When the execution is interrupted, you can investigate the state of your objects. It often helps you to understand what is happening in your code.

In the editor, to the left of the text, is the breakpoint area (Figure 16). You can set a breakpoint by clicking into it. A small stop sign appears to mark the breakpoint. Try this now. Open the class *Demo*, find the method *loop*, and set a breakpoint somewhere in the *for* loop. The stop sign should appear in your editor.

```
    public int loop(int count)
    {
        int sum = 17;

        for (int i=0; i<count; i++) {
            sum = sum + i;
            sum = sum - 2;
        }
        return sum;
    }
```

Figure 16: A breakpoint

Do this in the main method of class demo.

When the line of code is reached that has the breakpoint attached, execution will be interrupted. Let's try that now.

Create an object of class *Demo* and call the *loop* method with a parameter of, say, 10. As soon as the breakpoint is reached, the editor window pops up, showing the current line of code, and a debugger window pops up. It looks something like Figure 17.

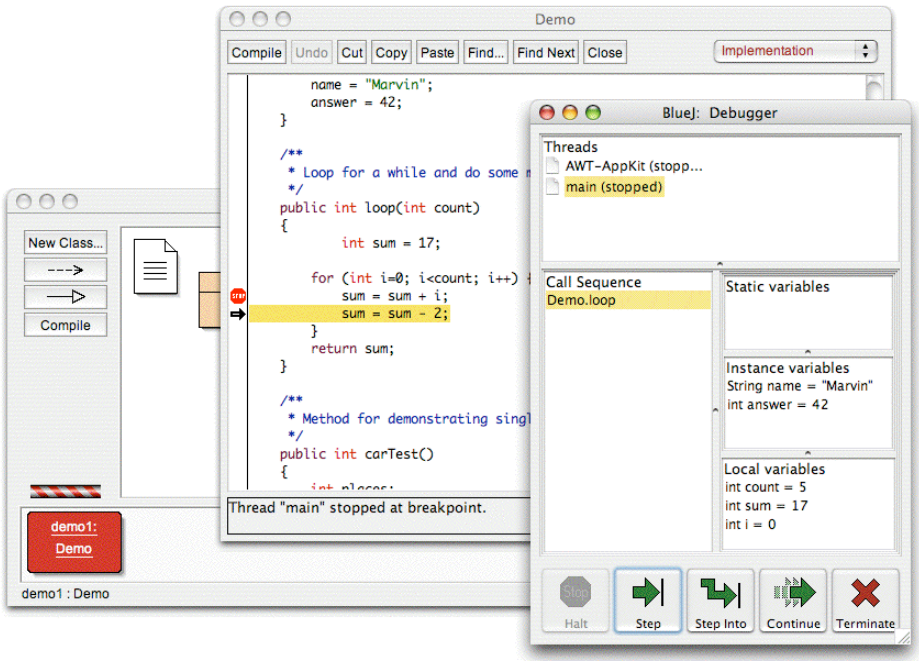

Figure 17: The debugger window

The highlight in the editor shows the line that will be executed next. (The execution is stopped *before* this line was executed.)

7.2 Stepping through the code

Summary: To single-step through your code, use the Step *and* Step Into *buttons in the debugger.*

Now that we have stopped the execution (which convinces us that the method really does get executed and this point in the code really does get reached), we can single-step through the code and see how the execution progresses. To do this, repeatedly click on the *Step* button in the debugger window. You should see the source line in the editor changing (the highlight moves with the line being executed). Every time you click the *Step* button, one single line of code gets executed and the execution stops again. Note also that the values of the variables displayed in the debugger window change (for example the value of *sum*.) So you can execute step by step and observe what happens. Once you get tired of this, you can click on the breakpoint again to remove it, and then click the *Continue* button in the debugger to restart the execution and continue normally.

Also do this in the main method

Let's try that again with another method. Set a breakpoint in class *Demo*, method *carTest()*, in the line reading

```
places = myCar.seats();
```

Call the method. When the breakpoint is hit, you are just about to execute a line that contains a method call to the method *seats()* in class *Car*. Clicking *Step* would step over the whole line. Let's try *Step Into* this time. If you *step into* a method call, then you enter the method and execute that method itself line by line (not as a single step). In this case, you are taken into the *seats()* method in class *Car*. You can now happily step through this method until you reach the end and return to the calling method. Note how the debugger display changes.

Step and *Step Into* behave identically if the current line does not contain a method call.

7.3 Inspecting variables

Summary: Inspecting variables is easy – they are automatically displayed in the debugger.

When you debug your code, it is important to be able to inspect the state of your objects (local variables and instance variables).

Doing it is trivial – most of it you have seen already. You do not need special commands to inspect variables; static variables, instance variables of the current object and local variables of the current method are always automatically displayed and updated.

You can select methods in the call sequence to view variables of other currently active objects and methods. Try, for example, a breakpoint in the *carTest()* method

again. On the left side of the debugger window, you see the call sequence. It currently shows

```
Car.seats
Demo.carTest
```

This indicates that `Car.seats` was called by `Demo.carTest`. You can select `Demo.carTest` in this list to inspect the source and the current variable values in this method.

If you step past the line that contains the `new Car(...)` instruction, you can observe that the value of the local variable `myCar` is shown as <*object reference*>. All values of object types (except for Strings) are shown in this way. You can inspect this variable by double-clicking on it. Doing so will open an object inspection window identical to those described earlier (section 4.1). There is no real difference between inspecting objects here and inspecting objects on the object bench.

7.4 Halt and terminate

Summary: Halt *and* Terminate *can be used to halt an execution temporarily or permanently.*

Sometimes a program is running for a long time, and you wonder whether everything is okay. Maybe there is an infinite loop, maybe is just takes this long. Well, we can check. Call the method *longloop()* from the *Demo* class. This one runs a while.

Now we want to know what's going on. Show the debugger window, if it is not already on screen.

Now click the *Halt* button. The execution is interrupted just as if we had hit a breakpoint. You can now step a couple of steps, observe the variables, and see that this is all okay – it just needs a bit more time to complete. You can just *Continue* and *Halt* several times to see how fast it is counting. If you don't want to go on (for example, you have discovered that you really are in an infinite loop) you can just hit *Terminate* to terminate the whole execution. Terminate should not be used too frequently – you can leave perfectly well written objects in an inconsistent state by terminating the machine, so it is advisable to use it only as an emergency mechanism.