Debugging in Eclipse

We are built to make mistakes, coded for error.

Lewis Thomas

It is one thing to show a man that he is in error, and another to put him in possession of the truth.

John Locke

Prerequisites for Eclipse

To use Eclipse you must have an installed version of the Java Runtime Environment (JRE).

The latest version is available from **java.com**.

Since Eclipse includes its own Java compiler, it is not strictly necessary to have a version of the Java Development Kit (JDK) installed on your computer.

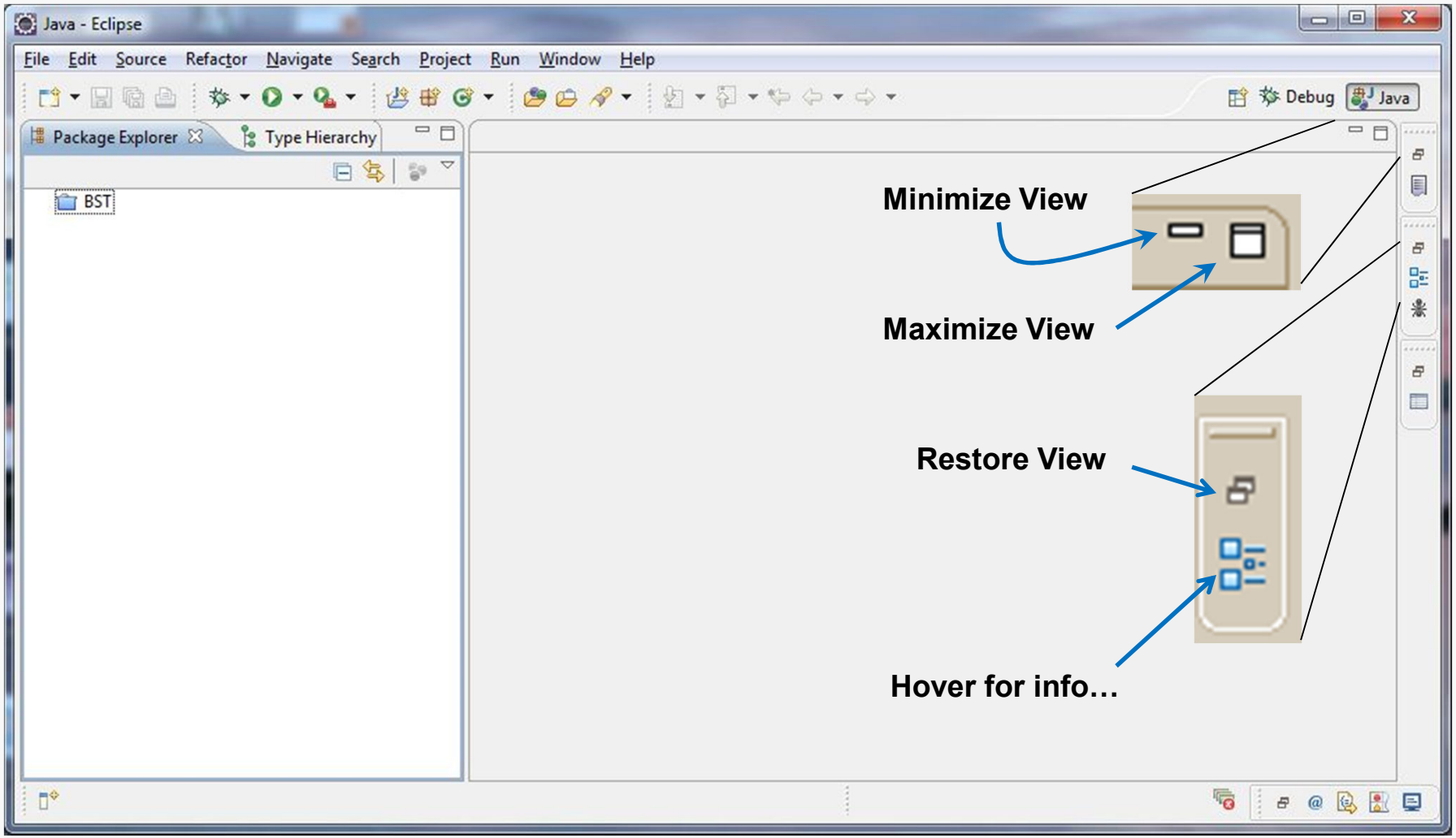
However, I recommend installing one anyway so that you can test your code against the "real" Java compiler.

The latest version is available from: **www.oracle.com/technetwork/java/**

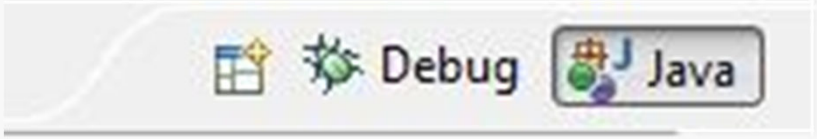
If you install the JDK, I recommend putting it in a root-level directory, and making sure there are no spaces in the pathname for the directory.

Eclipse Workbench

The initial Eclipse Workbench (my configuration):



Eclipse Java Perspective Toolbar



Choose a Perspective

New Project / Save / Save All / Print

Build Project

Start Debugging + configurations

Run Project + configurations

Run Last Tool + configurations

Eclipse Java Perspective Toolbar

New Java Project / Package / Class

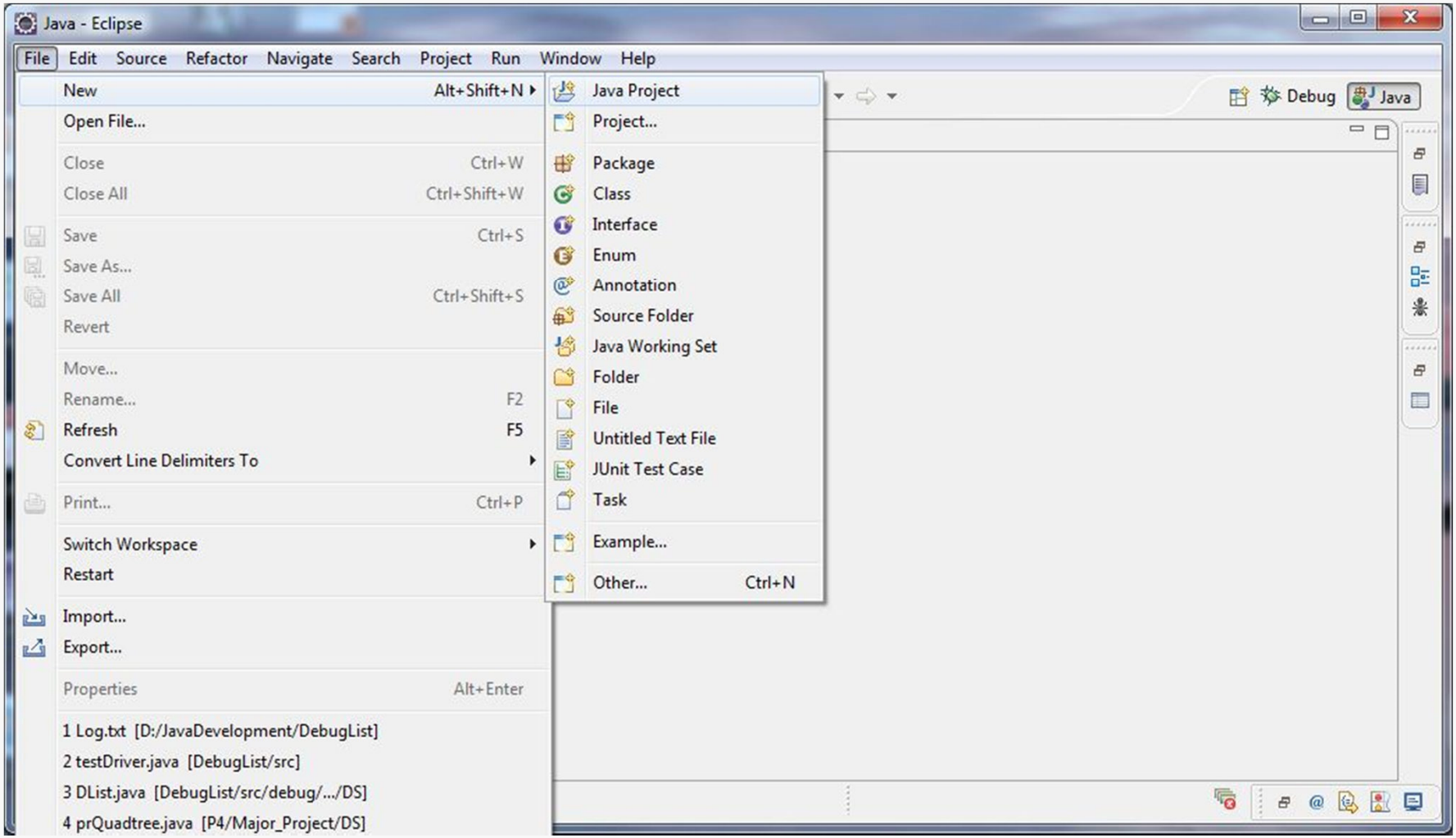
Open Type / Open Task / Search + options

Go to last edit location

Back/Next + more navigation options

Creating a New Java Project

In the Workbench, select **File/New/Java Project**:



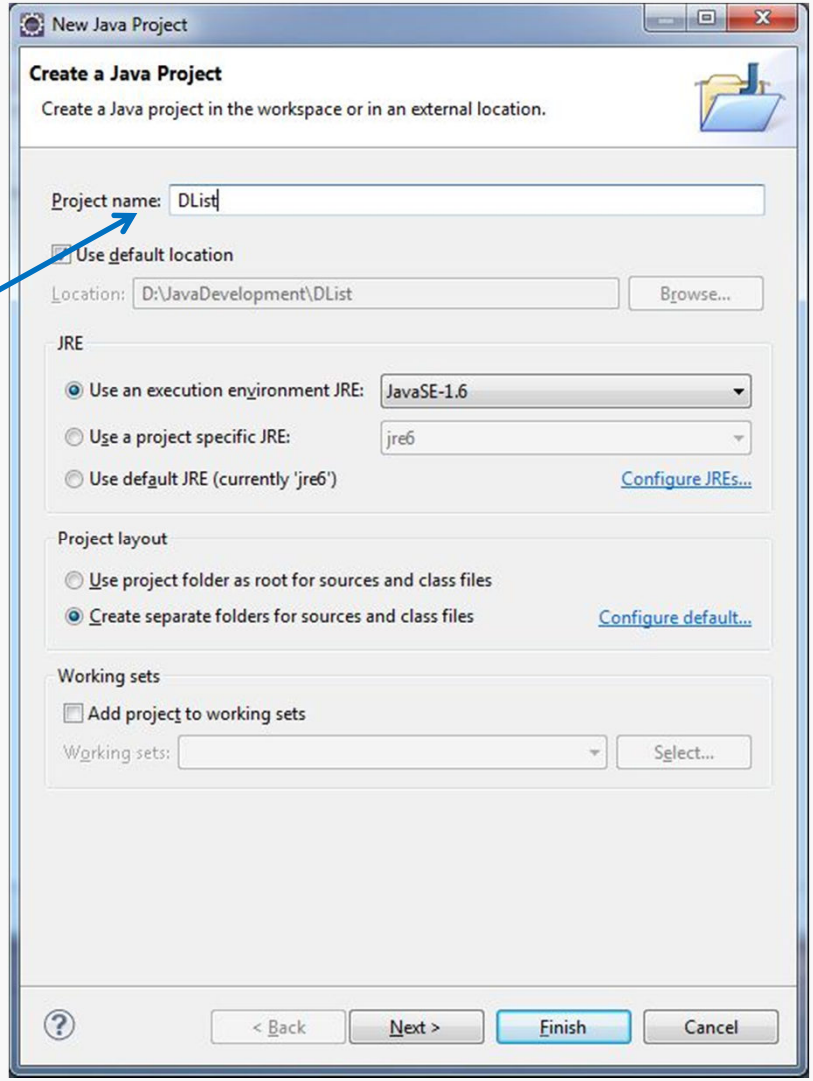
Creating a New Java Project

In the resulting dialog box:

Enter a name for the Project.

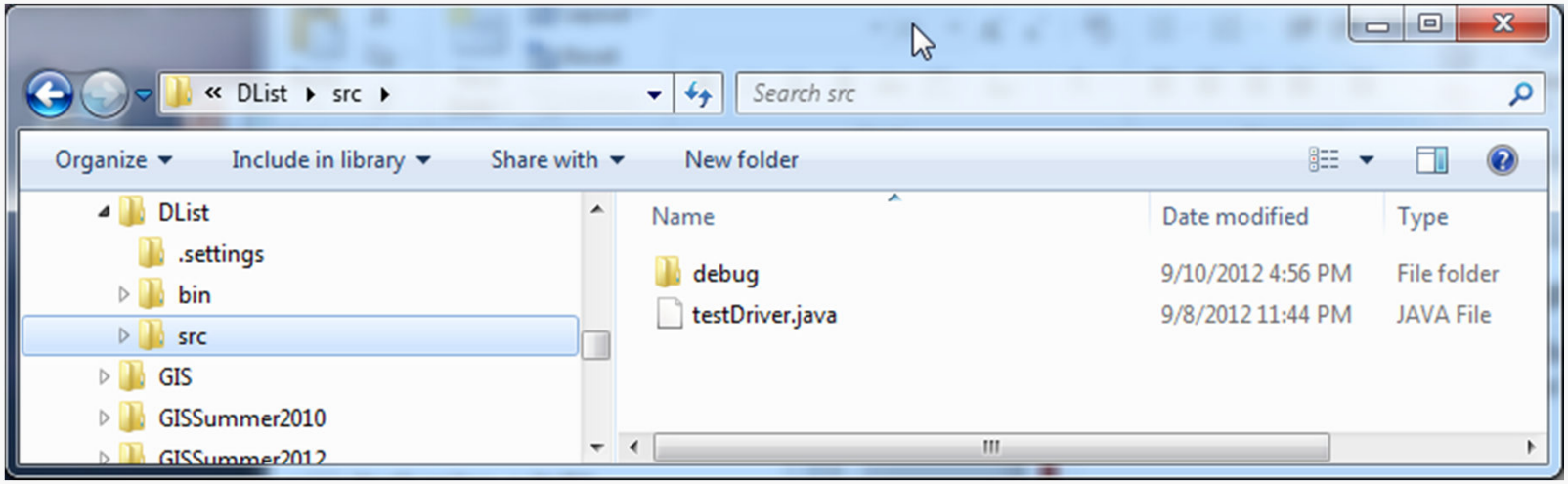
For now, just take the defaults for the remaining options.

Click **Next** and then **Finish** in the next dialog.



Adding Source for the DList Example

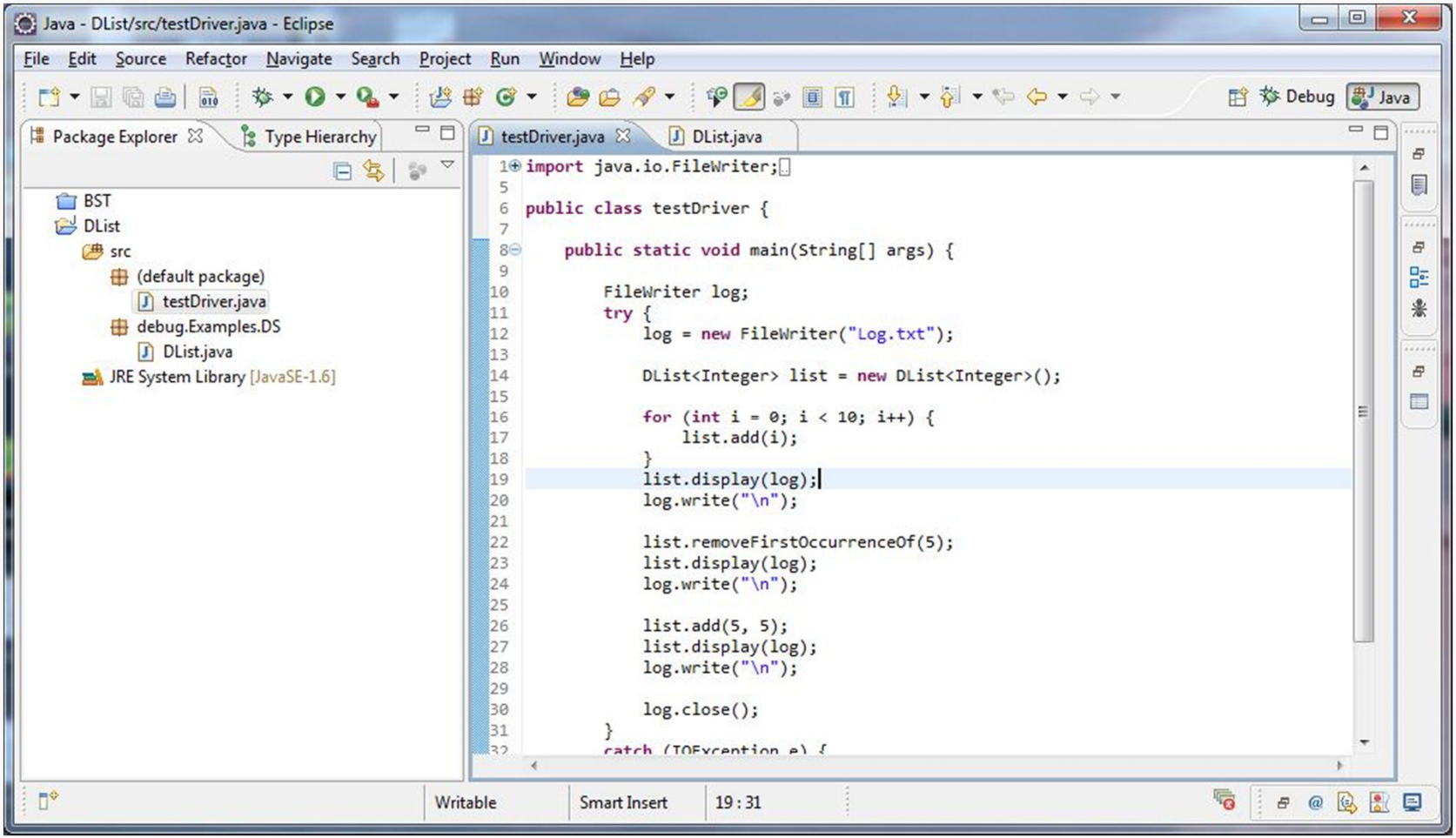
Download the file **DListExample.zip** from the course website Resources page, and place the contents into the **src** directory for the Eclipse project you just created:



Performing a Build

Back in Eclipse, right-click on the project icon for **DList** and select **Refresh**…

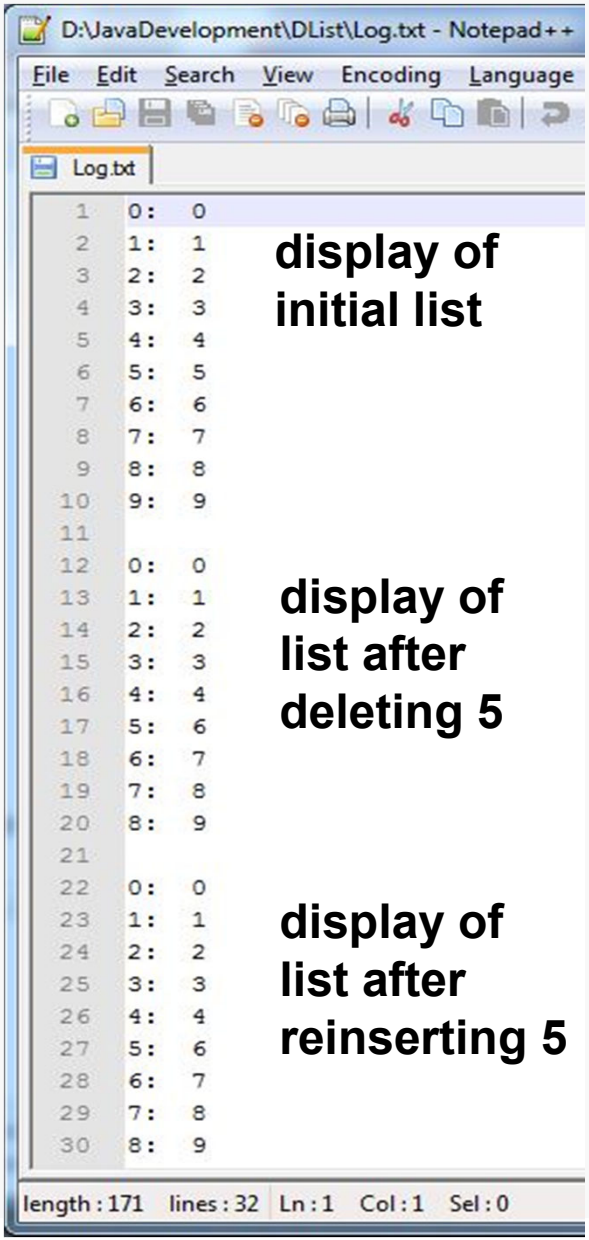
Use the **Project** menu or click on the **Build All** button ( ) to compile the code.



Running the Program

To execute the program, click on the Run button ( ).

As indicated by the source code, the test driver writes its output to a file named **Log.txt**:



Unfortunately, there appears to be an error; the value 5 should have been added to the list and appear in the final listing of the contents… it's not there.

Controlling Execution

Now, we have some clues about the error:

- The list appears to be OK after the first **for** loop completes; that doesn't indicate any problems with the **add()** method called there.
- The list appears to be OK after the call to the **removeFirstOccurrenceOf()** method; that doesn't indicate any problems there.
- The list is missing an element after the call to the second **add()** method; that seems to indicate the problem lies there…

It would be useful to be able to run the program to a certain point, check the state of the list (and perhaps other variables), and then step carefully through the subsequent execution, watching just how things change.

Fortunately, Eclipse provides considerable support for doing just that.

Kinds of Breakpoints

A *breakpoint* marks a location or condition under which we want the program's execution to be suspended.

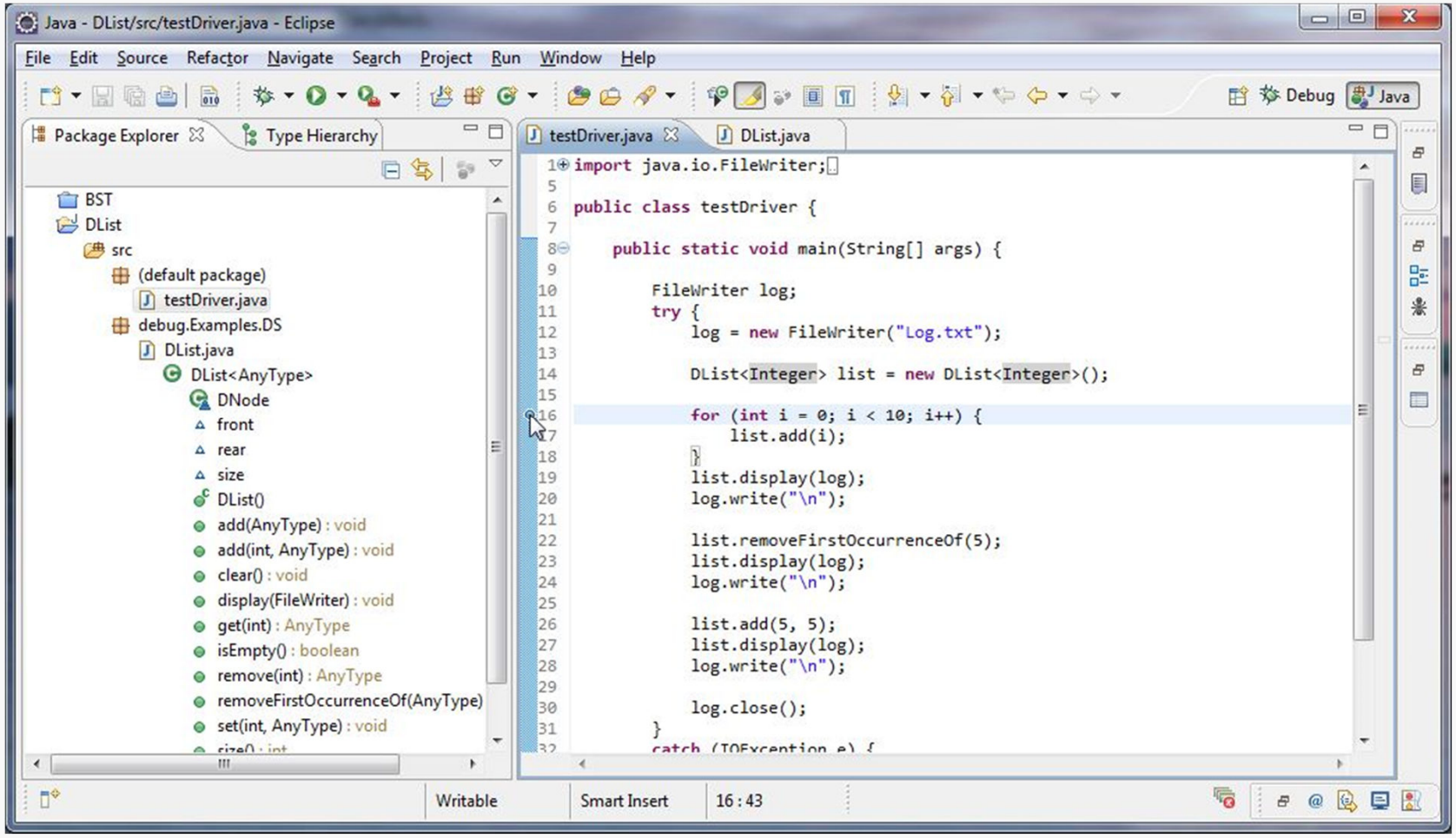
Eclipse supports setting four kinds of breakpoints:

line breakpoint	halt when execution reaches a specific statement
method breakpoint	halt when execution enters/exits a specific method
expression breakpoint	halt when a user-defined condition becomes true, or changes value
exception breakpoint	halt when a particular Java exception occurs (caught or not)

Setting a Line Breakpoint

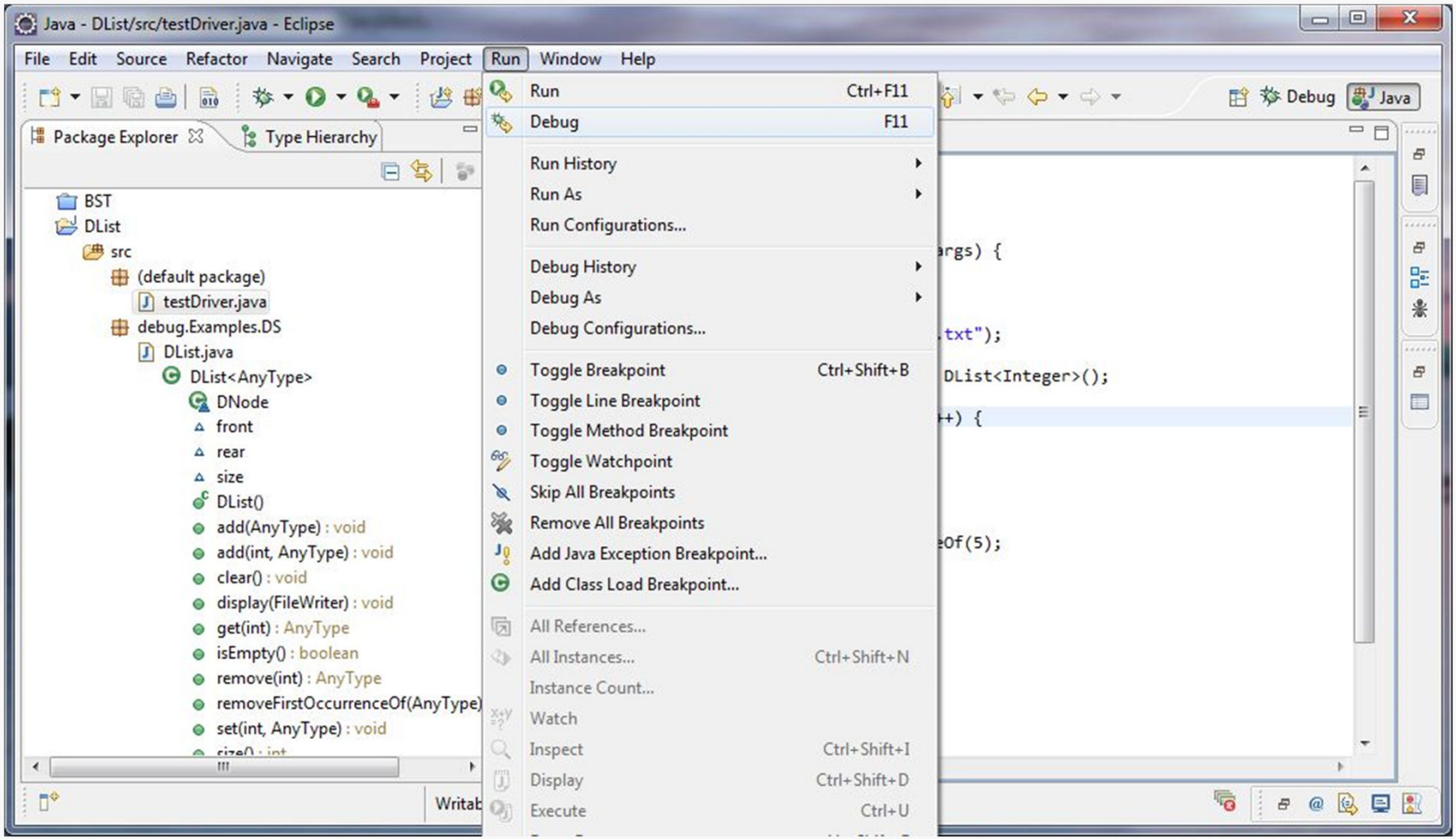
line breakpoint halt when execution reaches a specific statement

To set one, just double-click in the editor margin next to the selected line of code:



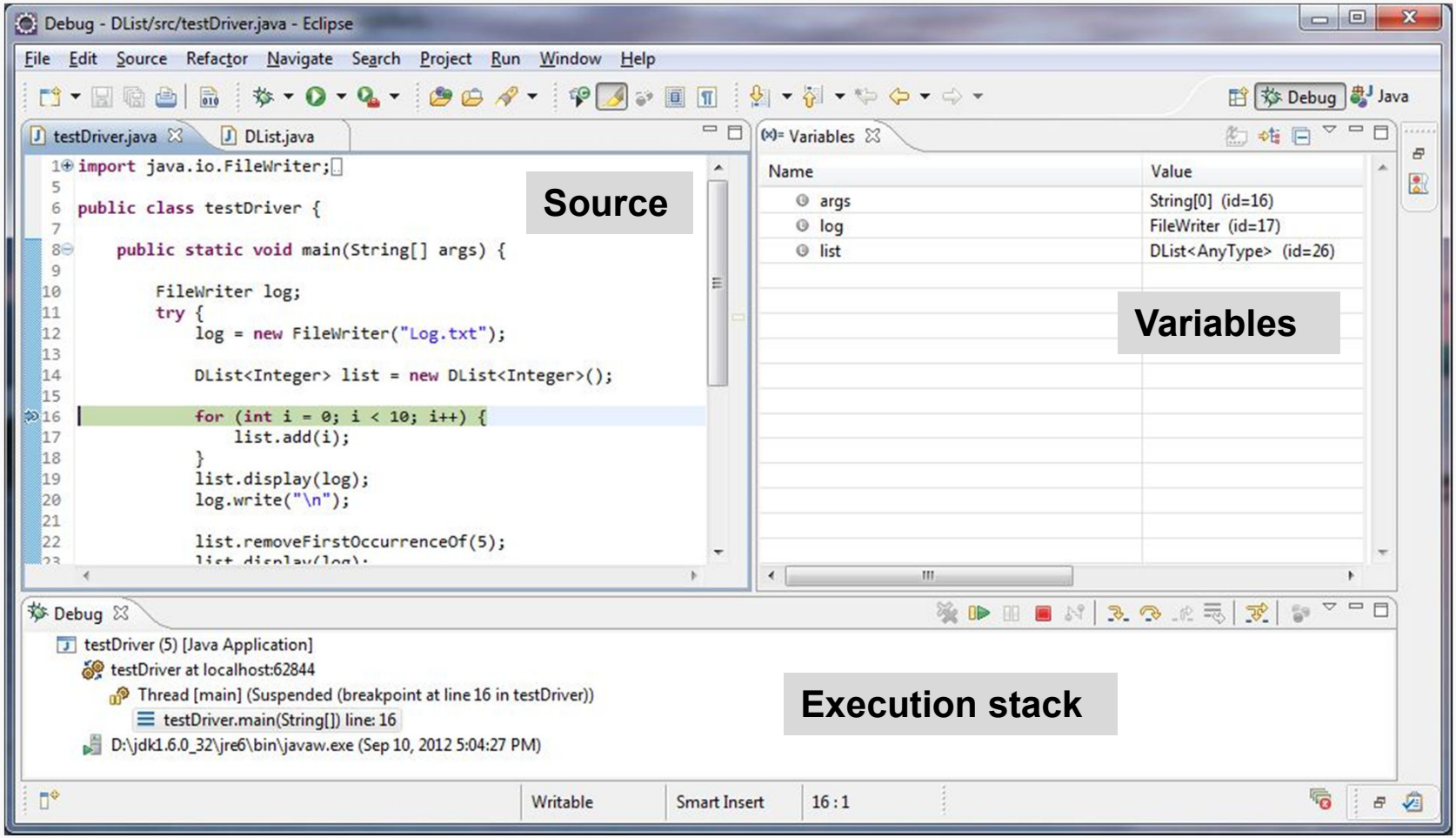
Running to a Breakpoint

Go to the **Run** menu and select **Debug** (or use the keyboard shortcut **F11**):



Debug Perspective

This opens the Debug Perspective:

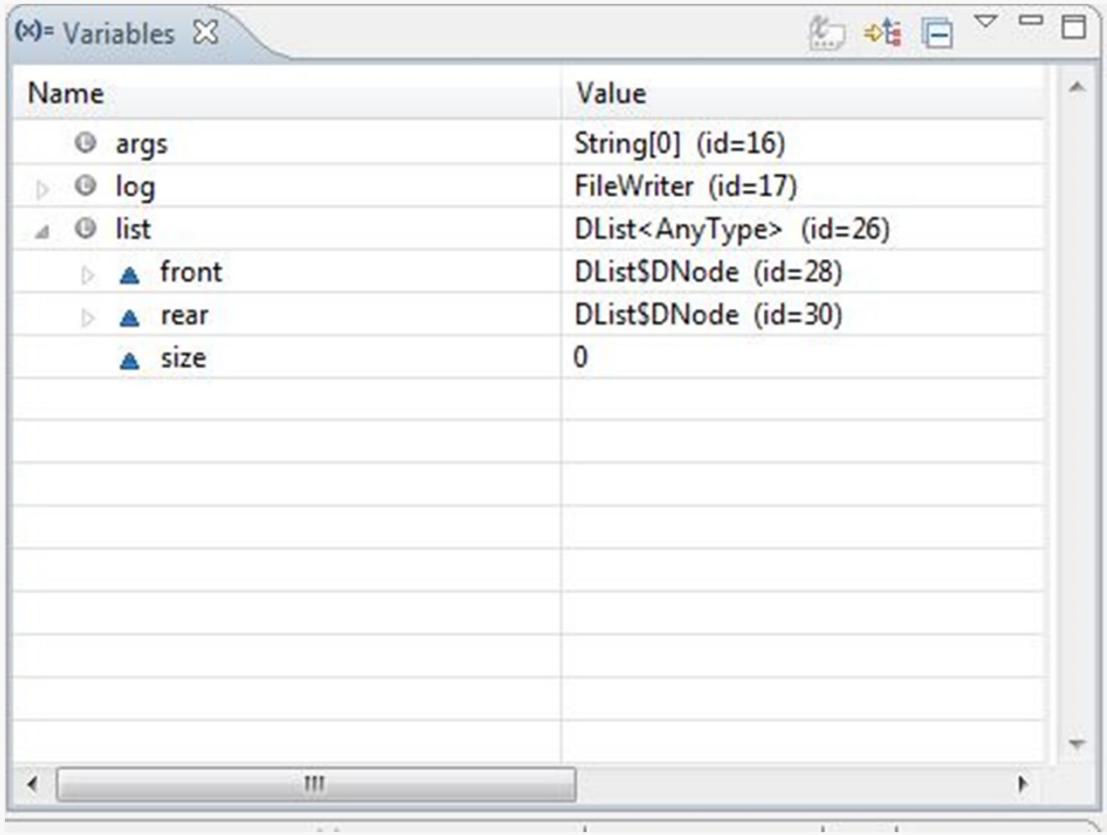



```
1  import java.io.FileWriter;
5
6  public class testDriver {
7
8      public static void main(String[] args) {
9
10         FileWriter log;
11         try {
12             log = new FileWriter("Log.txt");
13
14             DList<Integer> list = new DList<Integer>();
15
16             for (int i = 0; i < 10; i++) {
17                 list.add(i);
18             }
19             list.display(log);
20             log.write("\n");
21
22             list.removeFirstOccurrenceOf(5);
```

You may see a different window layout; feel free to close other Views, like Outline if they are visible.

Using the Variables View

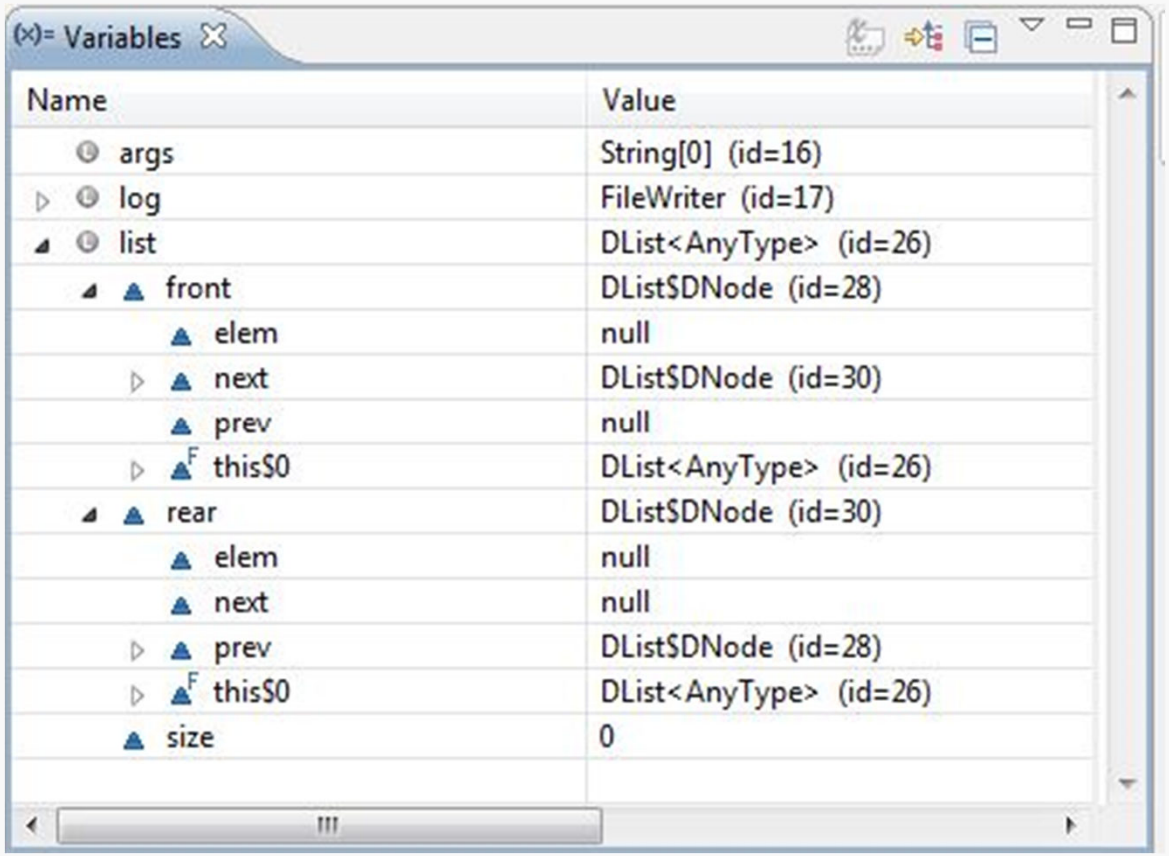
At this point, the list constructor has run… let's examine the structure:

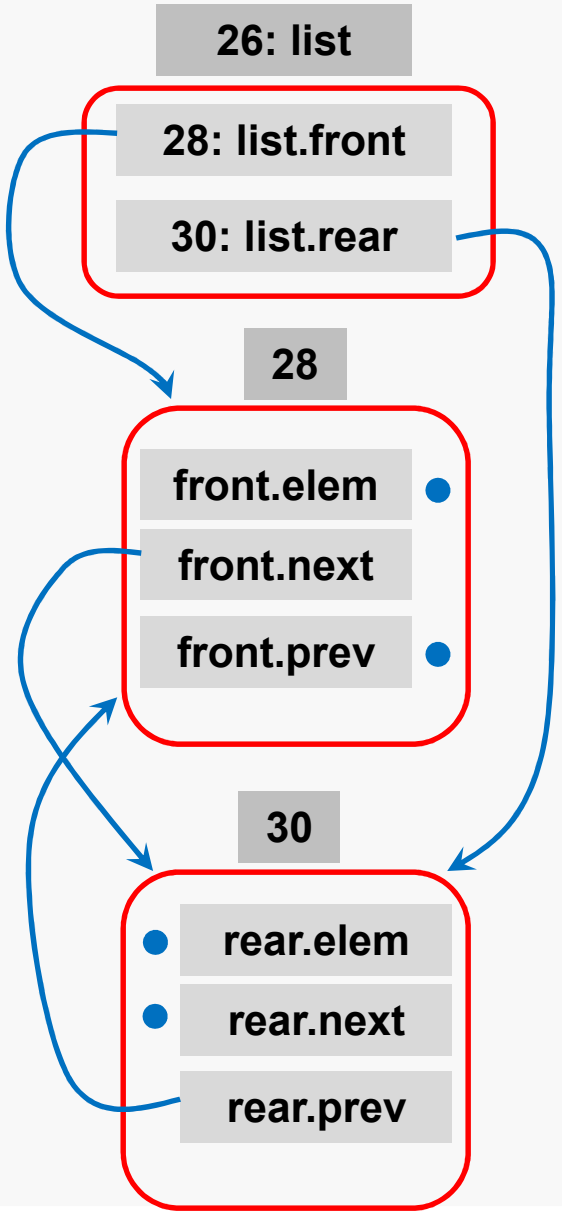


Objects are assigned unique IDs as they are created; these allow us to infer the physical structure…

Using the Variables View

Examine the values of the fields of **front** and **rear**:





OK, that looks just fine… two guard nodes pointing at each other, neither holding a data value.

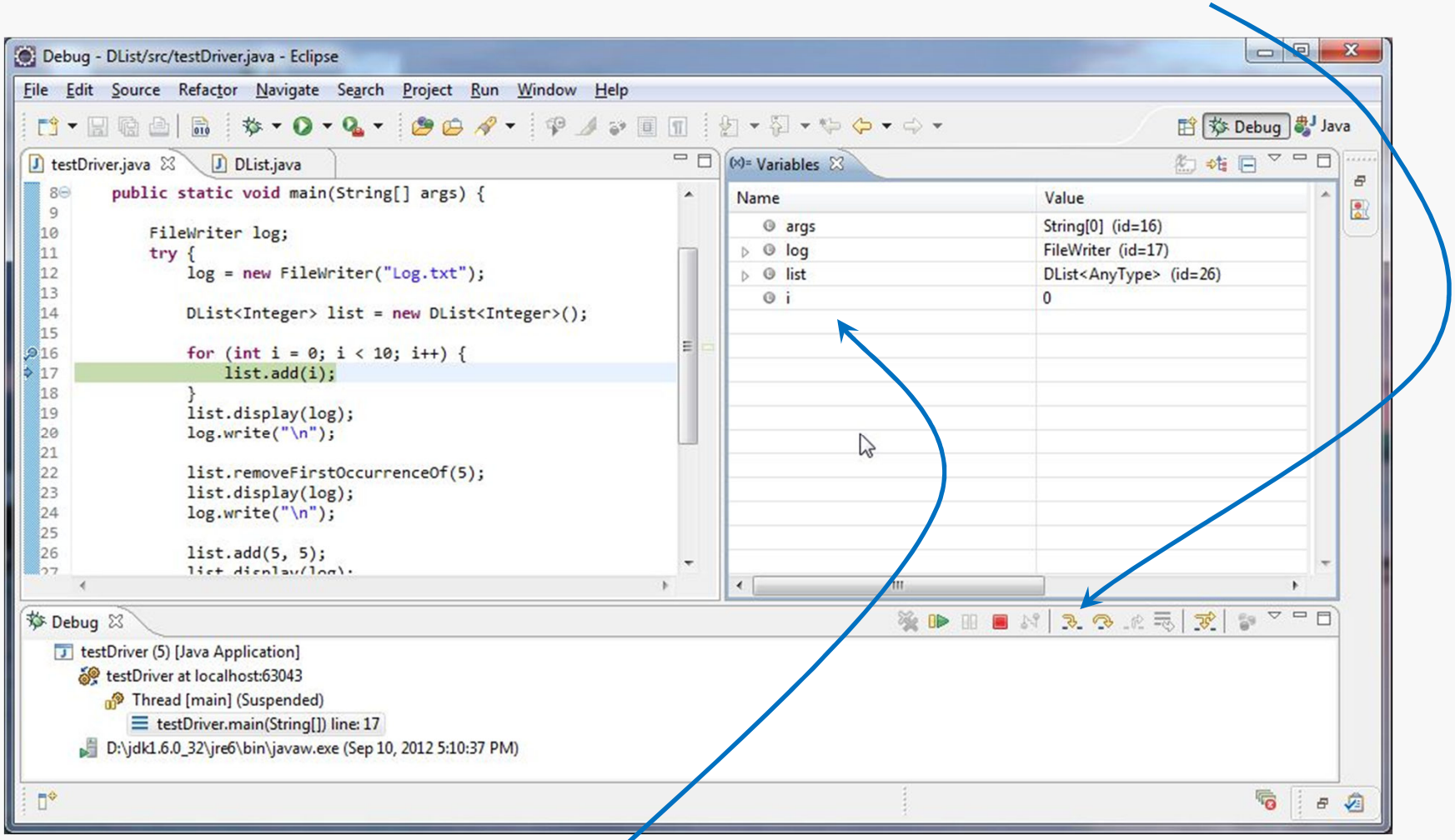
The Debug Toolbar

1 2 3 4 5 6 7 8 9 10

1. **Resume** – Continues execution until breakpoint or thread ends
2. **Suspend** – Interrupts a running thread
3. **Terminate** – Ends the execution of the selected thread
4. **Disconnect** – Disconnect from a remote debugging session
5. **Remove terminated launches** – Closes all terminated debug sessions
6. **Step Into** – Steps into a method and executes its first line of code
7. **Step Over** – Executes the next line of code in the current method
8. **Step Return** – Continues execution until the end of the current method (until a return)
9. **Drop to Frame** – Returns to a previous stack frame
10. **Step with Filters** – Continues execution until the next line of code which is not filtered out

Step-by-step Execution

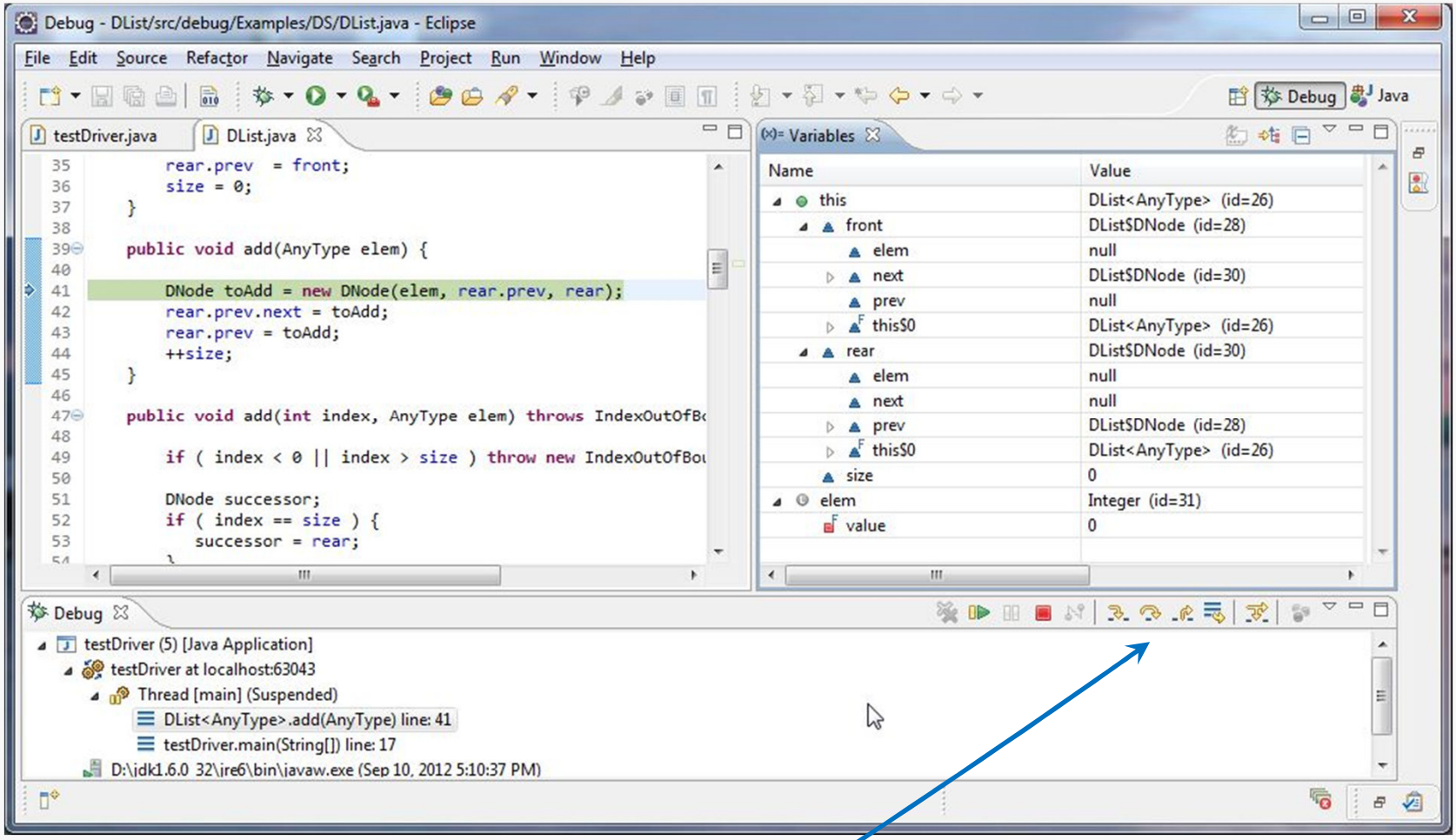
For illustration, we'll examine the insertion of the first data node, step by step:



Note the appearance of the variable **i** and its value.

Step-by-step Execution

Click the **step-into** button again; now we'll enter the call to **add()**:



Now, I don't really want to trace the constructor, much less the call to **new**, so this time I'll click the **step-over** button…

Step-over versus Step-into

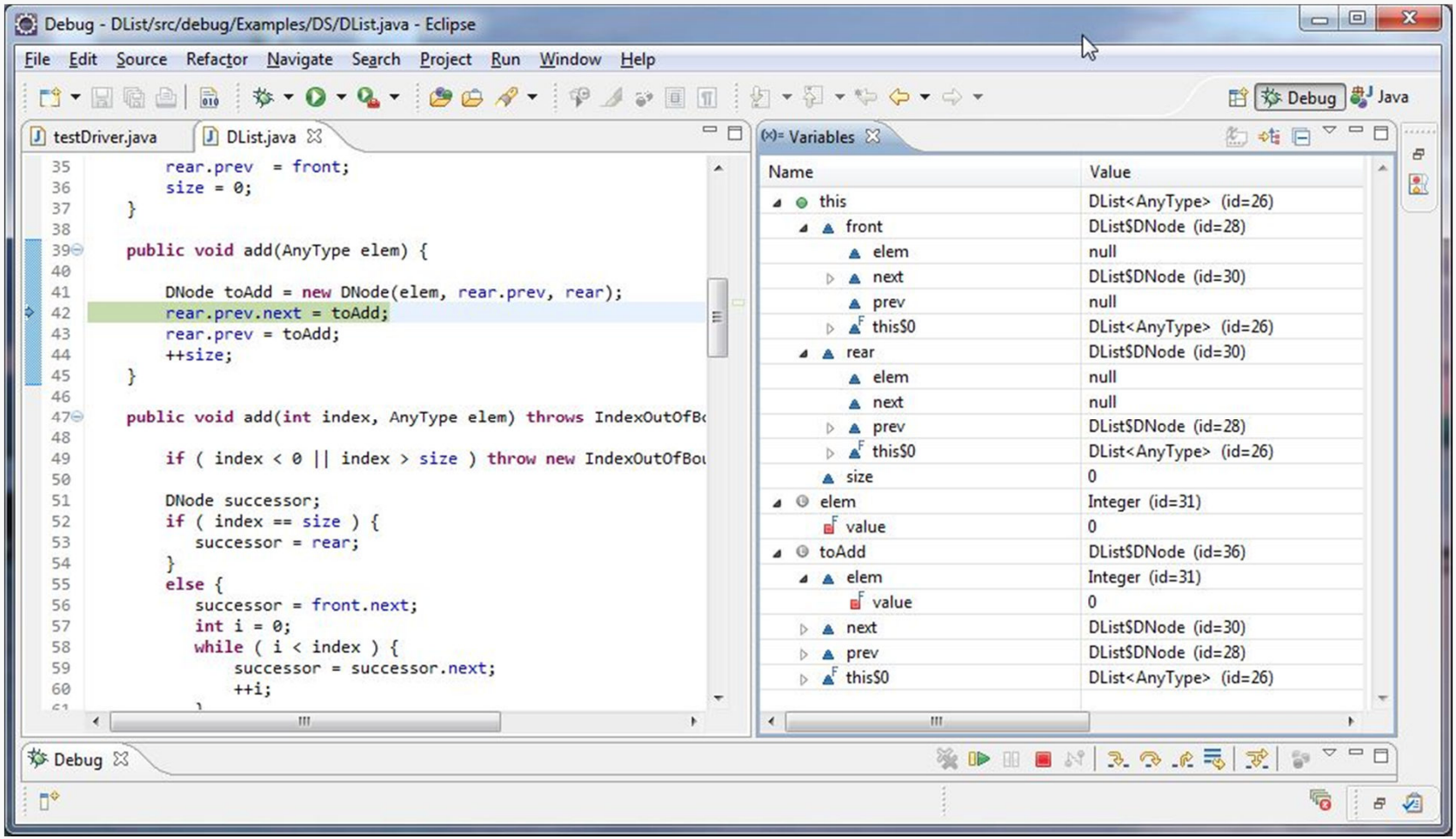
The difference is that if you are executing a method call (or invoking `new`, for example) in the current statement:

step-into	takes you into the implementation of that method
step-over	calls the method, but does not step you through its execution

Both are useful… step-into is frustrating when system code is involved.

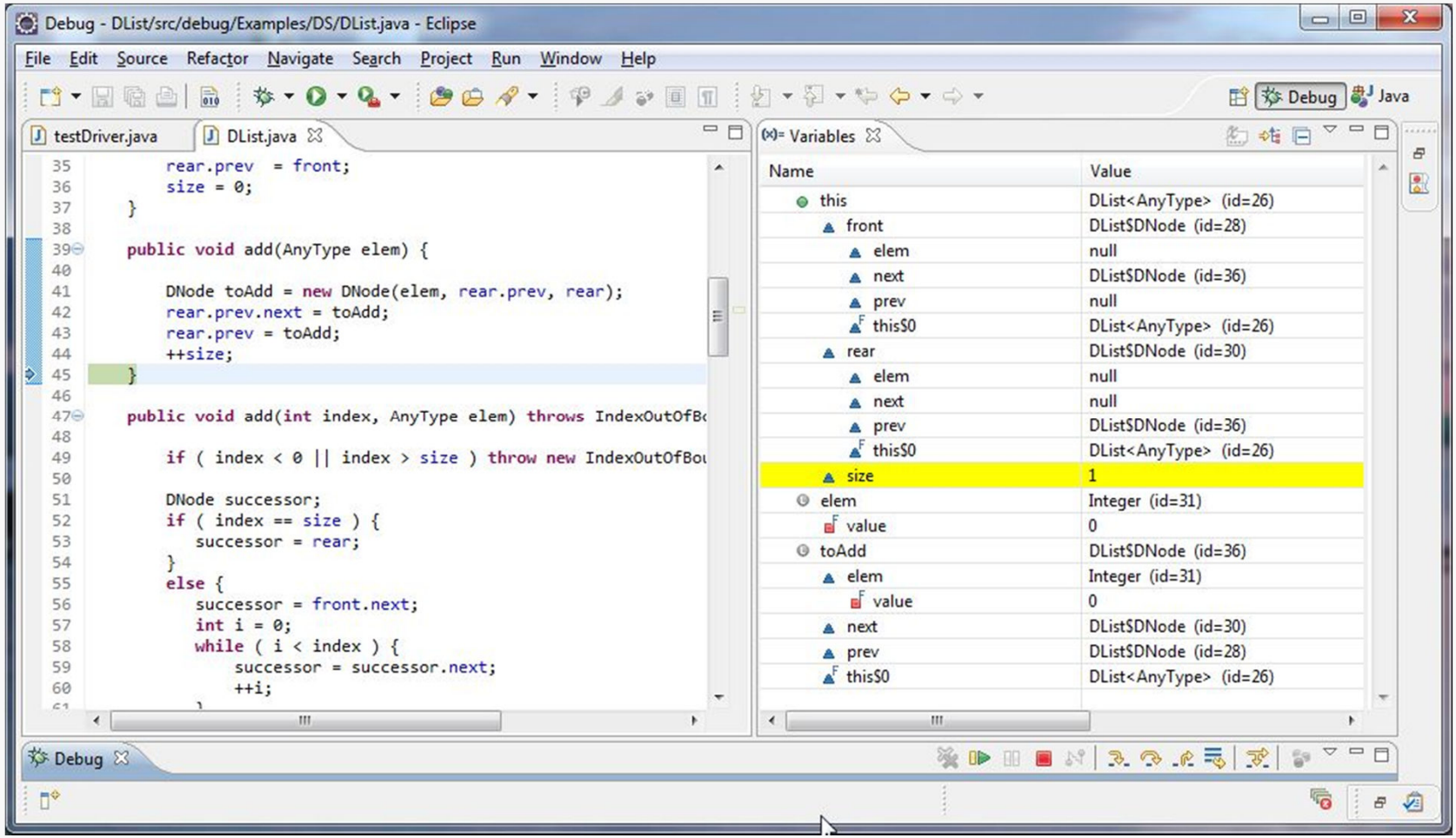
Step-by-step Execution

So, we see that the needed node has been properly initialized:



Step-by-step Execution

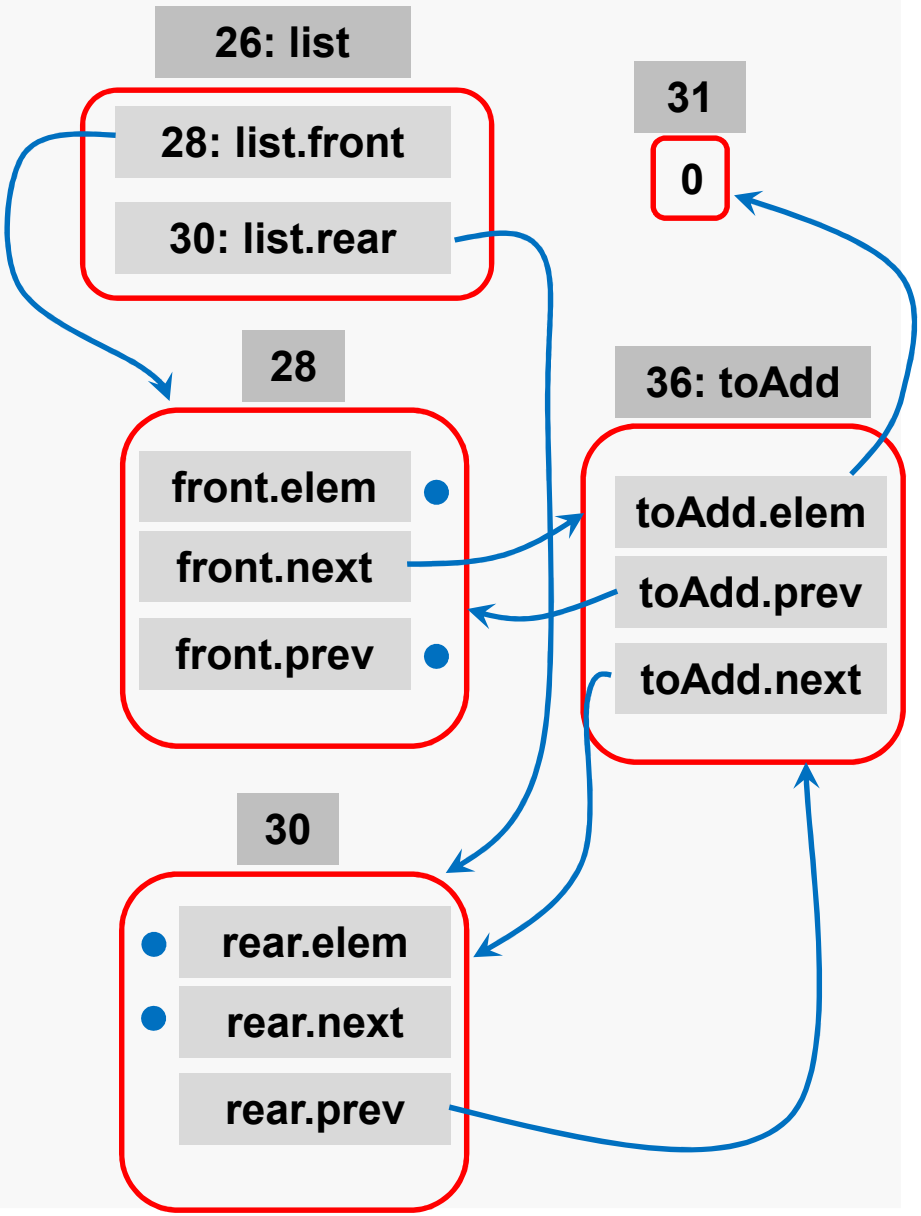
Three clicks on **step-over** (or **step-into**) bring us to this point:



Checking the List Structure

Variables

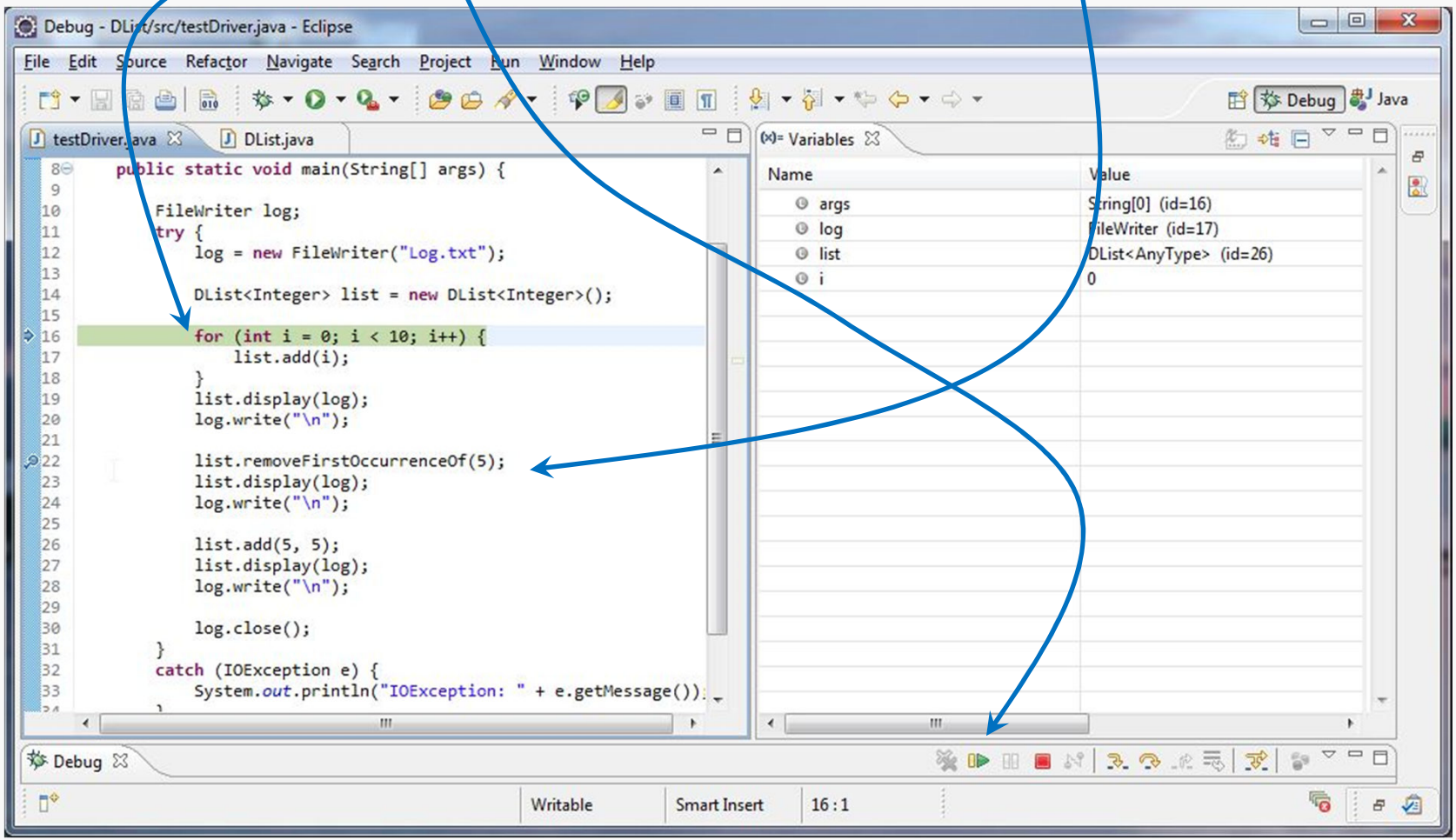
Name	Value
this	DList<AnyType> (id=26)
front	DList$DNode (id=28)
elem	null
next	DList$DNode (id=36)
prev	null
this$0	DList<AnyType> (id=26)
rear	DList$DNode (id=30)
elem	null
next	null
prev	DList$DNode (id=36)
this$0	DList<AnyType> (id=26)
size	1
elem	Integer (id=31)
value	0
toAdd	DList$DNode (id=36)
elem	Integer (id=31)
value	0
next	DList$DNode (id=30)
prev	DList$DNode (id=28)
this$0	DList<AnyType> (id=26)



Well, that looks OK.

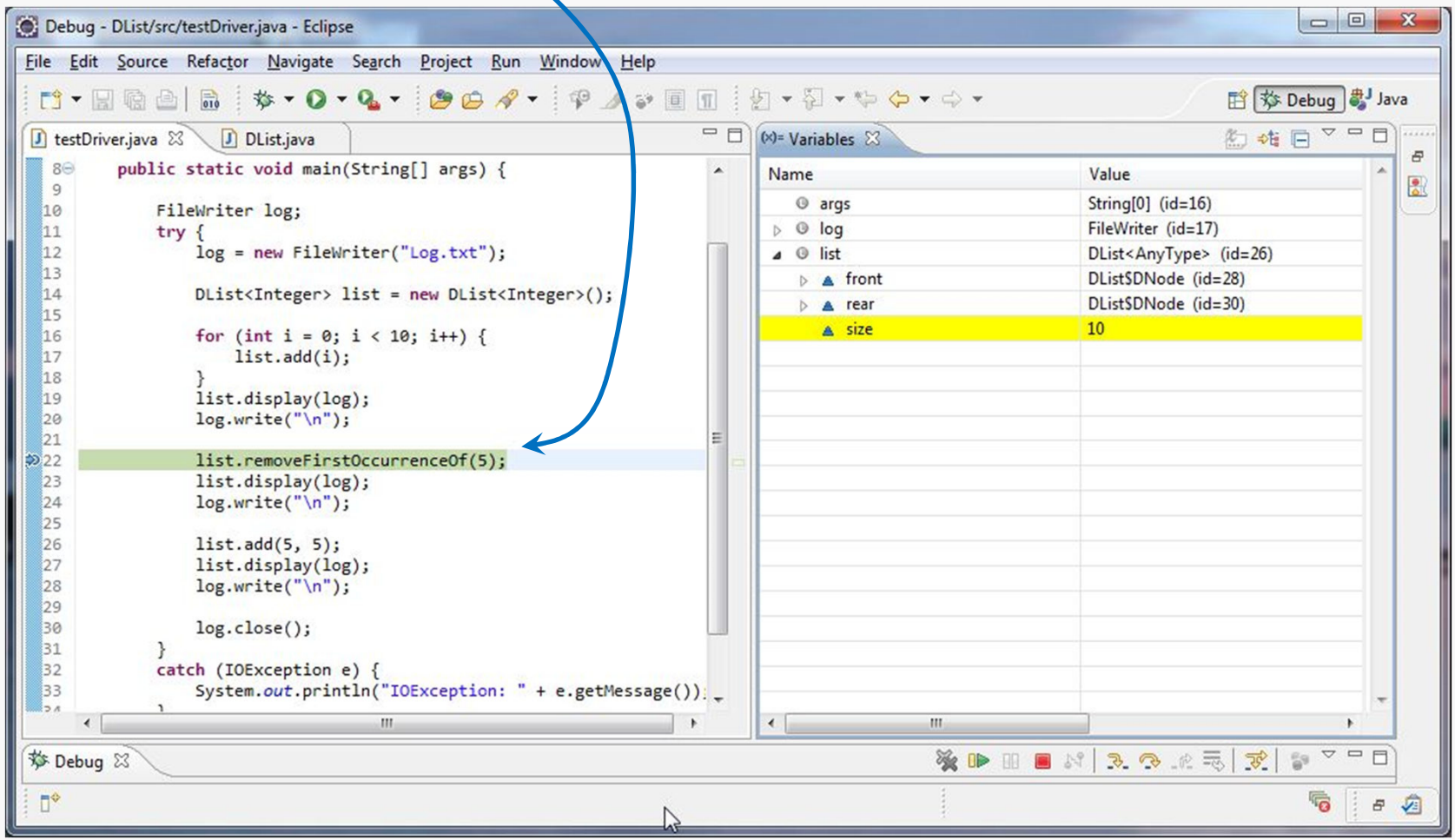
Resetting Breakpoints and Resuming

OK, we've confirmed that the first data node is inserted properly; now we can remove the breakpoint at the **for** loop, and set one at the call to the **removeFirstOccurrenceOf()** method, and then click **Resume** to continue execution:

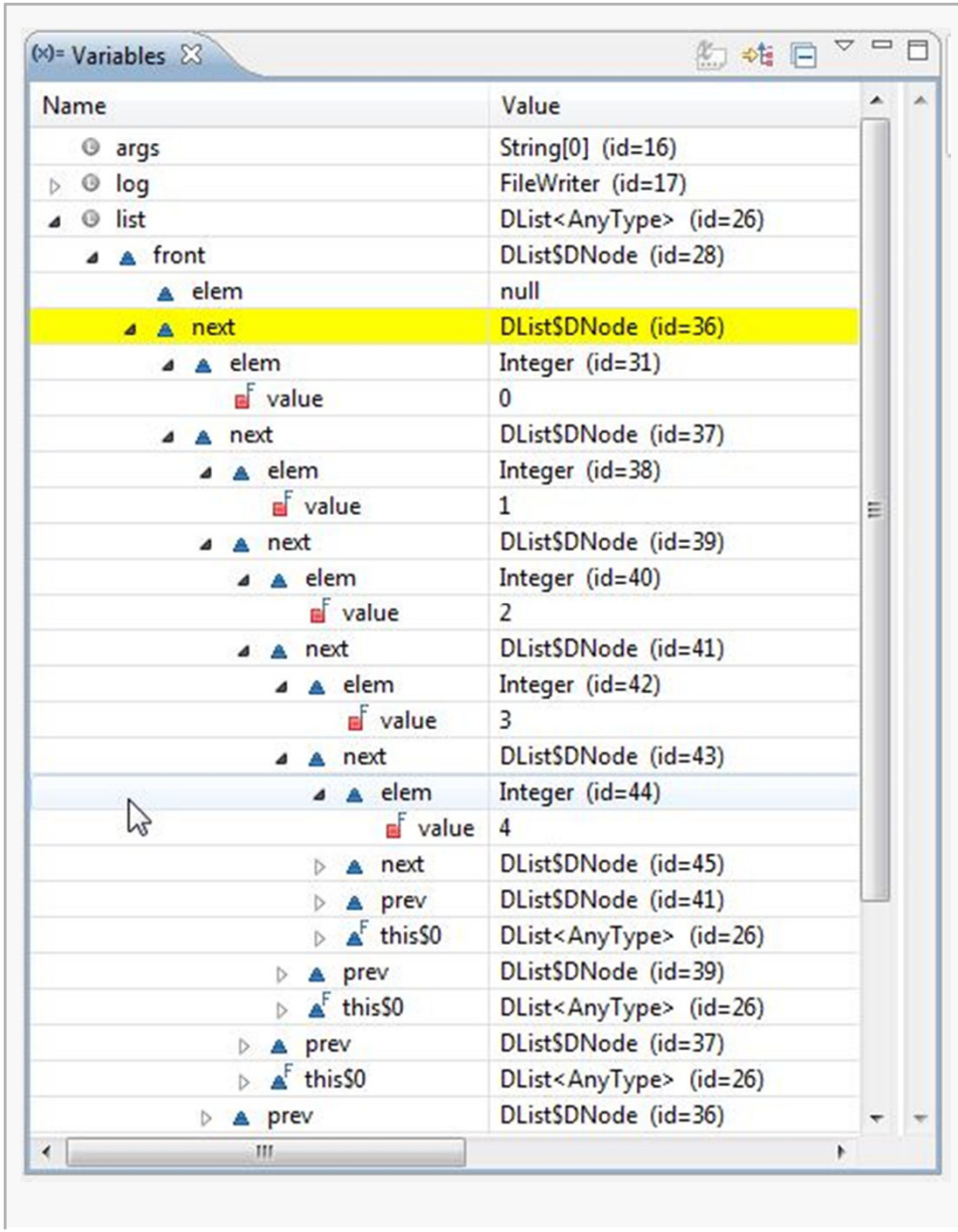


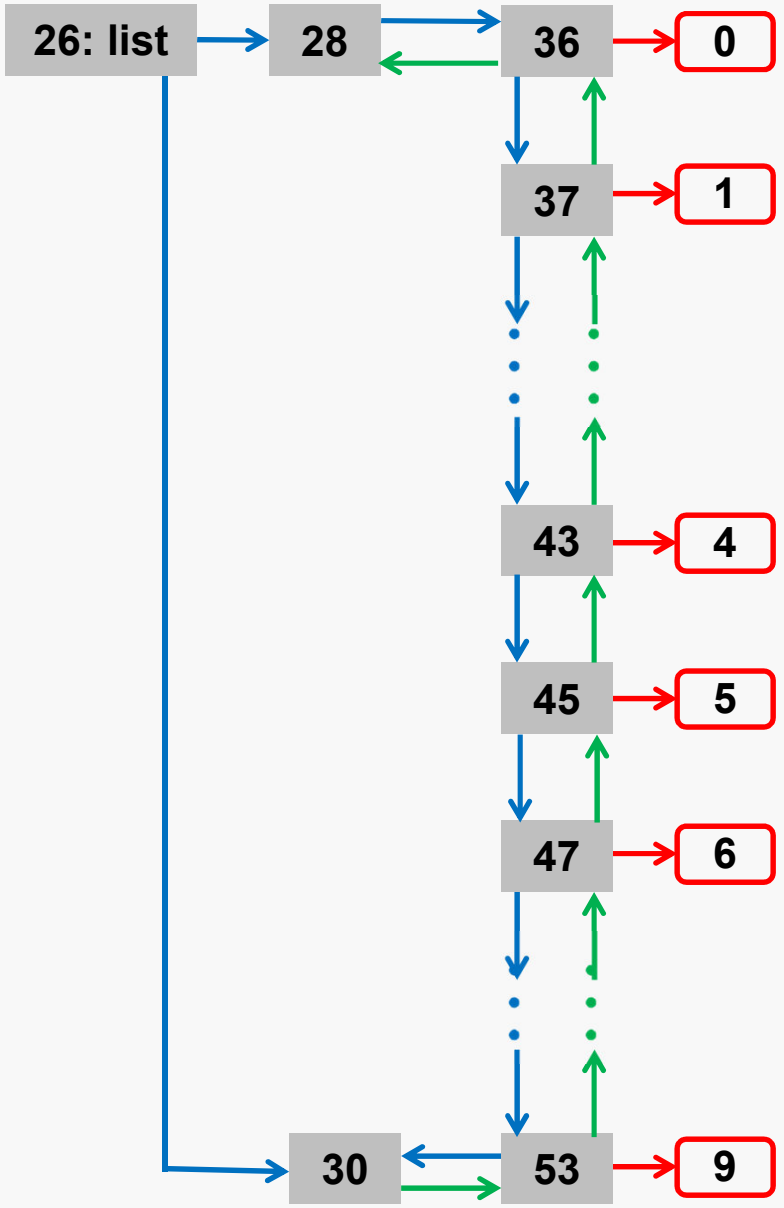
After Resuming… the List is Constructed

Execution proceeds to the new breakpoint:



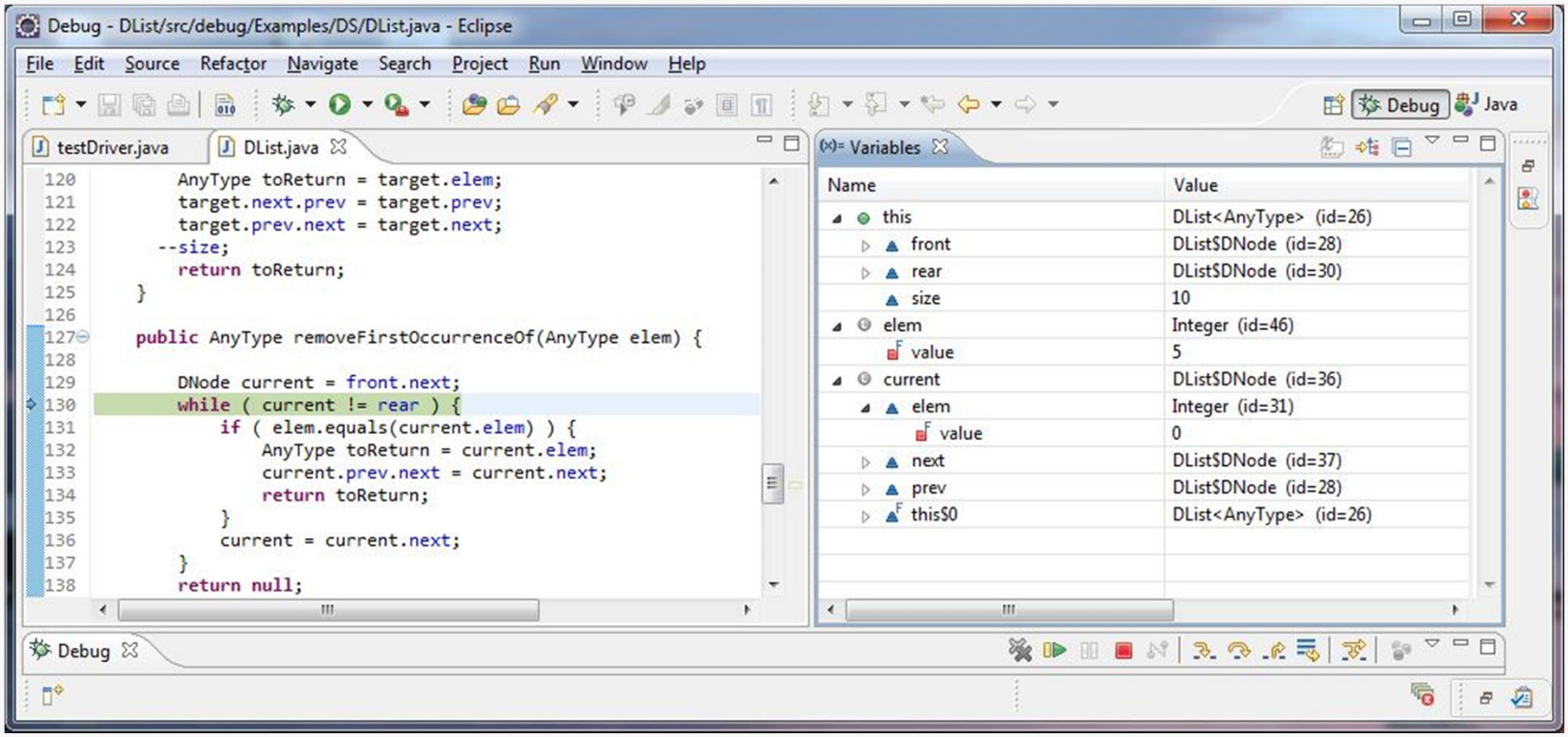
Complete List Structure





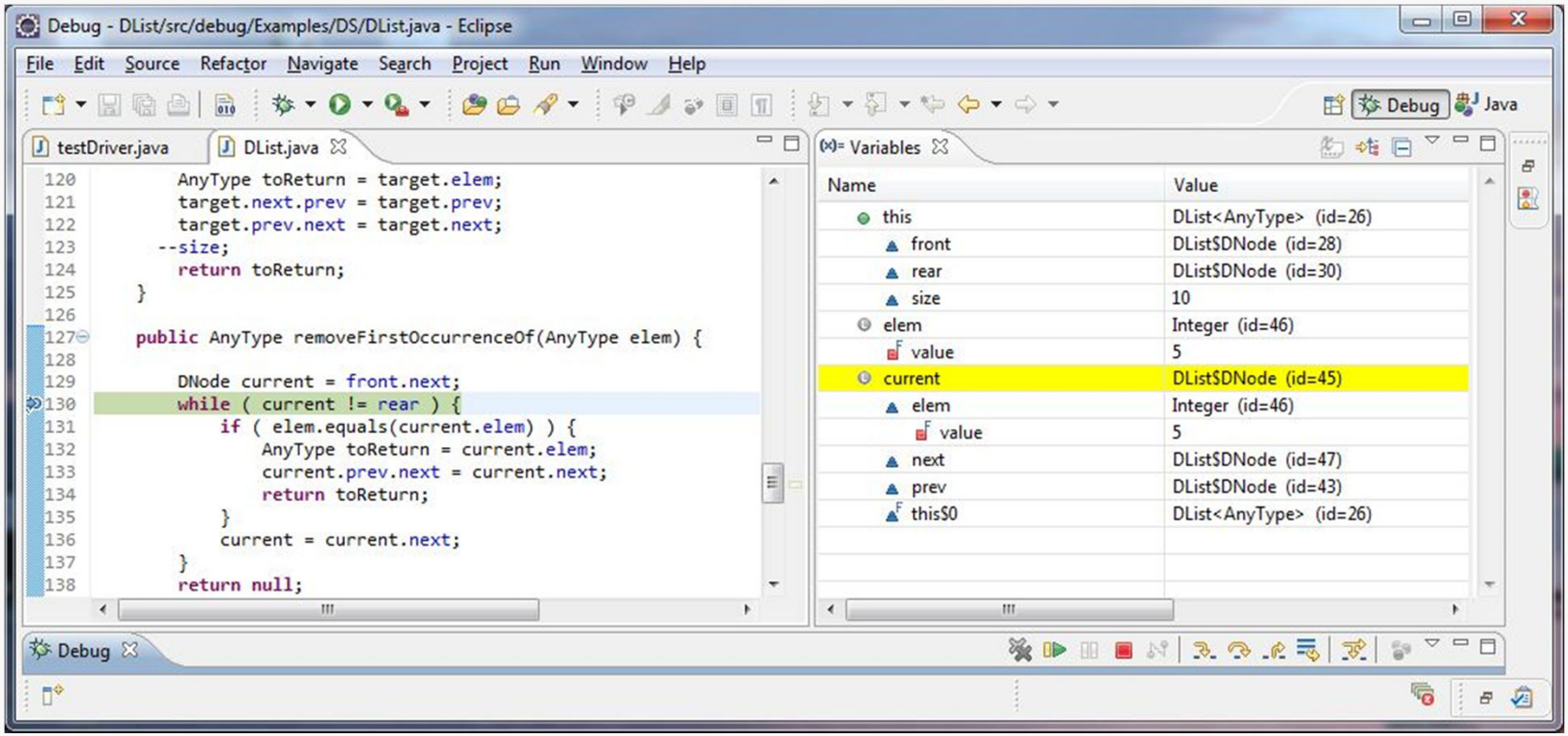
Step Into removeFirstOccurrenceOf()

Use **step-into** and proceed to the **while** loop that will walk to the first occurrence of the target value:



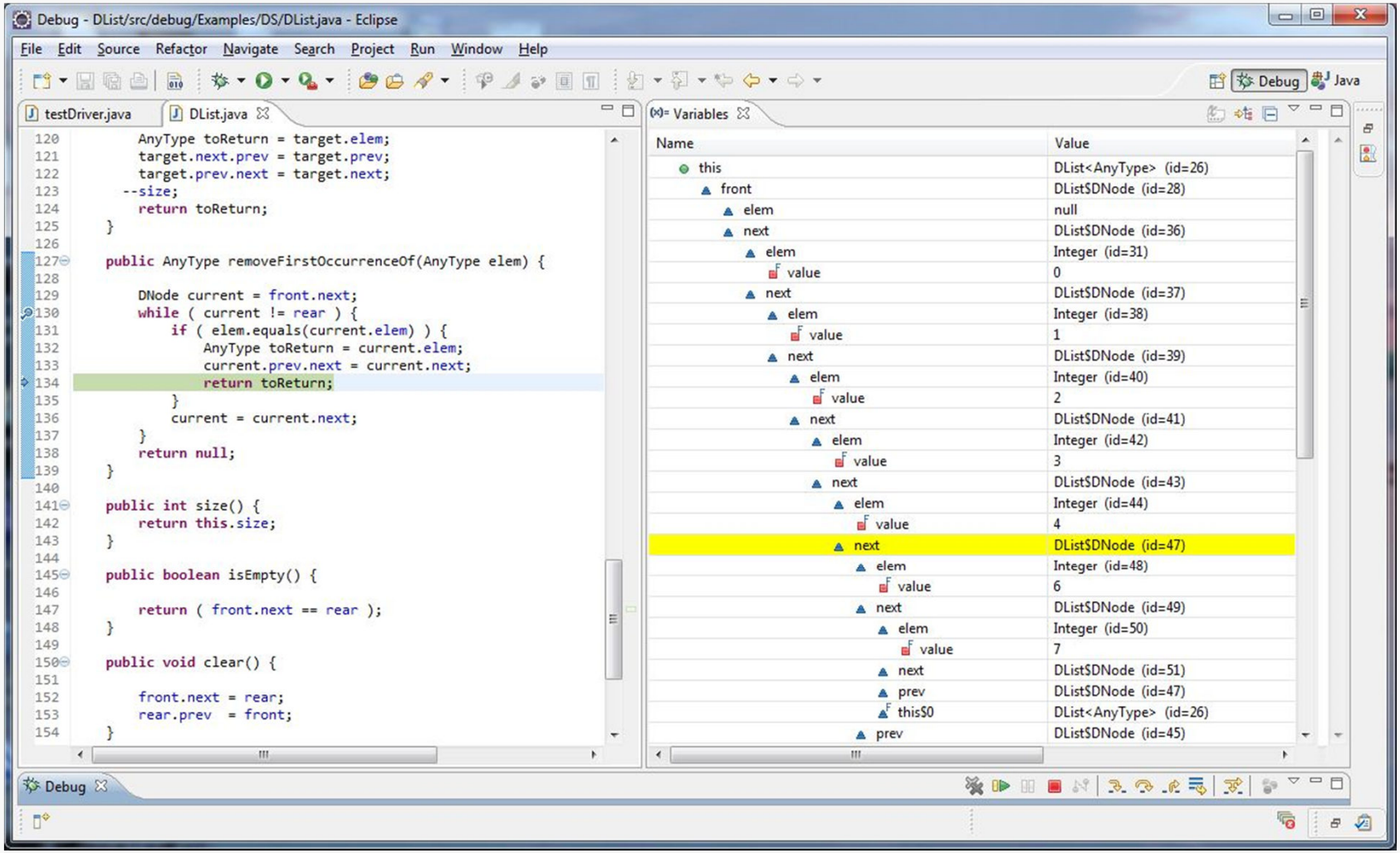
In removeFirstOccurrenceOf()

Continue stepping until **current** reaches the node holding the target value:



At End of removeFirstOccurrenceOf()

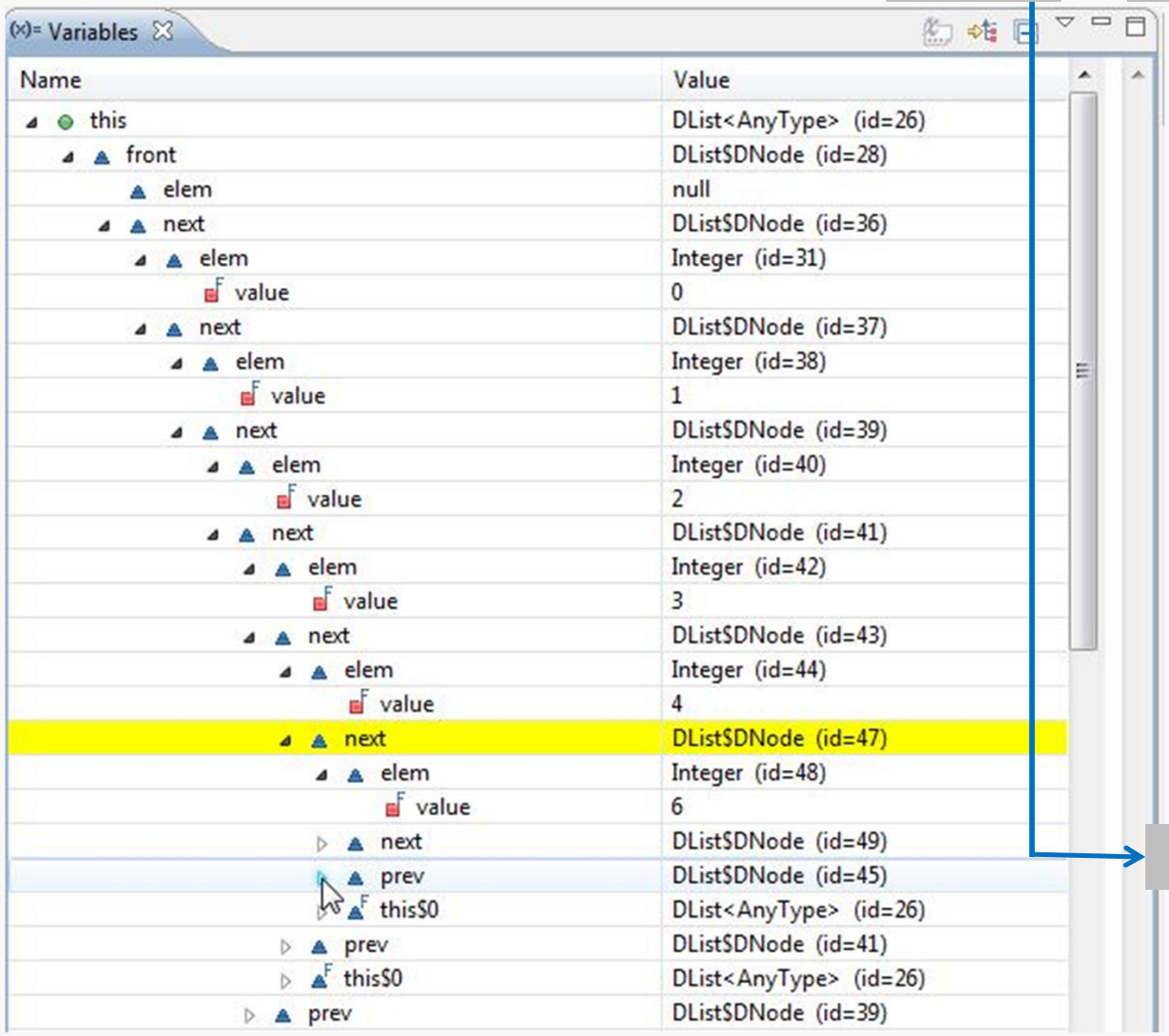
Continue stepping through the **if** statement and examine the list structure right before the **return** is executed:

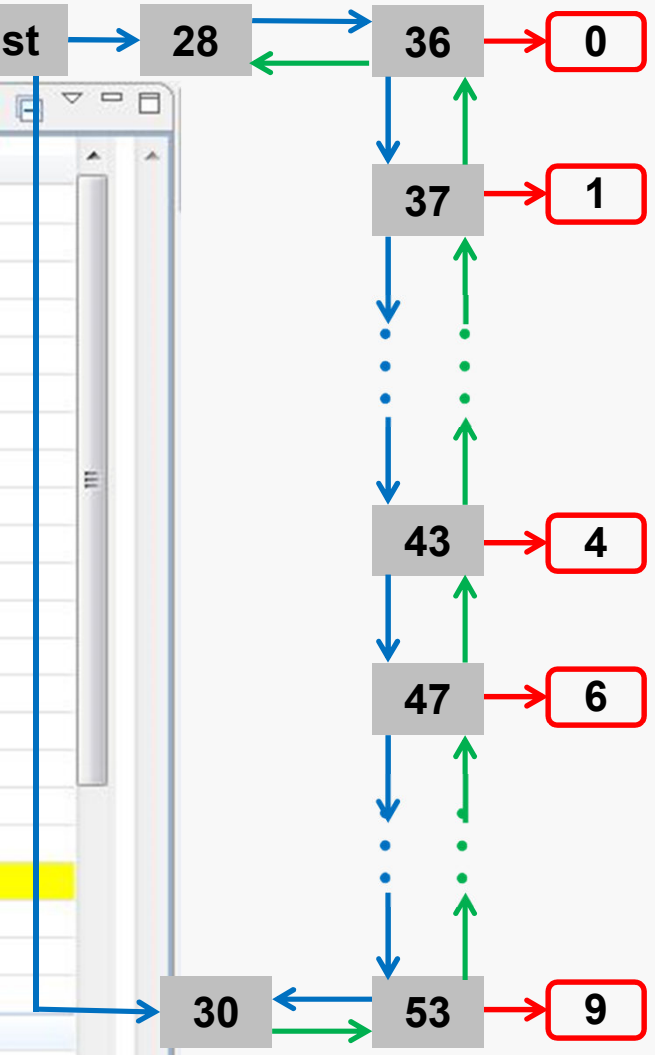



```
120         AnyType toReturn = target.elem;
121         target.next.prev = target.prev;
122         target.prev.next = target.next;
123       --size;
124         return toReturn;
125     }
126
127     public AnyType removeFirstOccurrenceOf(AnyType elem) {
128
129         DNode current = front.next;
130         while ( current != rear ) {
131             if ( elem.equals(current.elem) ) {
132                 AnyType toReturn = current.elem;
133                 current.prev.next = current.next;
134                 return toReturn;
135             }
136             current = current.next;
137         }
138         return null;
139     }
140
141     public int size() {
142         return this.size;
143     }
144
145     public boolean isEmpty() {
146
147         return ( front.next == rear );
148     }
149
150     public void clear() {
151
152         front.next = rear;
153         rear.prev  = front;
154     }
```

List Details

Does the list structure seem to be OK?





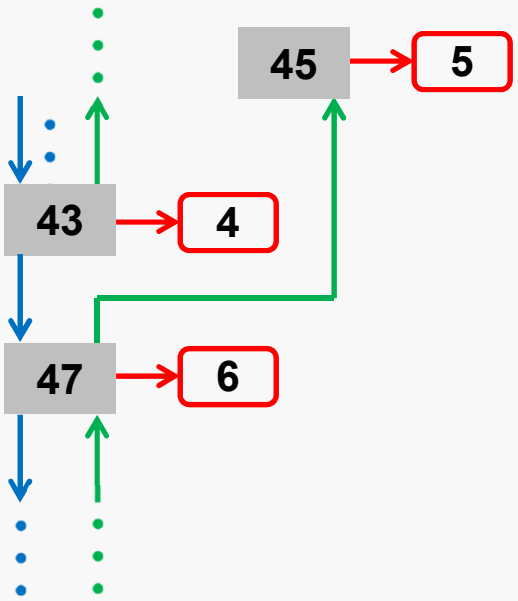
Buggered List Structure (more detail)

A careful examination indicates that something odd has happened:

next	DList$DNode (id=43)
elem	Integer (id=44)
value	4
next	DList$DNode (id=47)
elem	Integer (id=48)
value	6
next	DList$DNode (id=49)
prev	DList$DNode (id=45)
elem	Integer (id=46)
value	5
next	DList$DNode (id=47)

OK

??

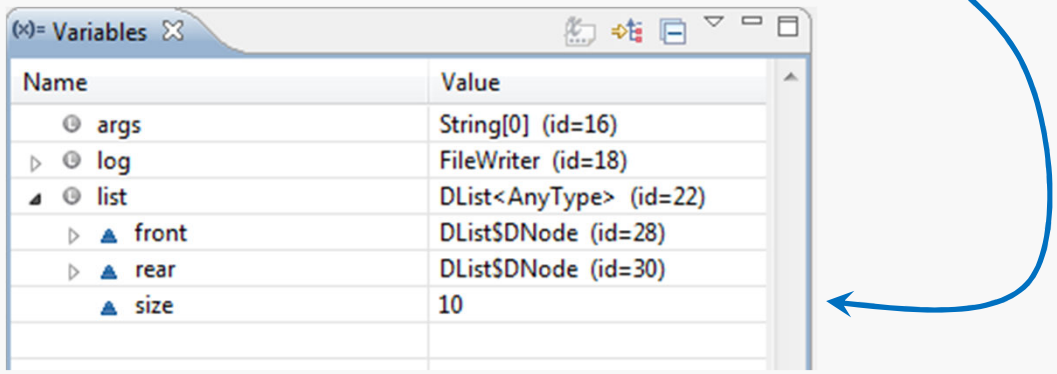


Apparently the removal method did not correctly reset the **prev** pointer in the node after the node that was removed from the list.

We should check that…

Another Error

A careful examination also reveals another bug



A Look at the Code

It should be obvious that two statements are missing from the given code

testDriver.java | DList.java

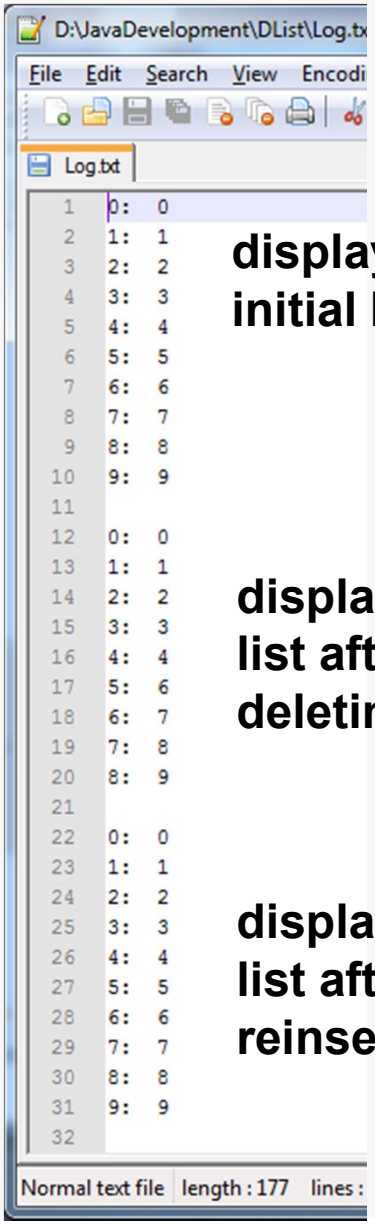
```
125     }
126
127     public AnyType removeFirstOccurrenceOf(AnyType elem) {
128
129         DNode current = front.next;
130         while ( current != rear ) {
131             if ( elem.equals(current.elem) ) {
132                 AnyType toReturn = current.elem;
133                 current.prev.next = current.next;
134                 return toReturn;
135             }
136             current = current.next;
137         }
138         return null;
139     }
140
141     public int size() {
```

testDriver.java | DList.java

```
126
127     public AnyType removeFirstOccurrenceOf(AnyType elem) {
128
129         DNode current = front.next;
130         while ( current != rear ) {
131             if ( elem.equals(current.elem) ) {
132                 AnyType toReturn = current.elem;
133                 current.prev.next = current.next;
134                 current.next.prev = current.prev;
135                 --size;
136                 return toReturn;
137             }
138             current = current.next;
139         }
140         return null;
141     }
142
```

Testing Again

Let's execute the modified program:



display of initial list

display of list after deleting 5

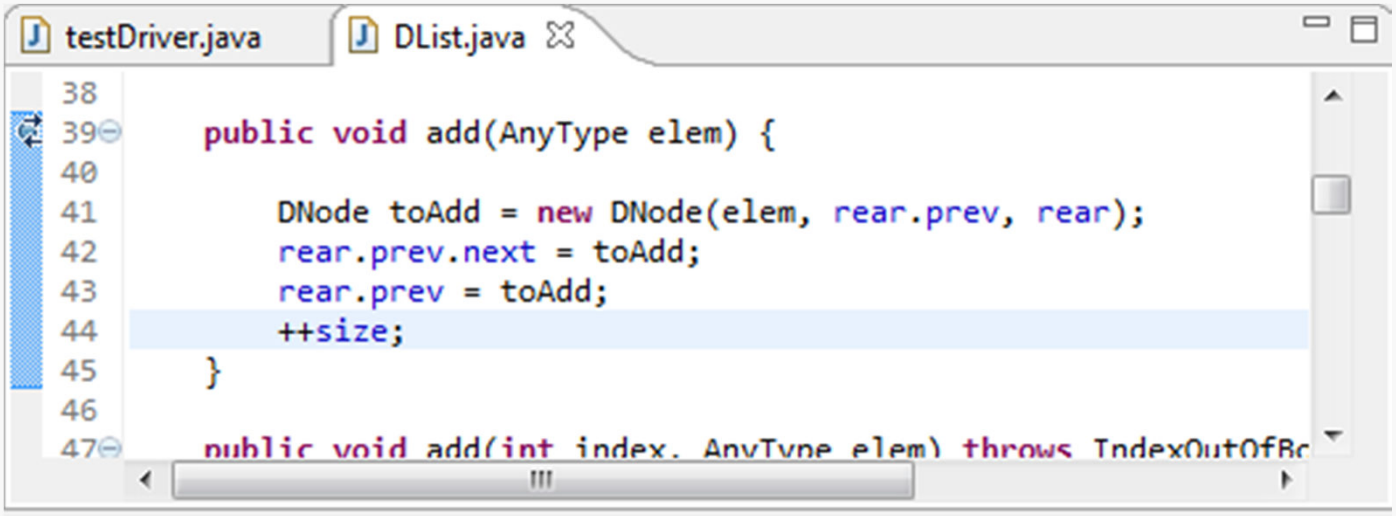
display of list after reinserting 5

Now, the list contents seem to be correct… so, more testing is in order…

Method Breakpoints

method breakpoint halt when execution enters and/or exits a selected method

To set one, just double-click in the editor margin next to the method header:

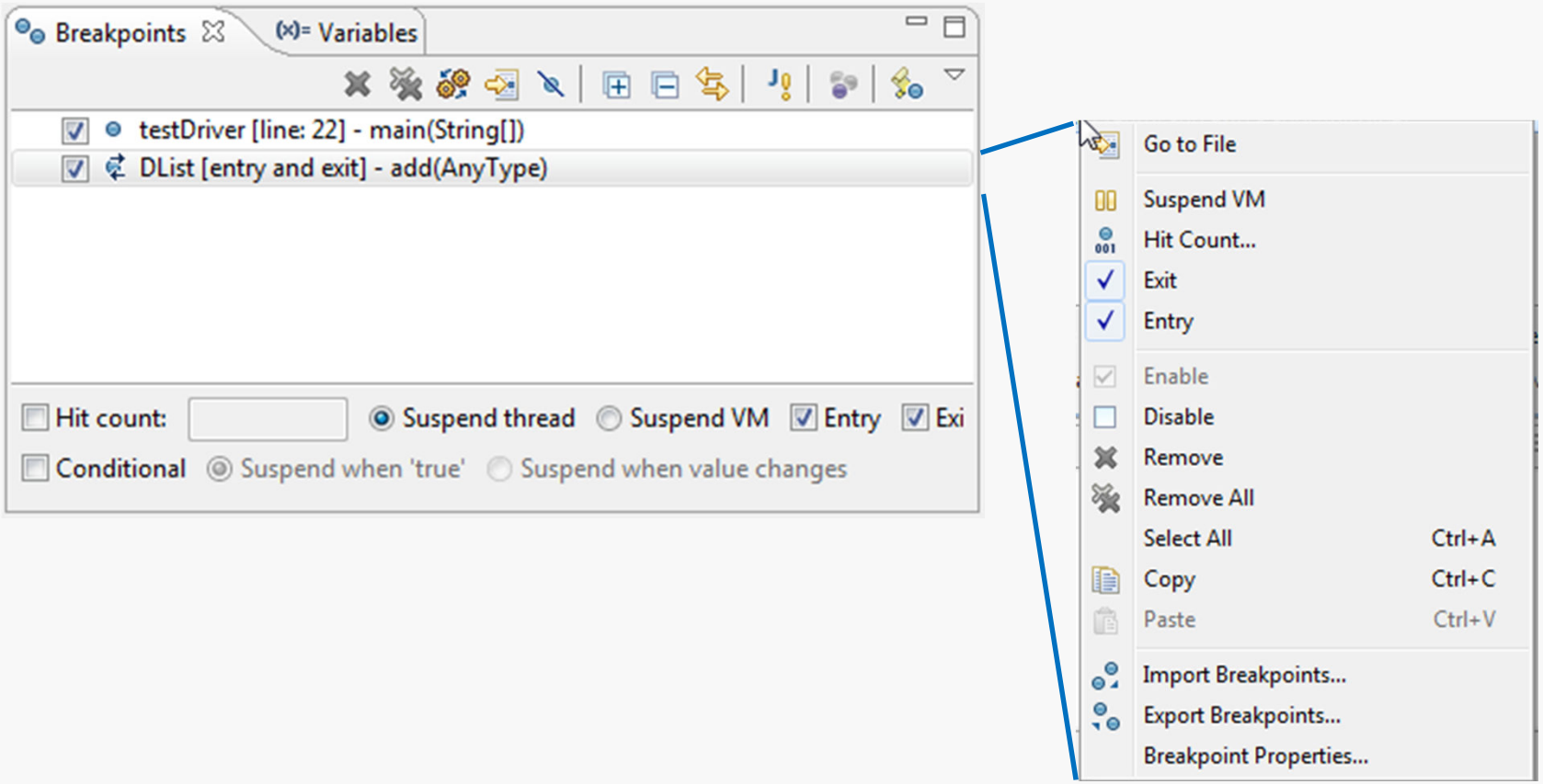


By default, this causes a break when execution enters the method…

Breakpoint View

Go to **Window/Show View** and open the **Breakpoint View**.

You can right-click on a selected breakpoint to alter its properties:



Breakpoint View Toolbar

1 2 3 4 5 6 7 8 9

1. remove selected breakpoints
2. remove all breakpoints
3. show breakpoints
4. go to file for breakpoint
5. skip all breakpoints
6. expand all (details)
7. collapse all (details)
8. link with the Debug View
9. set a Java exception breakpoint

Exception Breakpoints