

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

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

The latest version is available from java.com.

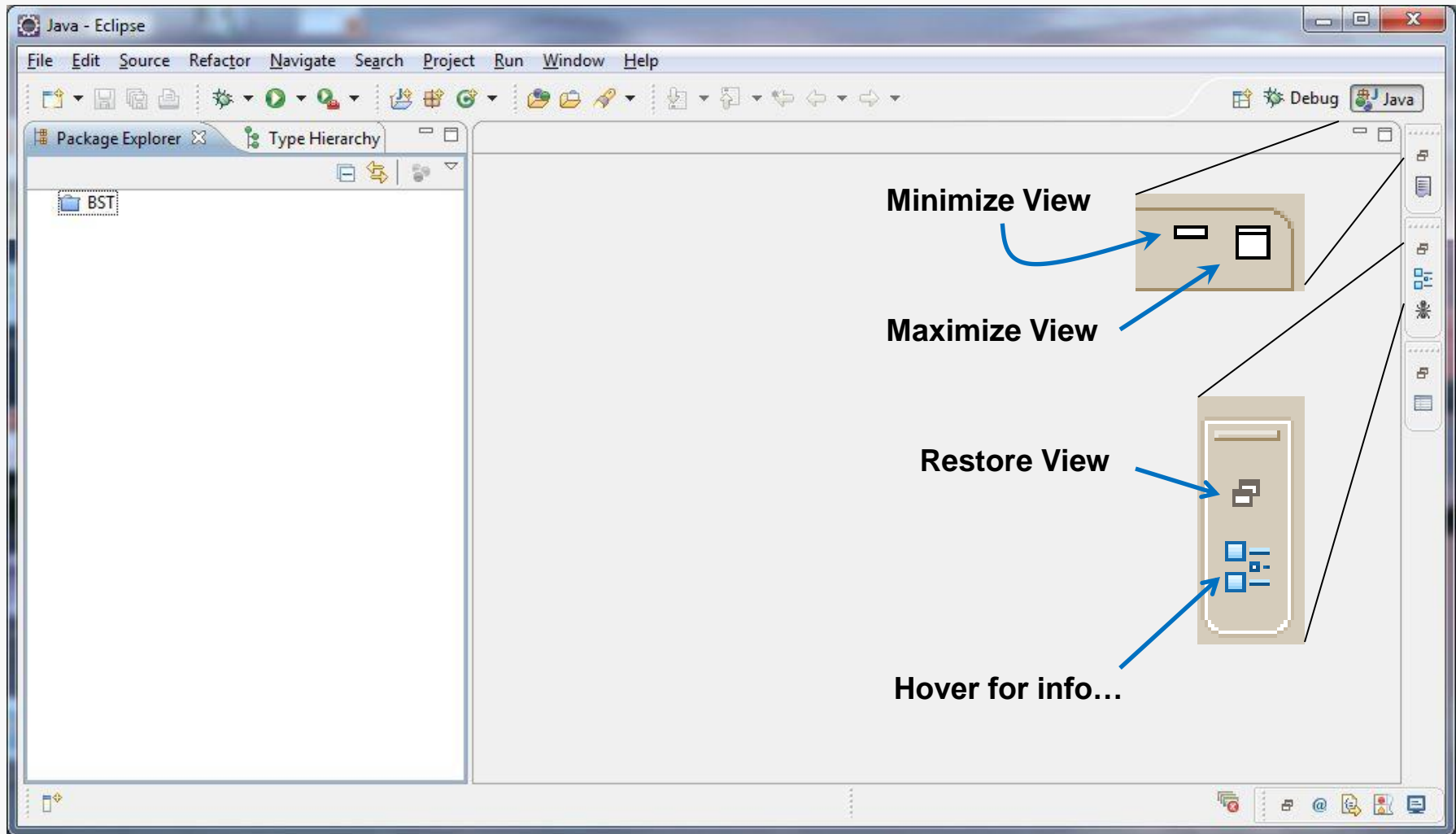
Since Eclipse includes its own Java compiler, it is not strictly necessary to have a version of the Java Development Kit (JDK) installed on your computer.

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.

The initial Eclipse Workbench (my configuration):

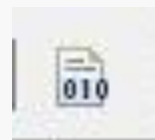




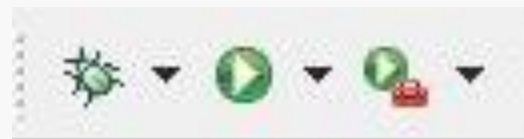
Choose a Perspective



New Project / Save / Save All / Print



Build Project



Start Debugging + configurations

Run Project + configurations

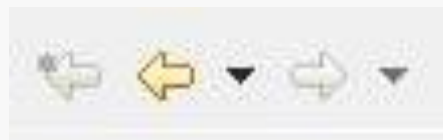
Run Last Tool + configurations



New Java Project / Package / Class



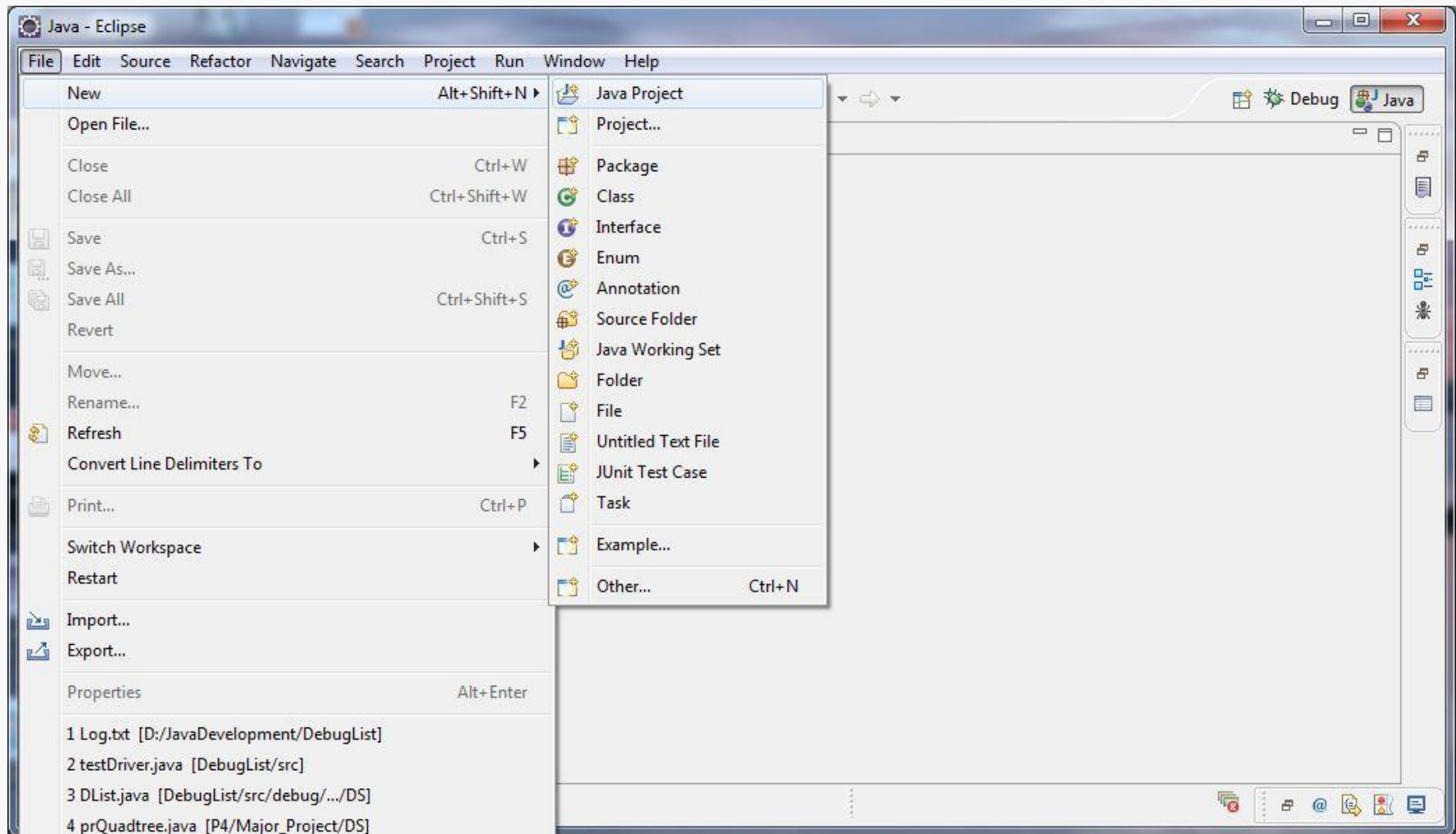
Open Type / Open Task / Search + options



Go to last edit location

Back/Next + more navigation options

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

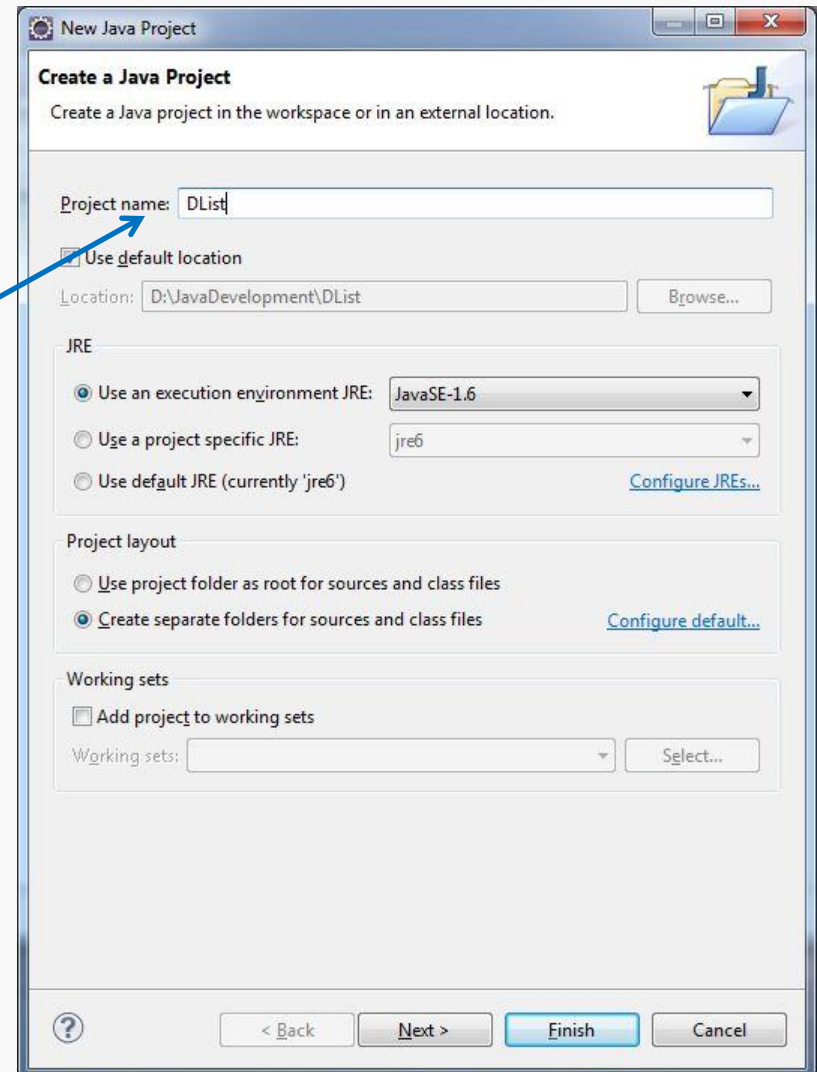


In the resulting dialog box:

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.

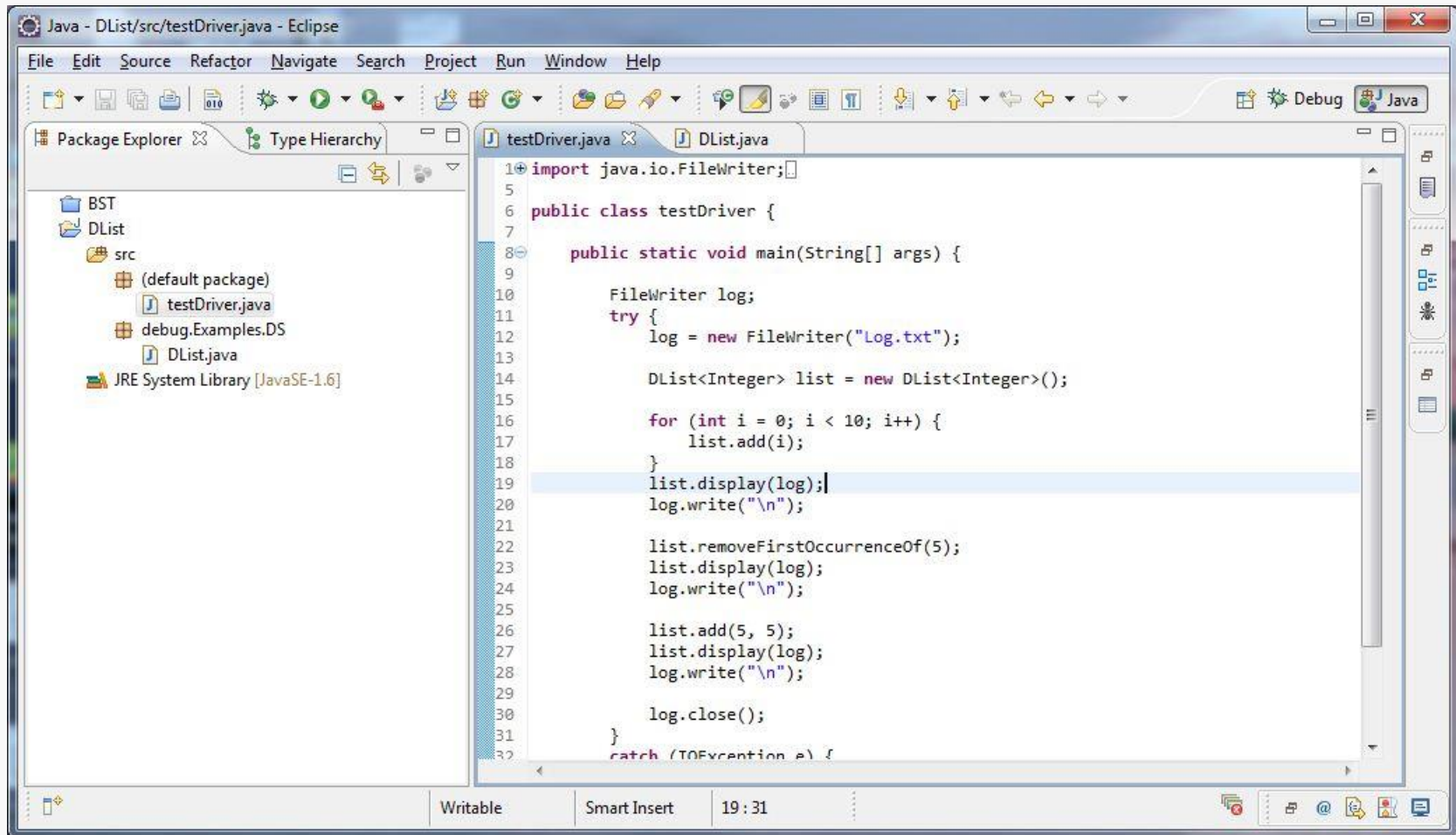


Download the file **DListExample.zip** from the course website Resources page, and place the contents into the **src** directory for the Eclipse project you just created:



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.



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.

```
D:\JavaDevelopment\DLList\Log.txt - Notepad++
File Edit Search View Encoding Language
Log.txt
1 0: 0
2 1: 1
3 2: 2
4 3: 3
5 4: 4
6 5: 5
7 6: 6
8 7: 7
9 8: 8
10 9: 9
11
12 0: 0
13 1: 1
14 2: 2
15 3: 3
16 4: 4
17 5: 6
18 6: 7
19 7: 8
20 8: 9
21
22 0: 0
23 1: 1
24 2: 2
25 3: 3
26 4: 4
27 5: 6
28 6: 7
29 7: 8
30 8: 9
length: 171 lines: 32 Ln: 1 Col: 1 Sel: 0
```

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.

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

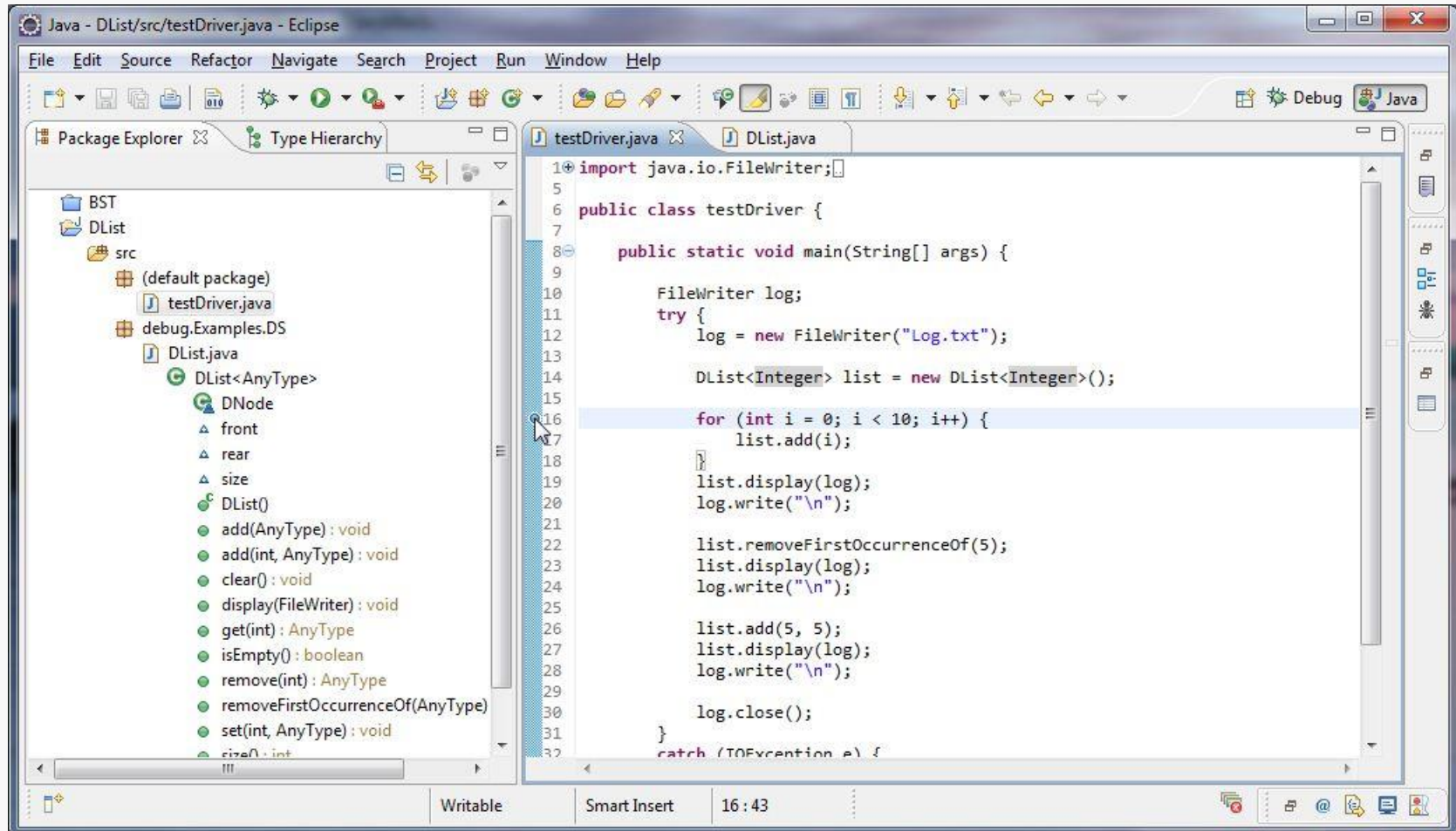
Eclipse supports setting four kinds of breakpoints:

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

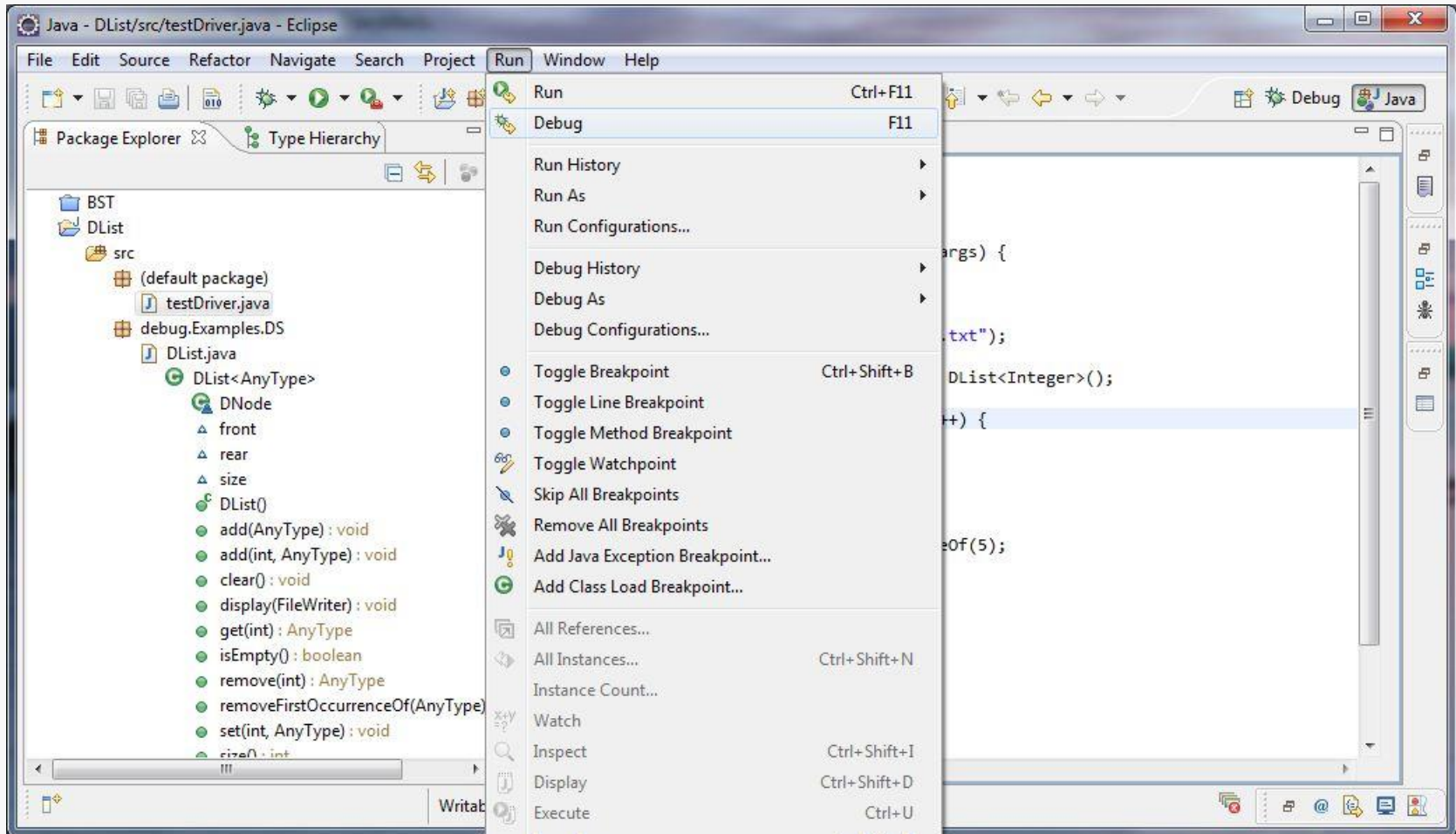
line breakpoint

halt when execution reaches a specific statement

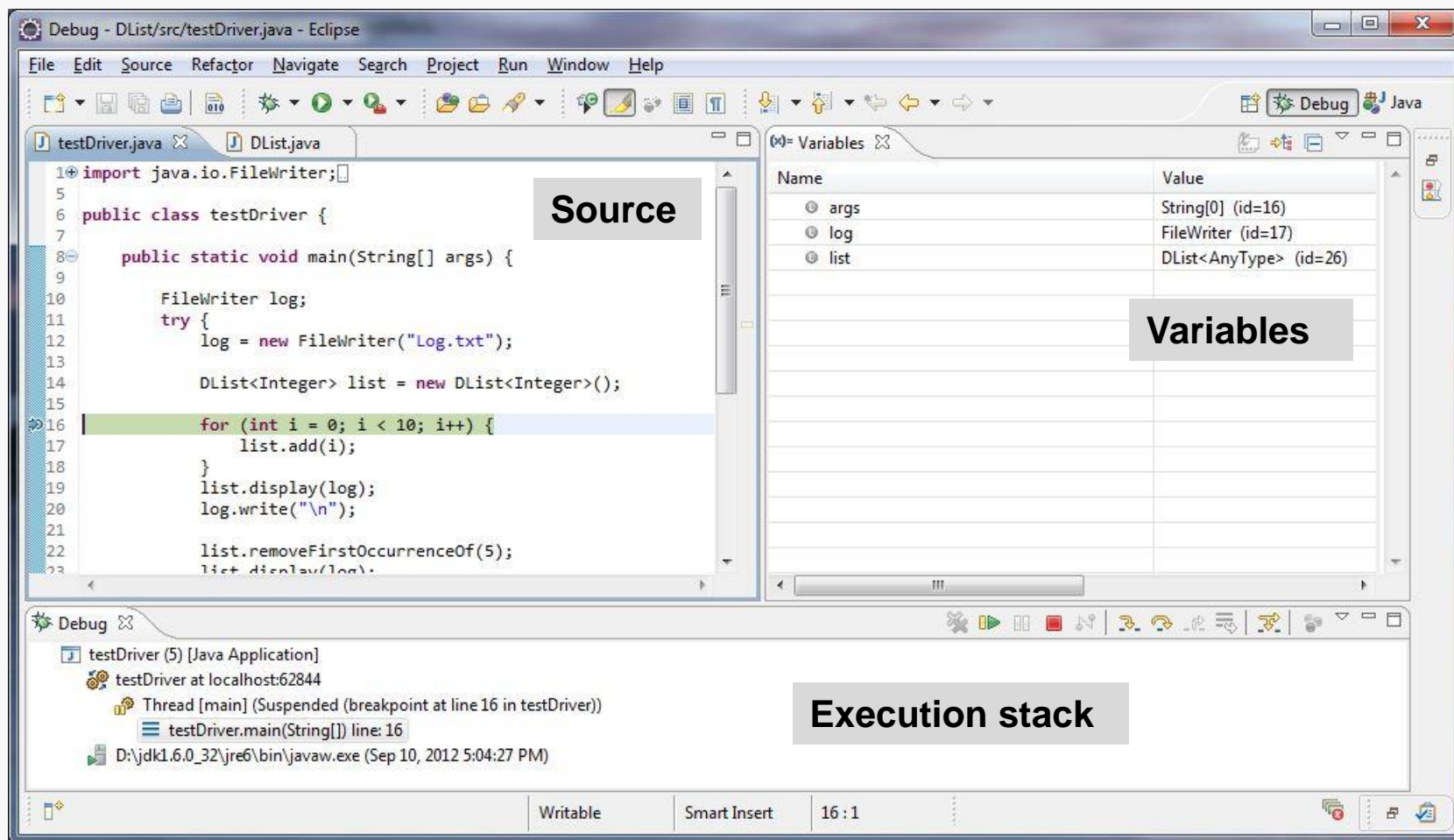
To set one, just double-click in the editor margin next to the selected line of code:



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

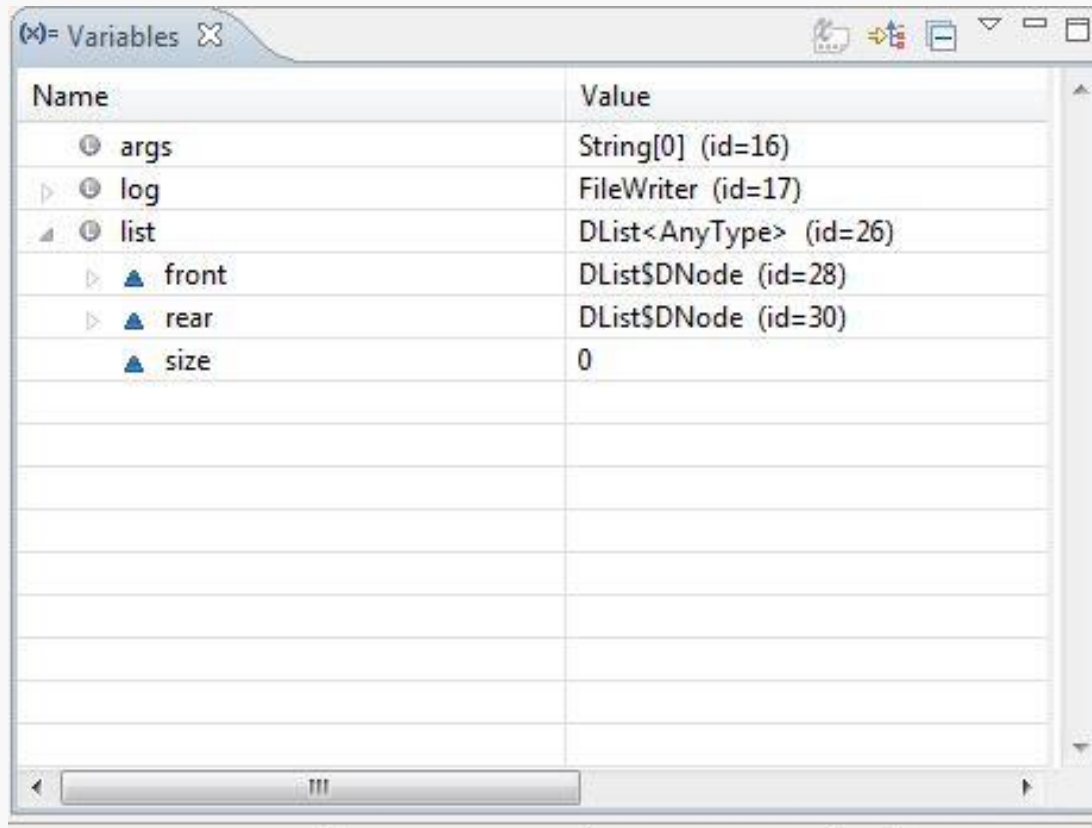


This opens the Debug Perspective:



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

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

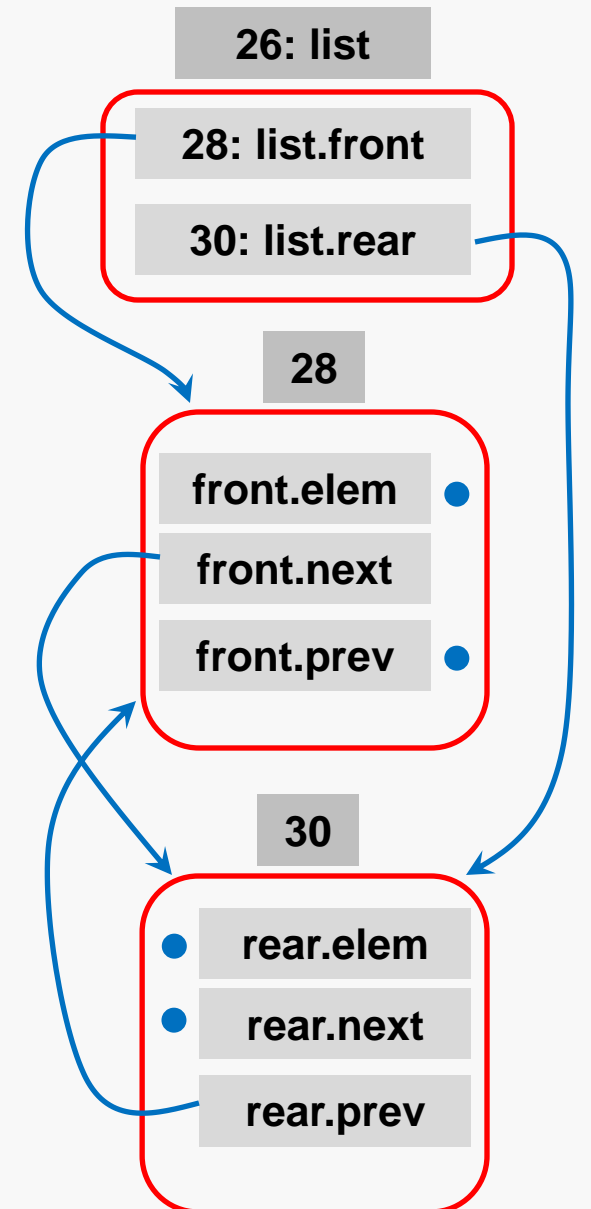


Objects are assigned unique IDs as they are created; these allow us to infer the physical structure...

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

Name	Value
args	String[0] (id=16)
log	FileWriter (id=17)
list	DList<AnyType> (id=26)
front	DList\$SDNode (id=28)
elem	null
next	DList\$SDNode (id=30)
prev	null
this\$0	DList<AnyType> (id=26)
rear	DList\$SDNode (id=30)
elem	null
next	null
prev	DList\$SDNode (id=28)
this\$0	DList<AnyType> (id=26)
size	0

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

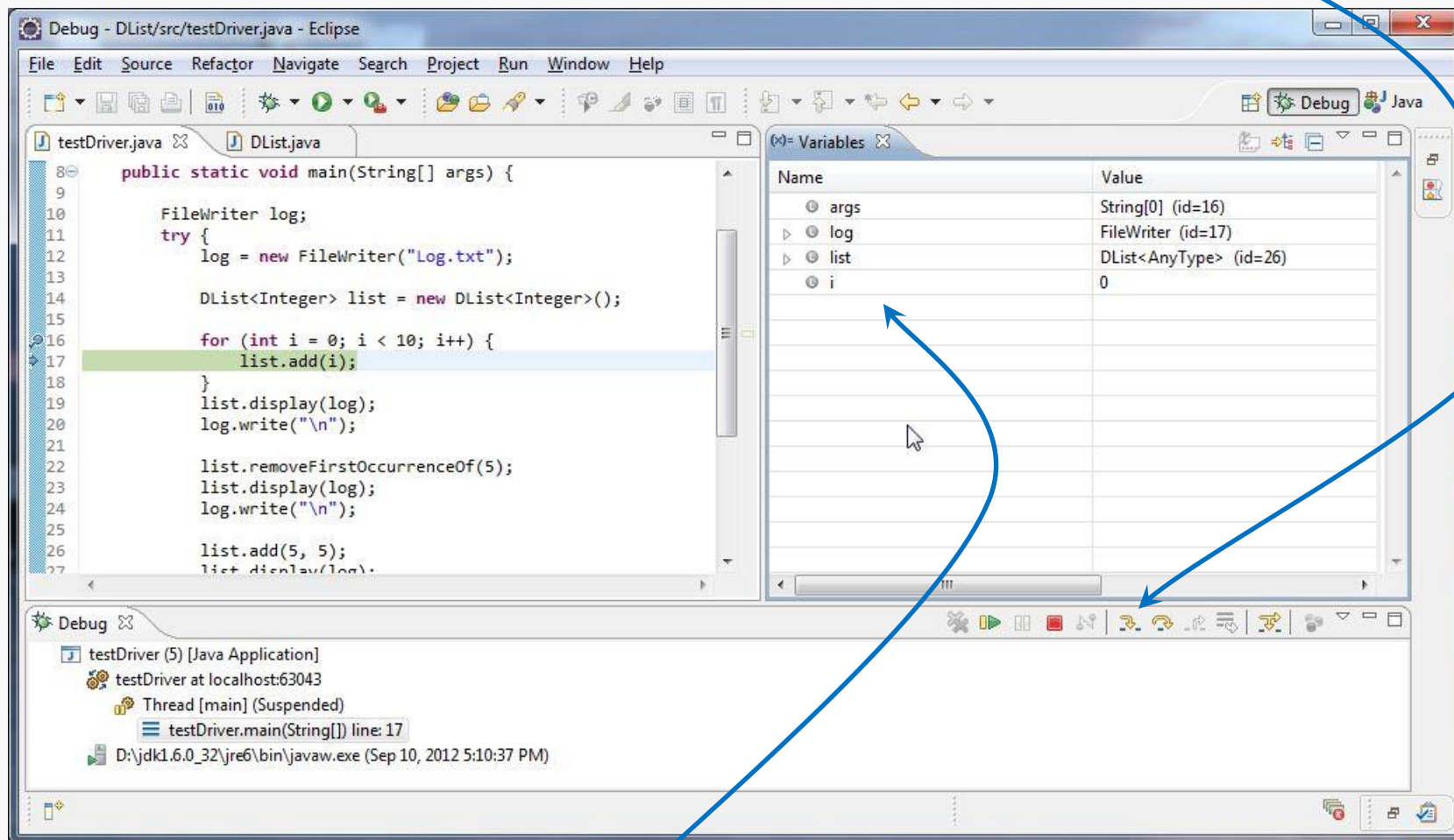




1 2 3 4 5 6 7 8 9 10

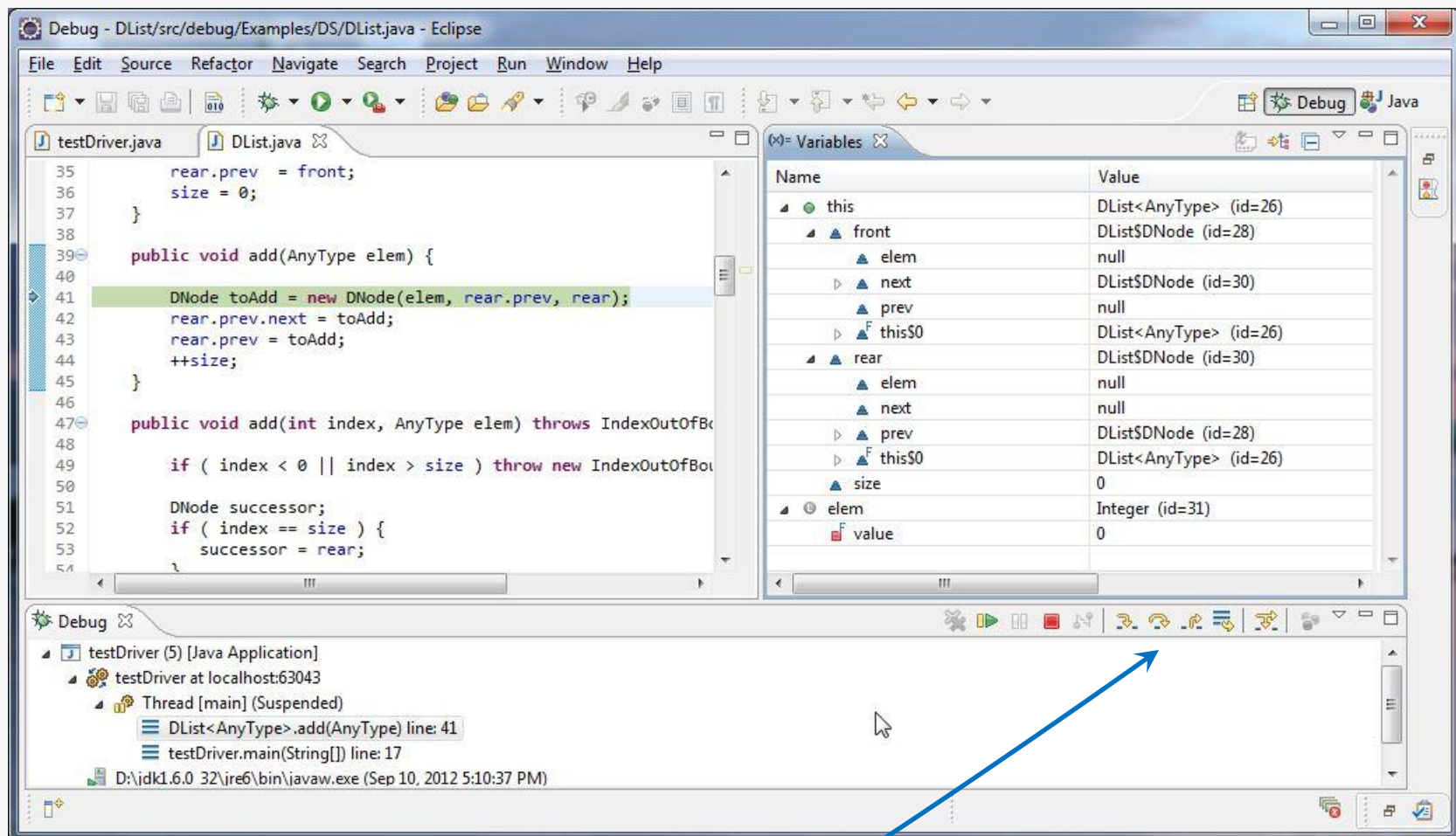
1. **Resume** – Continues execution until breakpoint or thread ends
2. **Suspend** – Interrupts a running thread
3. **Terminate** – Ends the execution of the selected thread
4. **Disconnect** – Disconnect from a remote debugging session
5. **Remove terminated launches** – Closes all terminated debug 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

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.

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...

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.

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

The screenshot shows the Eclipse IDE in a debug state. The main editor displays the source code for `DList.java`. The current line of execution is line 42: `rear.prev.next = toAdd;`. The Variables window on the right shows the state of the program:

Name	Value
this	DList<AnyType> (id=26)
front	DList\$DNode (id=28)
elem	null
next	DList\$DNode (id=30)
prev	null
this\$0	DList<AnyType> (id=26)
rear	DList\$DNode (id=30)
elem	null
next	null
prev	DList\$DNode (id=28)
this\$0	DList<AnyType> (id=26)
size	0
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)

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

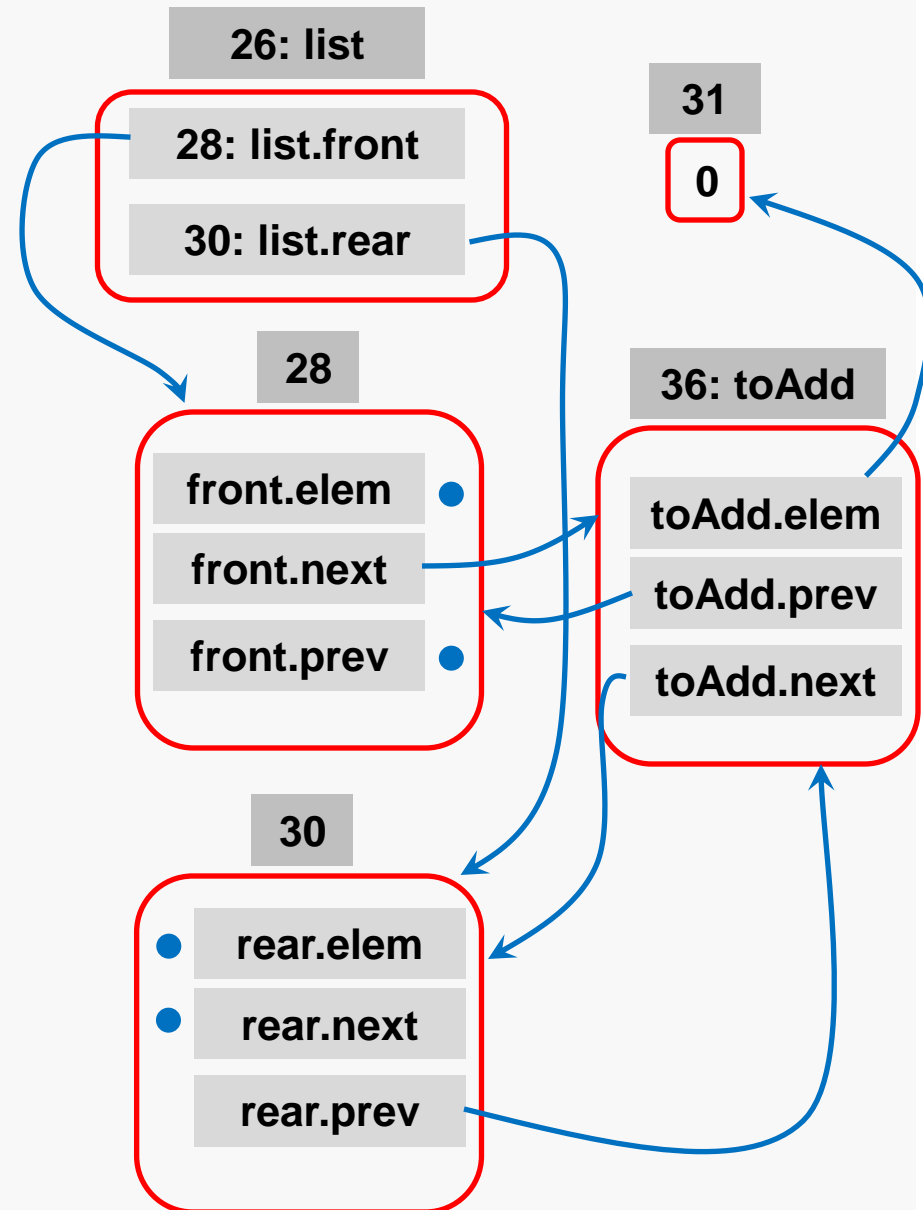
The screenshot shows the Eclipse IDE in debug mode. The main editor displays the source code of `DList.java`. The `add` method is currently being executed, and the `size` variable is highlighted in yellow. The Variables view on the right shows the state of the program, with `size` set to 1.

```
35     rear.prev = front;
36     size = 0;
37 }
38
39 public void add(AnyType elem) {
40
41     DNode toAdd = new DNode(elem, rear.prev, rear);
42     rear.prev.next = toAdd;
43     rear.prev = toAdd;
44     ++size;
45 }
46
47 public void add(int index, AnyType elem) throws IndexOutOfBoundsException {
48
49     if ( index < 0 || index > size ) throw new IndexOutOfBoundsException();
50
51     DNode successor;
52     if ( index == size ) {
53         successor = rear;
54     }
55     else {
56         successor = front.next;
57         int i = 0;
58         while ( i < index ) {
59             successor = successor.next;
60             ++i;
61         }
62     }
63     rear.prev.next = successor;
64     rear.prev = successor;
65     ++size;
66 }
```

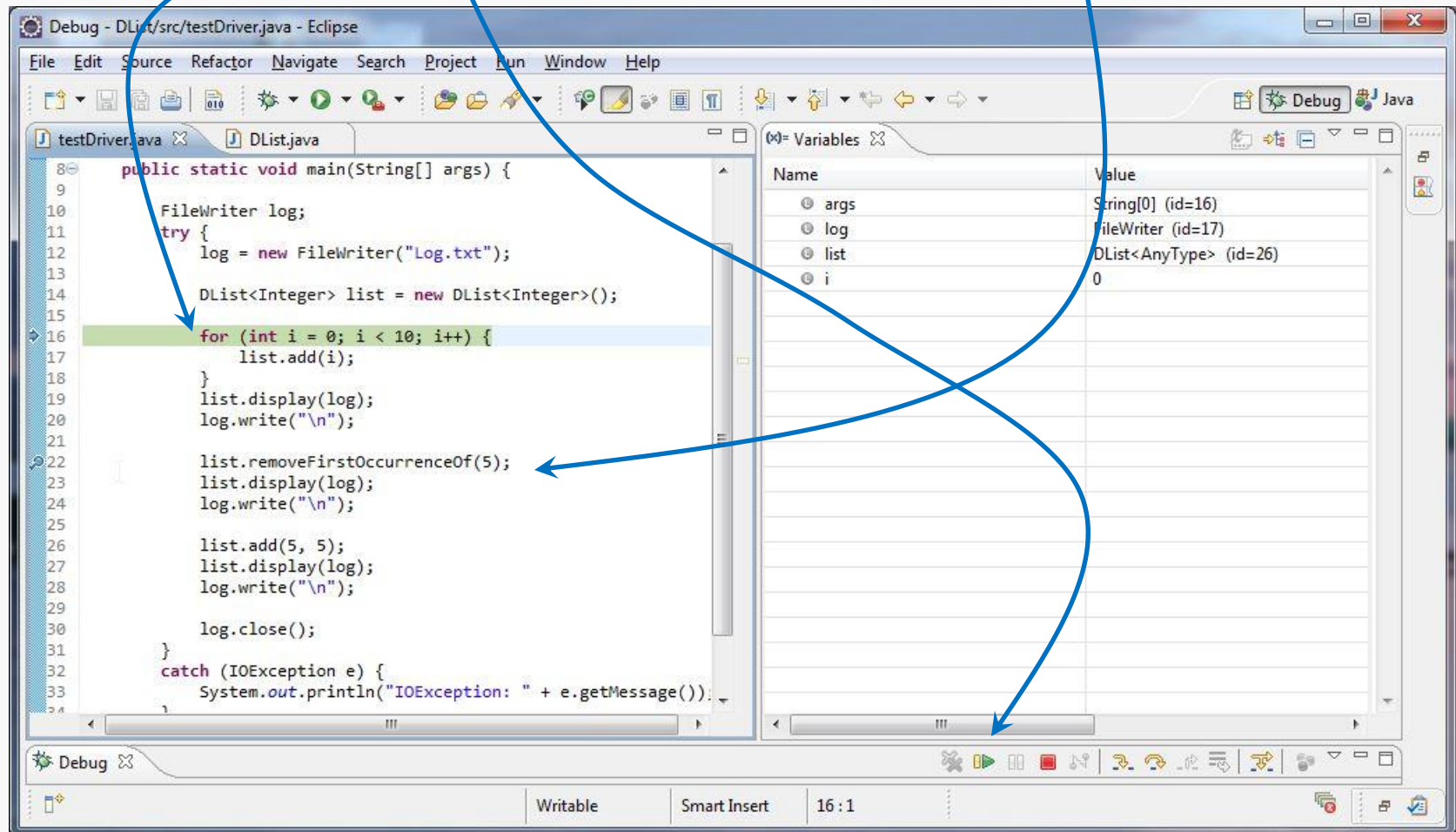
Name	Value
• this	DList<AnyType> (id=26)
▲ front	DListSDNode (id=28)
▲ elem	null
▲ next	DListSDNode (id=36)
▲ prev	null
▲ this\$0	DList<AnyType> (id=26)
▲ rear	DListSDNode (id=30)
▲ elem	null
▲ next	null
▲ prev	DListSDNode (id=36)
▲ this\$0	DList<AnyType> (id=26)
▲ size	1
• elem	Integer (id=31)
▲ value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

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

Well, that looks OK.



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:



Execution proceeds to the new breakpoint:

The screenshot shows the Eclipse IDE in a debug state. The main editor displays the source code of `testDriver.java`. A breakpoint is set at line 22, which is highlighted in green. The code includes a `main` method that creates a `DList<Integer>` object, adds elements from 0 to 9, and then reaches the breakpoint at `list.removeFirstOccurrenceOf(5);`. The `Variables` window on the right shows the current state of the program's variables. The `list` variable is expanded to show its internal structure, including `front`, `rear`, and `size` attributes. The `size` attribute is highlighted in yellow and has a value of 10. A blue arrow points from the text above to the breakpoint in the code.

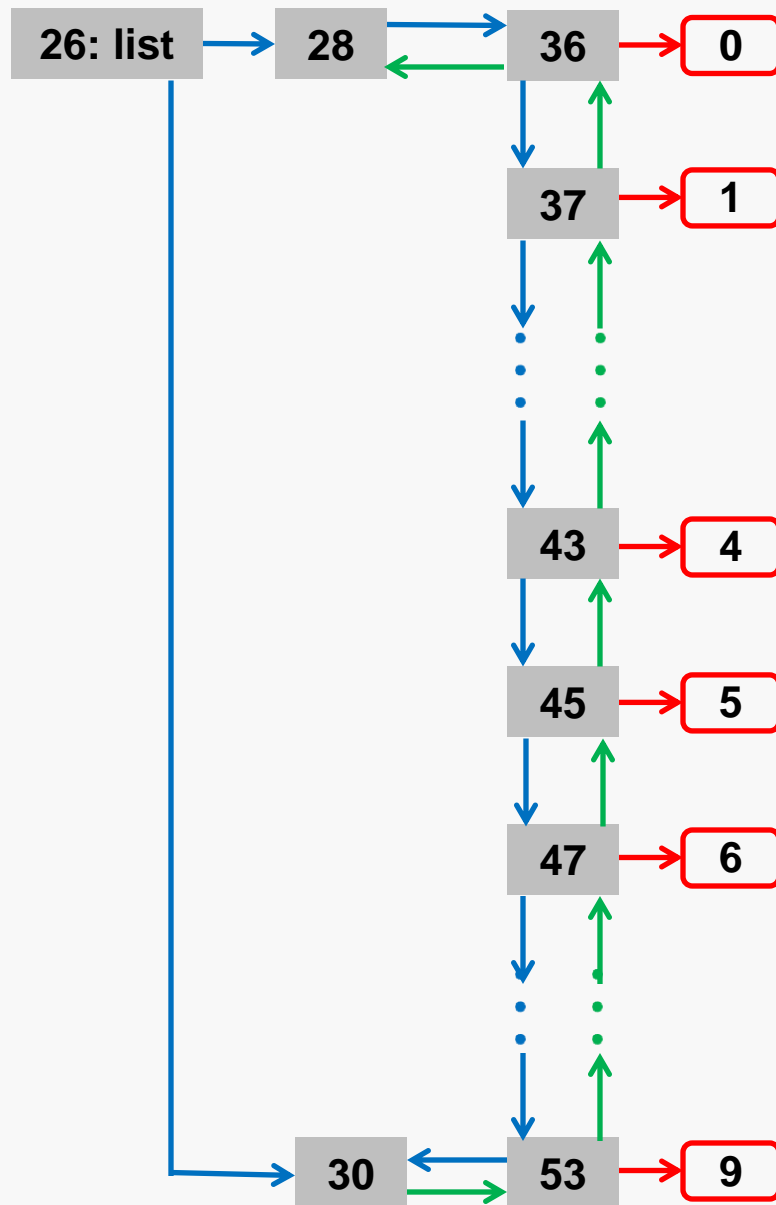
```
public static void main(String[] args) {
    9
    FileWriter log;
    11    try {
    12        log = new FileWriter("Log.txt");
    13
    14        DList<Integer> list = new DList<Integer>();
    15
    16        for (int i = 0; i < 10; i++) {
    17            list.add(i);
    18        }
    19        list.display(log);
    20        log.write("\n");
    21
    22    list.removeFirstOccurrenceOf(5);
    23    list.display(log);
    24    log.write("\n");
    25
    26    list.add(5, 5);
    27    list.display(log);
    28    log.write("\n");
    29
    30    log.close();
    31    }
    32    catch (IOException e) {
    33        System.out.println("IOException: " + e.getMessage());
    34    }
```

Name	Value
args	String[0] (id=16)
log	FileWriter (id=17)
list	DList<AnyType> (id=26)
front	DListSDNode (id=28)
rear	DListSDNode (id=30)
size	10

Complete List Structure

Debugging 27

Name	Value
args	String[0] (id=16)
log	FileWriter (id=17)
list	DList<AnyType> (id=26)
front	DListSDNode (id=28)
elem	null
next	DListSDNode (id=36)
elem	Integer (id=31)
value	0
next	DListSDNode (id=37)
elem	Integer (id=38)
value	1
next	DListSDNode (id=39)
elem	Integer (id=40)
value	2
next	DListSDNode (id=41)
elem	Integer (id=42)
value	3
next	DListSDNode (id=43)
elem	Integer (id=44)
value	4
next	DListSDNode (id=45)
prev	DListSDNode (id=41)
this\$0	DList<AnyType> (id=26)
prev	DListSDNode (id=39)
this\$0	DList<AnyType> (id=26)
prev	DListSDNode (id=37)
this\$0	DList<AnyType> (id=26)
prev	DListSDNode (id=36)



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

The screenshot shows the Eclipse IDE in a debug state. The main editor displays the `removeFirstOccurrenceOf` method in `DList.java`. The code is as follows:

```
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 }
```

The Variables view on the right shows the following state:

Name	Value
this	DList<AnyType> (id=26)
front	DListSDNode (id=28)
rear	DListSDNode (id=30)
size	10
elem	Integer (id=46)
value	5
current	DListSDNode (id=36)
elem	Integer (id=31)
value	0
next	DListSDNode (id=37)
prev	DListSDNode (id=28)
this\$0	DList<AnyType> (id=26)

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

The screenshot shows the Eclipse IDE with a Java project named 'DList'. The main editor displays the source code for 'DList.java'. The code is as follows:

```
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 }
```

The 'Variables' window on the right shows the state of the program. The 'current' variable is highlighted, indicating it is the current value of the 'current' pointer. The 'current' variable is of type 'DListSDNode' and has an id of 45. Other variables shown include 'this' (DList<AnyType> (id=26)), 'front' (DListSDNode (id=28)), 'rear' (DListSDNode (id=30)), 'size' (10), 'elem' (Integer (id=46)), 'value' (5), 'next' (DListSDNode (id=47)), 'prev' (DListSDNode (id=43)), and 'this\$0' (DList<AnyType> (id=26)).

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

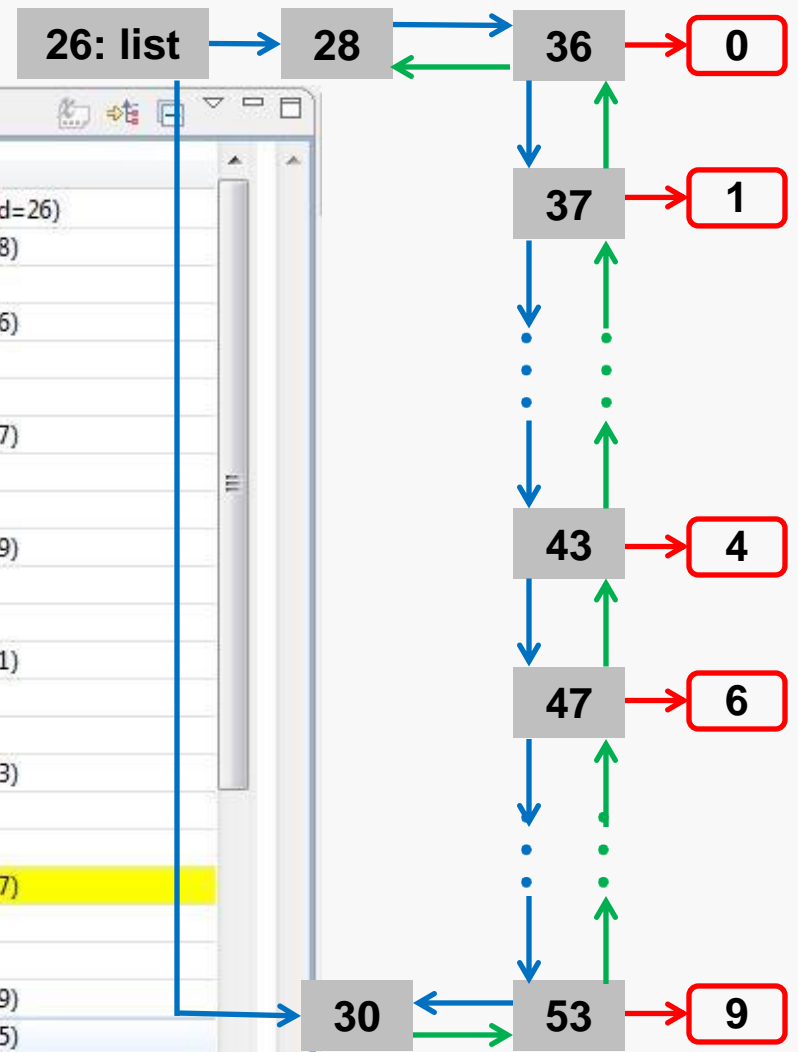
The screenshot shows the Eclipse IDE with the following components:

- Code Editor:** Shows the `removeFirstOccurrenceOf` method. Line 134 is highlighted, showing `return toReturn;`. The method is currently executing at this line.
- Variables Window:** Displays the state of the `DList` object. The `next` pointer of the current node is highlighted in yellow, pointing to the next node in the list.

Name	Value
this	DList<AnyType> (id=26)
front	DList\$DNode (id=28)
elem	null
next	DList\$DNode (id=36)
elem	Integer (id=31)
value	0
next	DList\$DNode (id=37)
elem	Integer (id=38)
value	1
next	DList\$DNode (id=39)
elem	Integer (id=40)
value	2
next	DList\$DNode (id=41)
elem	Integer (id=42)
value	3
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)
elem	Integer (id=50)
value	7
next	DList\$DNode (id=51)
prev	DList\$DNode (id=47)
this\$0	DList<AnyType> (id=26)
prev	DList\$DNode (id=45)

Does the list structure seem to be OK?

Name	Value
this	DList<AnyType> (id=26)
front	DListSDNode (id=28)
elem	null
next	DListSDNode (id=36)
elem	Integer (id=31)
value	0
next	DListSDNode (id=37)
elem	Integer (id=38)
value	1
next	DListSDNode (id=39)
elem	Integer (id=40)
value	2
next	DListSDNode (id=41)
elem	Integer (id=42)
value	3
next	DListSDNode (id=43)
elem	Integer (id=44)
value	4
next	DListSDNode (id=47)
elem	Integer (id=48)
value	6
next	DListSDNode (id=49)
prev	DListSDNode (id=45)
this\$0	DList<AnyType> (id=26)
prev	DListSDNode (id=41)
this\$0	DList<AnyType> (id=26)
prev	DListSDNode (id=39)

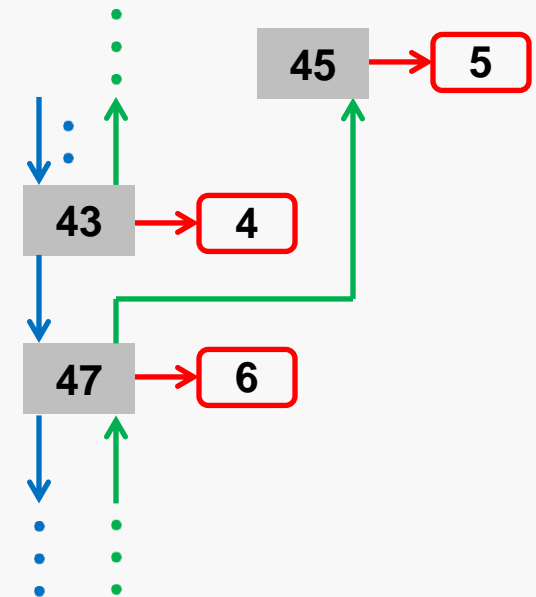


A careful examination indicates that something odd has happened:

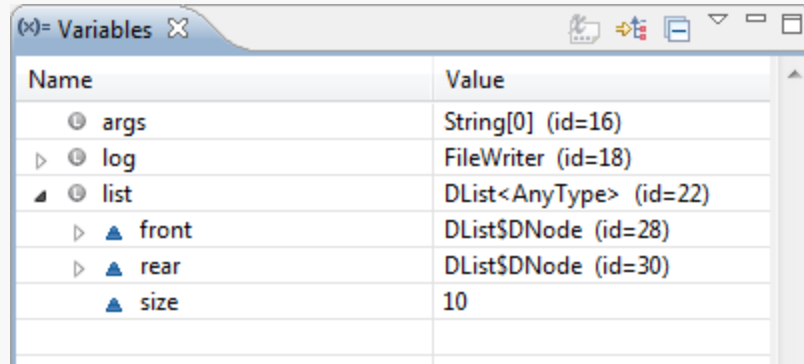
next	DListSDNode (id=43)	}	OK
elem	Integer (id=44)		
value	4		
next	DListSDNode (id=47)	}	??
elem	Integer (id=48)		
value	6		
next	DListSDNode (id=49)		
prev	DListSDNode (id=45)		
elem	Integer (id=46)		
value	5		
next	DListSDNode (id=47)		

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...



A careful examination also reveals another bug



Name	Value
args	String[0] (id=16)
log	FileWriter (id=18)
list	DList<AnyType> (id=22)
front	DList\$DNode (id=28)
rear	DList\$DNode (id=30)
size	10

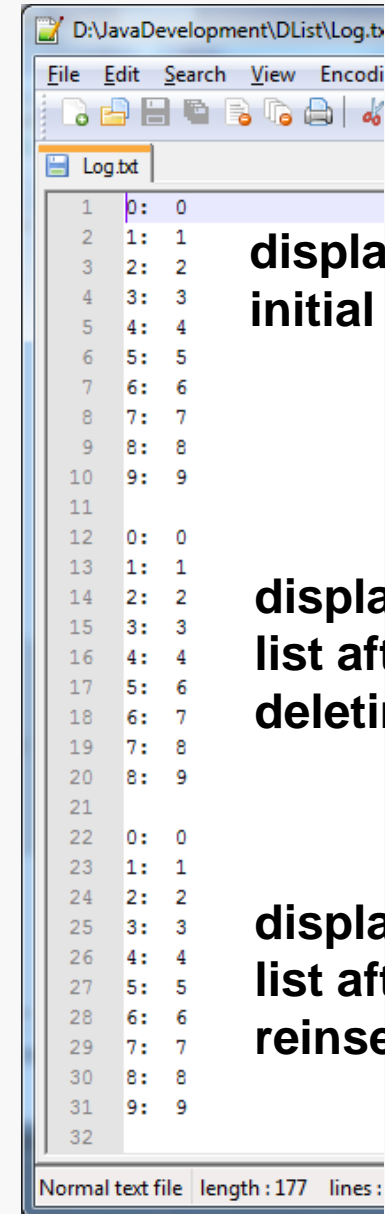
It should be obvious that two statements are missing from the given code

```
testDriver.java  DList.java X
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 X
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 }
```

Let's execute the modified program:

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



The screenshot shows a Notepad window titled "D:\JavaDevelopment\DLList\Log.txt" with a menu bar (File, Edit, Search, View, Encodi) and a toolbar. The text content is as follows:

```
1 0: 0
2 1: 1
3 2: 2
4 3: 3
5 4: 4
6 5: 5
7 6: 6
8 7: 7
9 8: 8
10 9: 9
11
12 0: 0
13 1: 1
14 2: 2
15 3: 3
16 4: 4
17 5: 6
18 6: 7
19 7: 8
20 8: 9
21
22 0: 0
23 1: 1
24 2: 2
25 3: 3
26 4: 4
27 5: 5
28 6: 6
29 7: 7
30 8: 8
31 9: 9
32
```

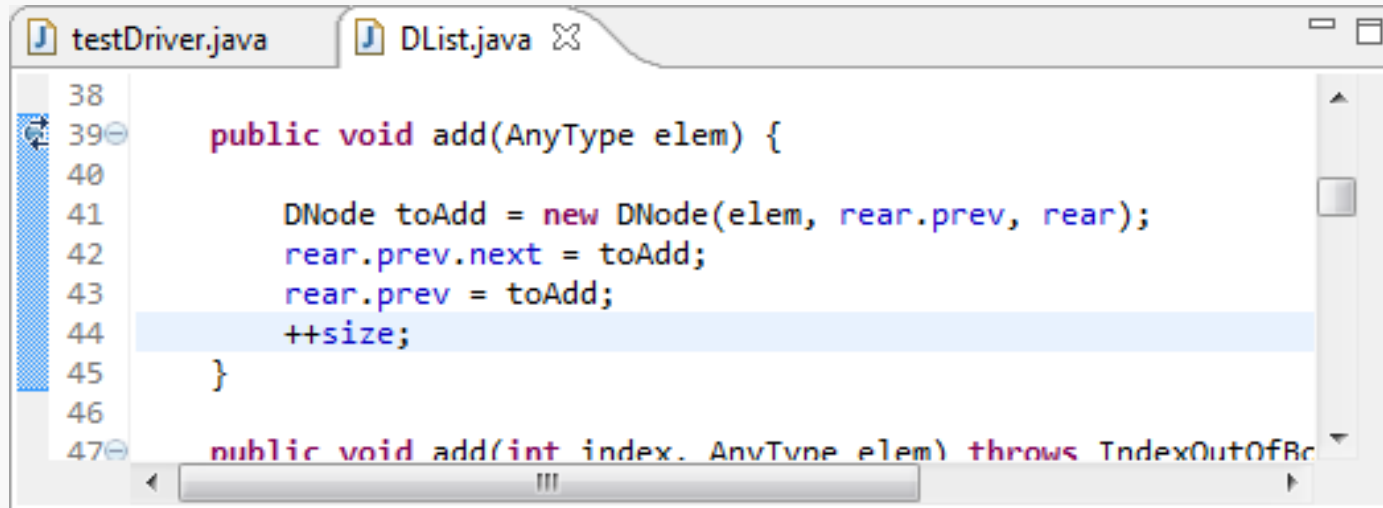
Annotations on the right side of the screenshot:

- display of initial list** (next to lines 1-10)
- display of list after deleting 5** (next to lines 12-20)
- display of list after reinserting 5** (next to lines 22-31)

At the bottom of the window, it says "Normal text file length : 177 lines :

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:



The screenshot shows an IDE window with two tabs: 'testDriver.java' and 'DList.java'. The 'DList.java' tab is active, showing the following code:

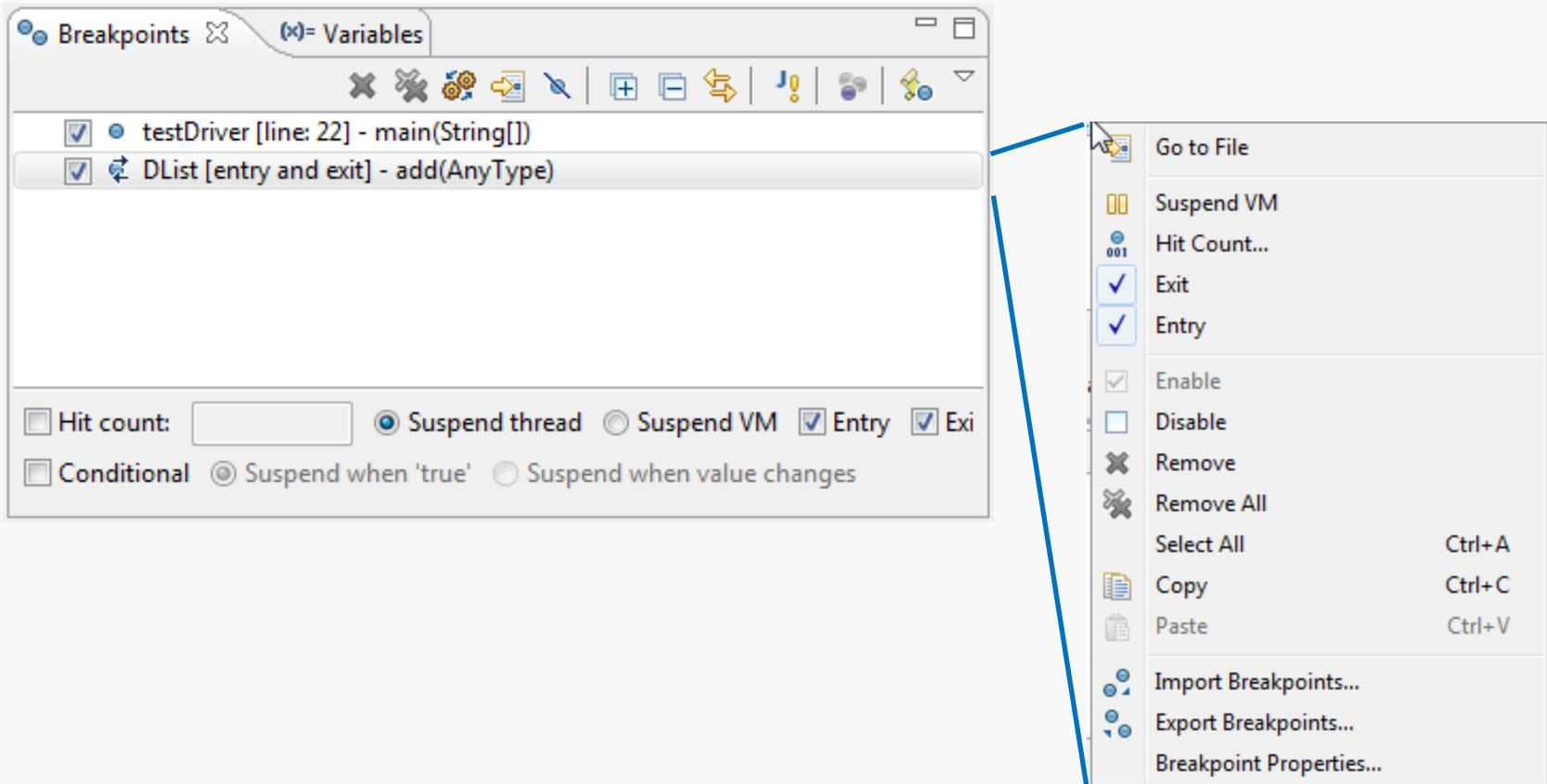
```
38  
39 public void add(AnyType elem) {  
40  
41     DNode toAdd = new DNode(elem, rear.prev, rear);  
42     rear.prev.next = toAdd;  
43     rear.prev = toAdd;  
44     ++size;  
45 }  
46  
47 public void add(int index, AnyType elem) throws IndexOutOfRangeException
```

A blue dotted breakpoint icon is placed on the left margin next to the method header on line 39. The code lines 41 through 44 are highlighted in light blue.

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

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

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





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

