

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](http://java.com).

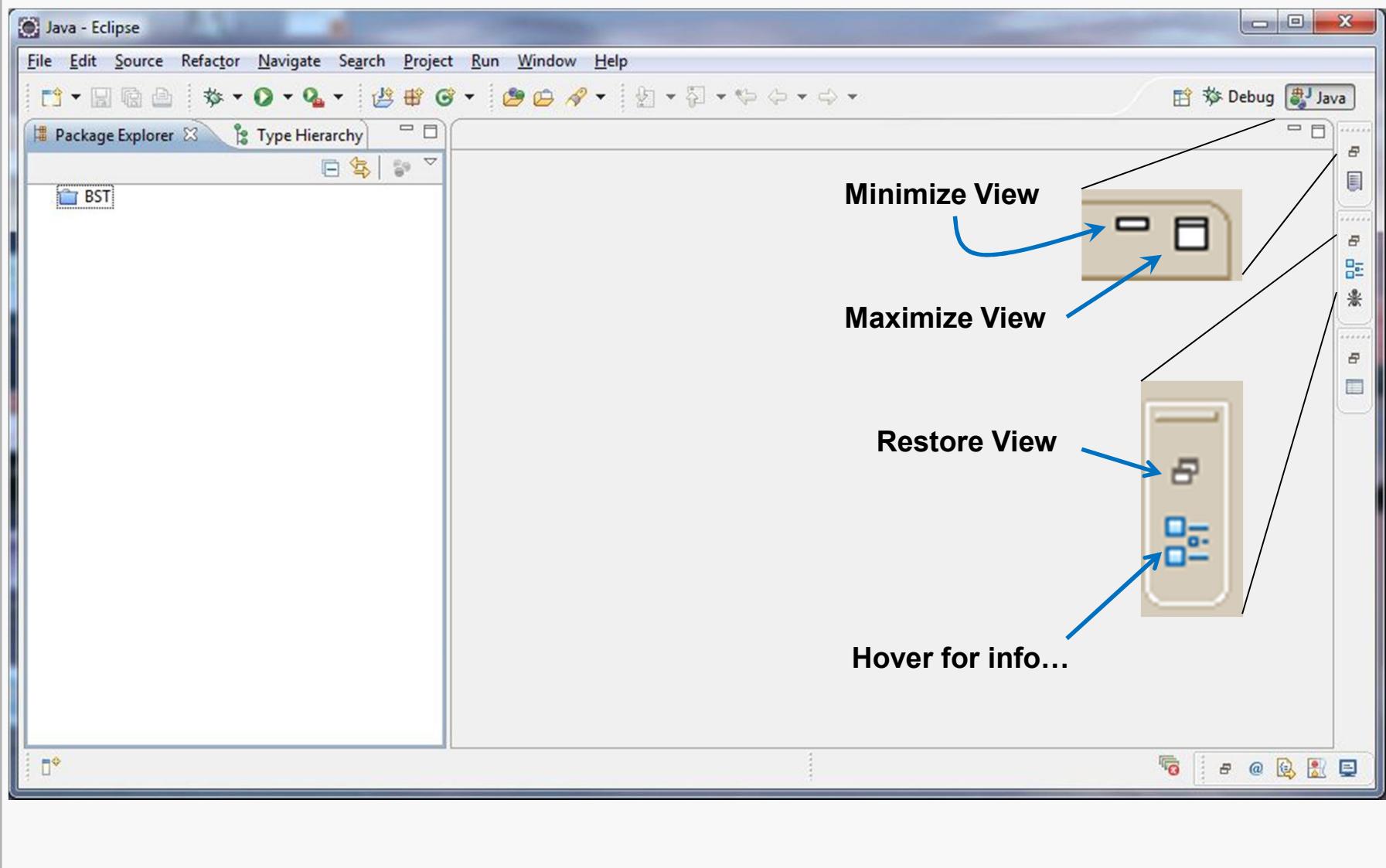
Since Eclipse includes its own Java compiler, it is not strictly necessary to have a version of the Java Development Kit (JDK) installed on your computer.

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/](http://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):

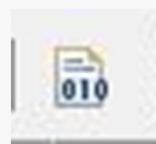




Choose a Perspective



New Project / Save / Save All / Print



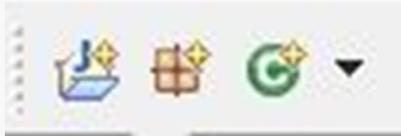
Build Project



Start Debugging + configurations

Run Project + configurations

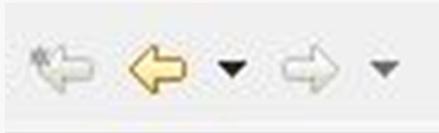
Run Last Tool + configurations



New Java Project / Package / Class



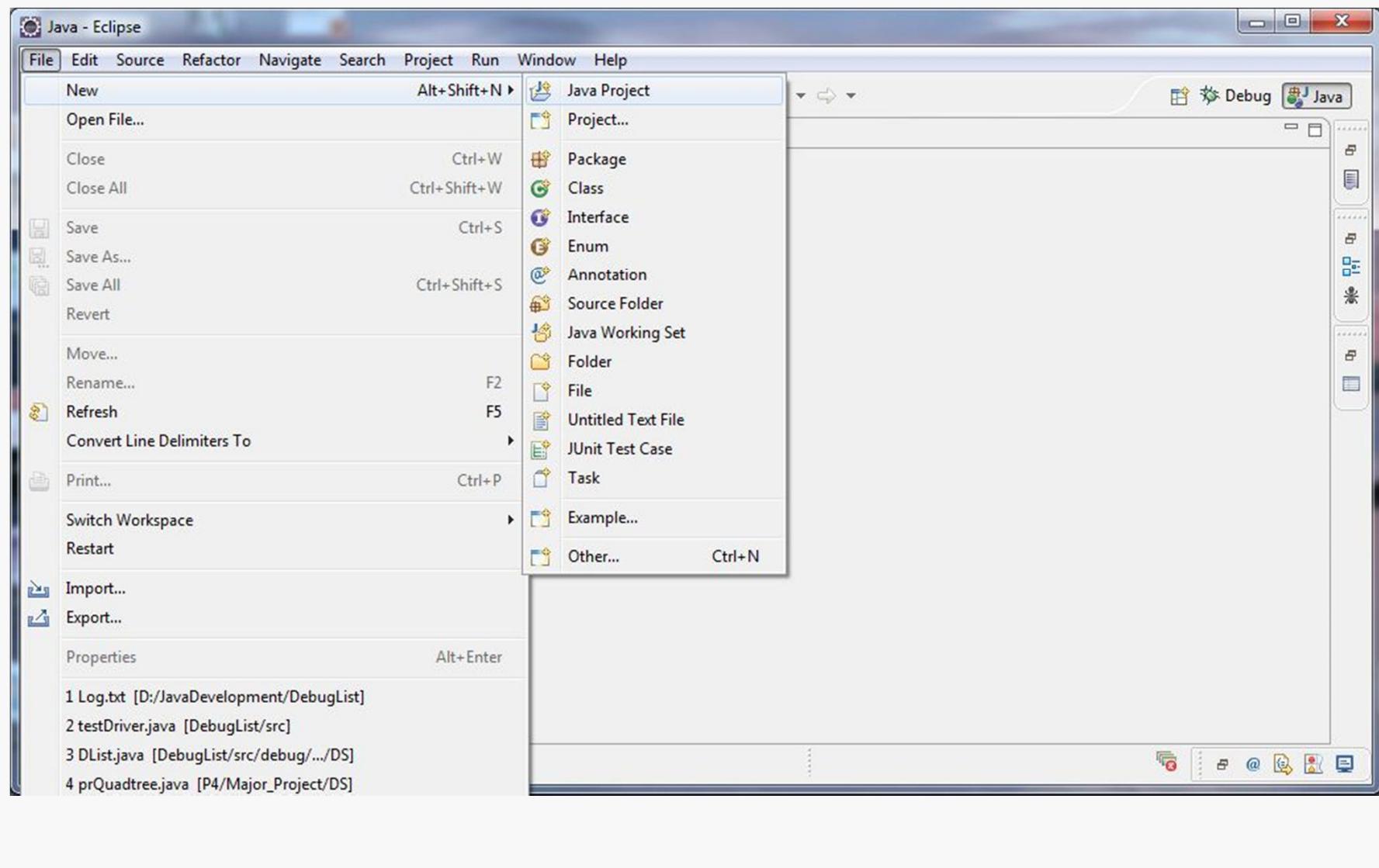
Open Type / Open Task / Search + options



Go to last edit location

Back/Next + more navigation options

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

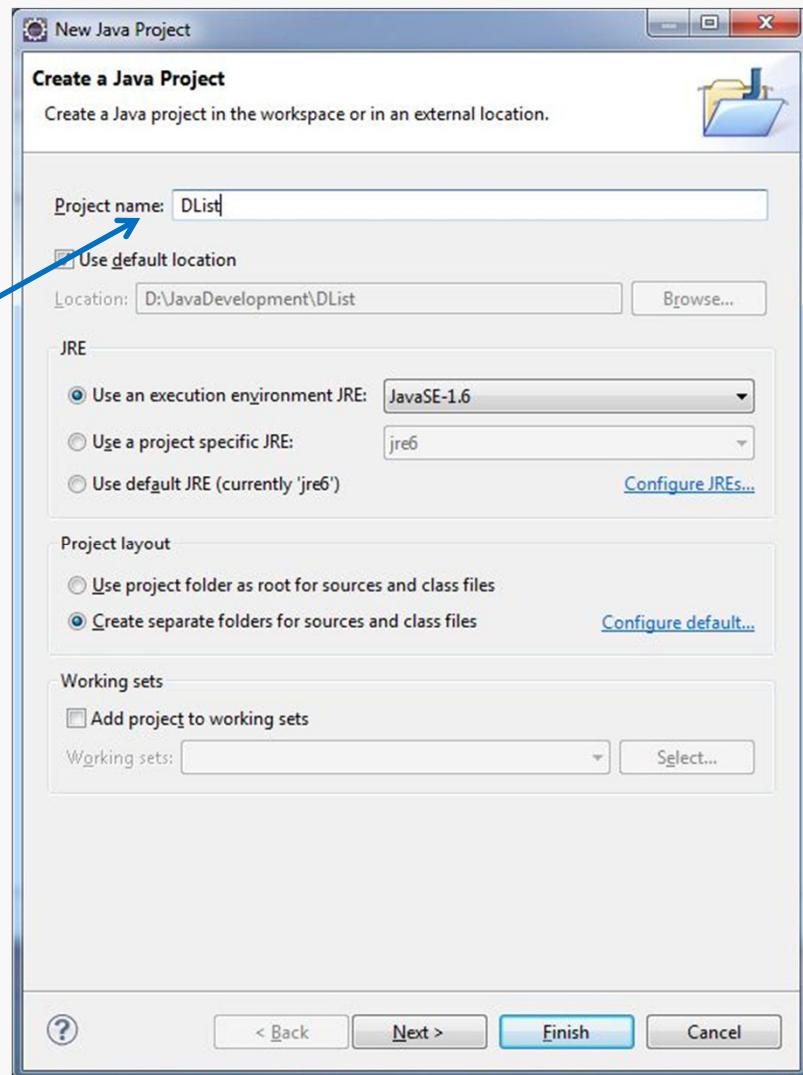


In the resulting dialog box:

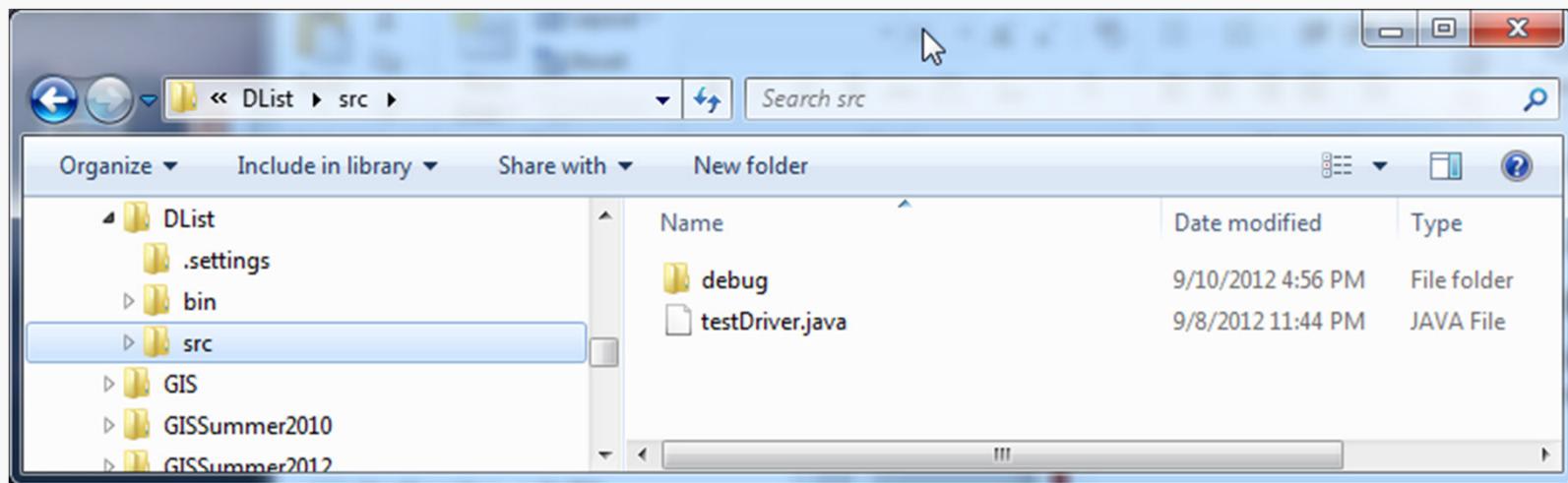
Enter a name for the Project.

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

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

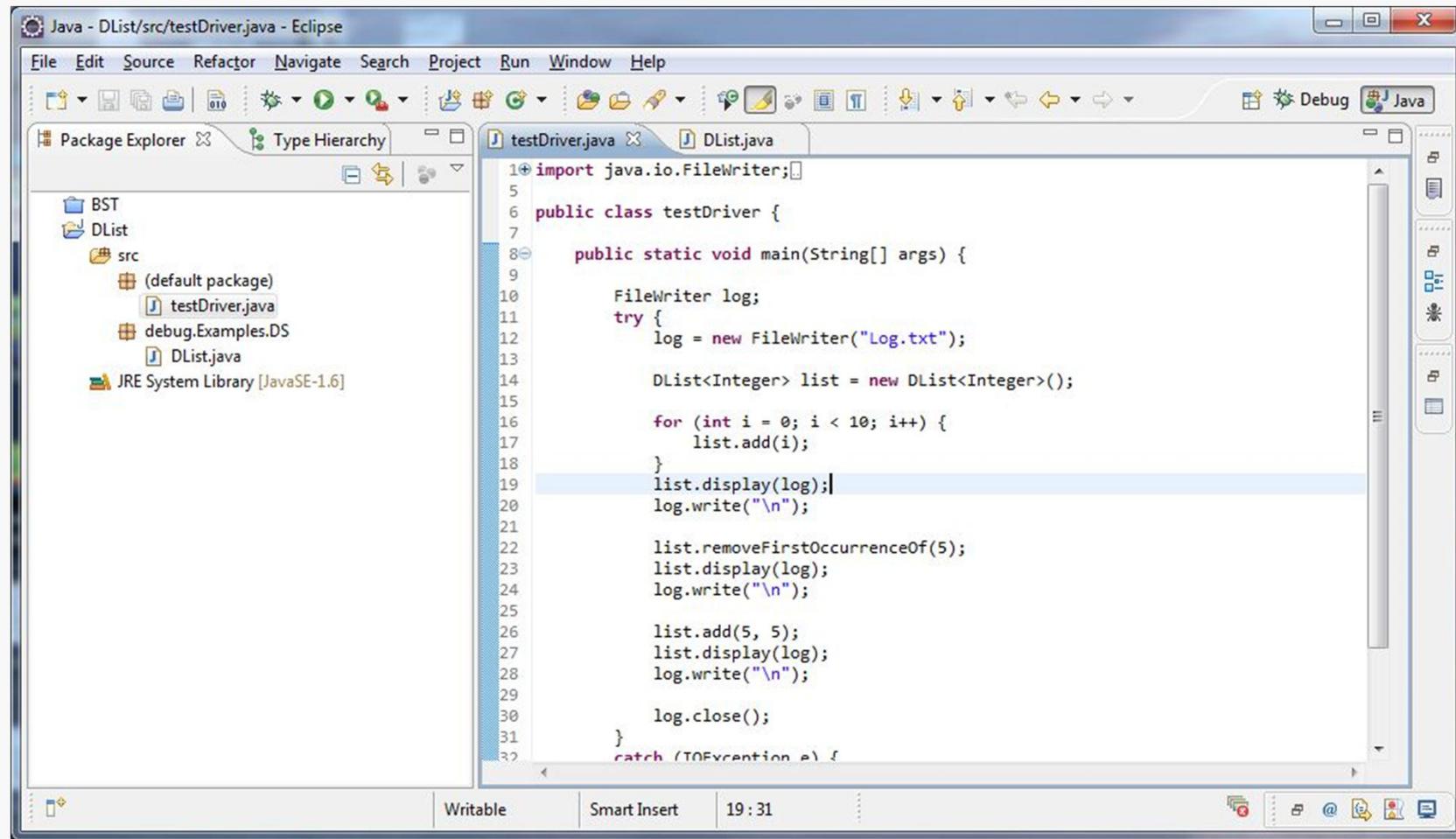


Download the file **DListExample.zip** from the course website Resources page, and place the contents into the **src** directory for the Eclipse project you just created:



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.



# Running the Program

Debugging 10

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.

The screenshot shows a Notepad++ window displaying the contents of a file named Log.txt. The file contains two sets of list displays, each preceded by a line number. The first set, labeled "display of initial list", shows a list from index 1 to 10 where each index is followed by a colon and a value (0 through 9). The second set, labeled "display of list after deleting 5", shows a list from index 12 to 20, which is identical to the first set except that index 5 is missing. The third set, labeled "display of list after reinserting 5", shows a list from index 22 to 30, which is identical to the second set except that index 5 has been reinserted with the value 6. The Notepad++ status bar at the bottom indicates the file has 171 length, 32 lines, and the cursor is at Ln:1 Col:1 Sel:0.

Index	Value
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

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:

*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)

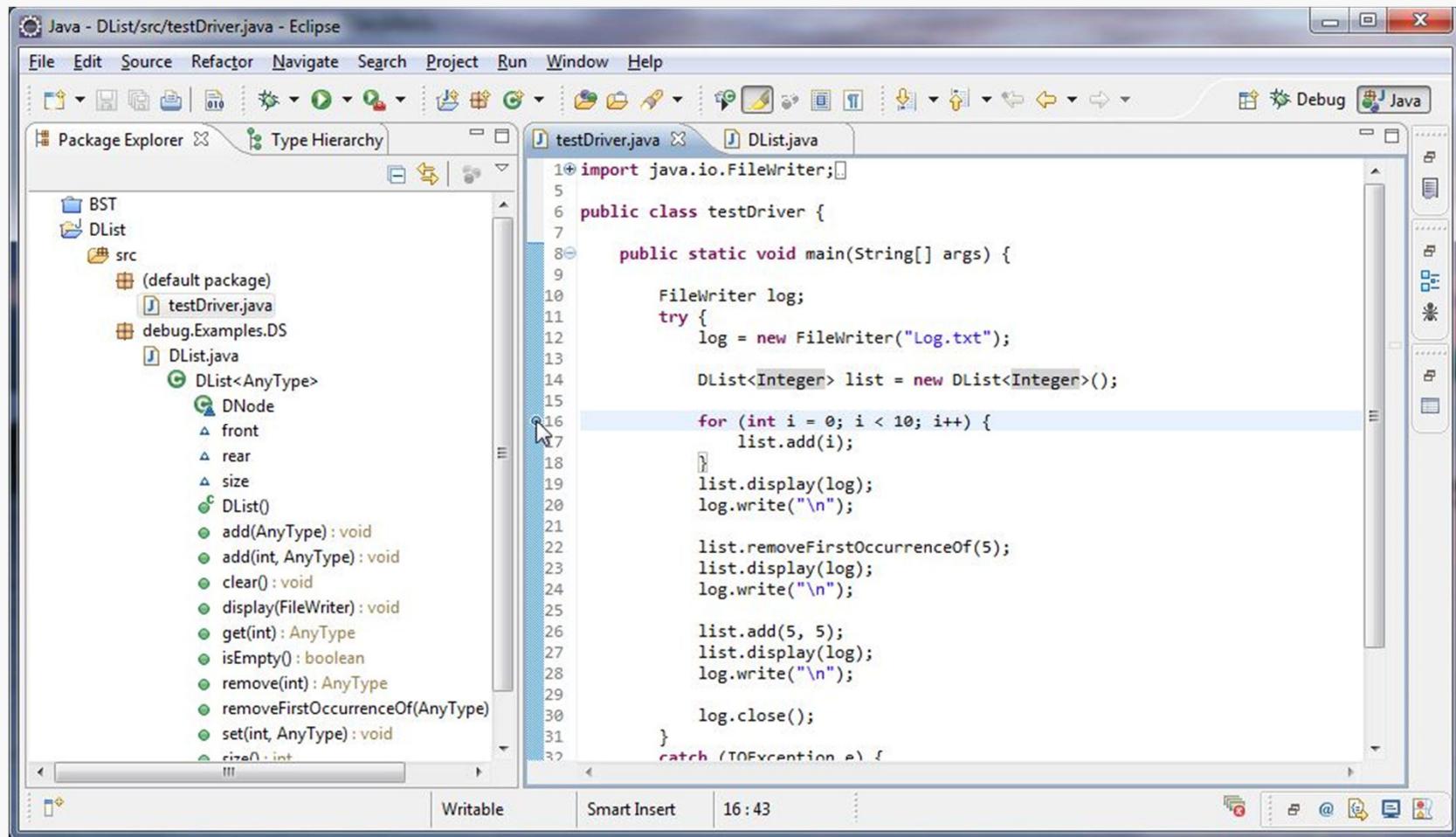
# Setting a Line Breakpoint

Debugging 13

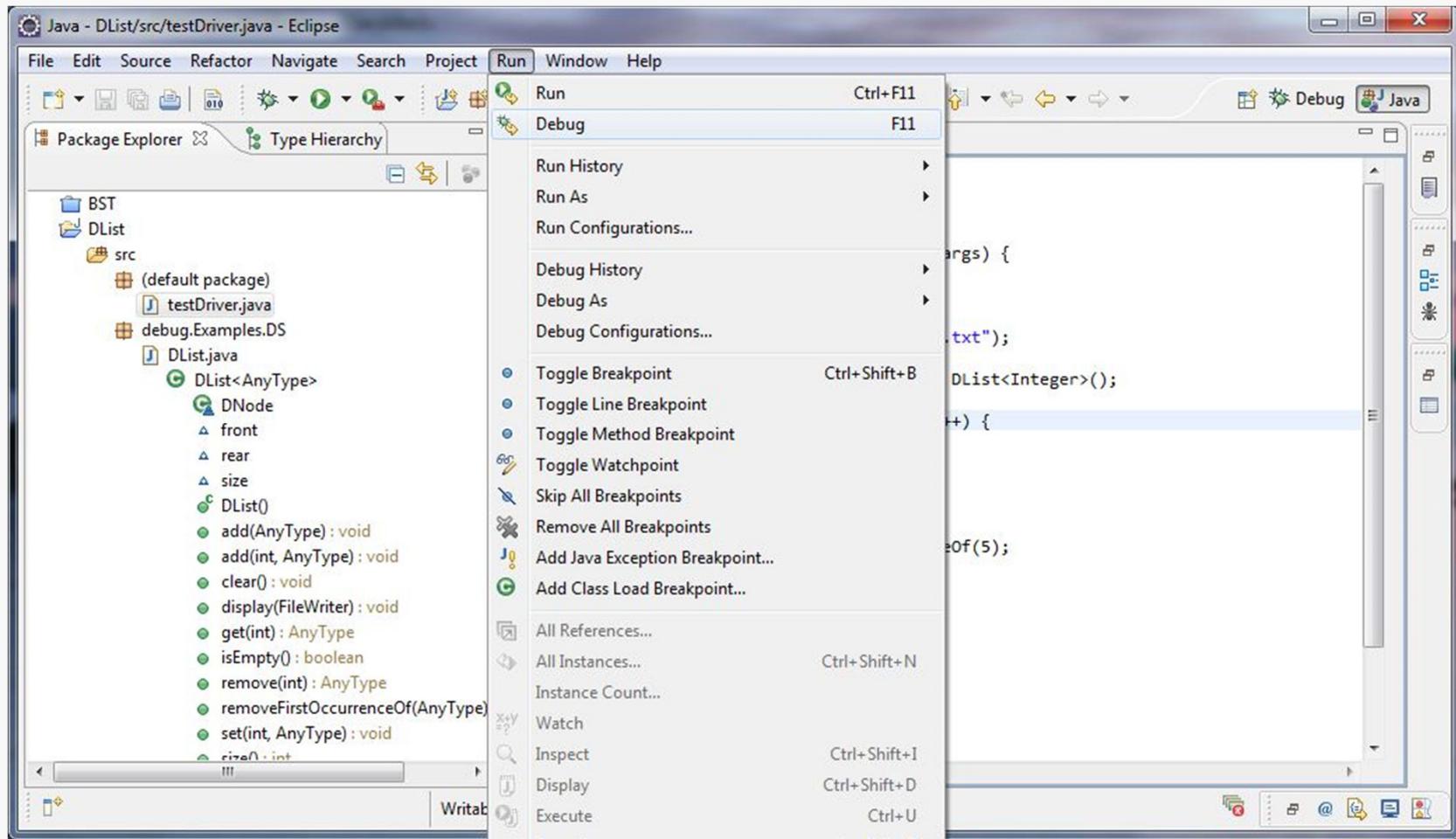
*line breakpoint*

halt when execution reaches a specific statement

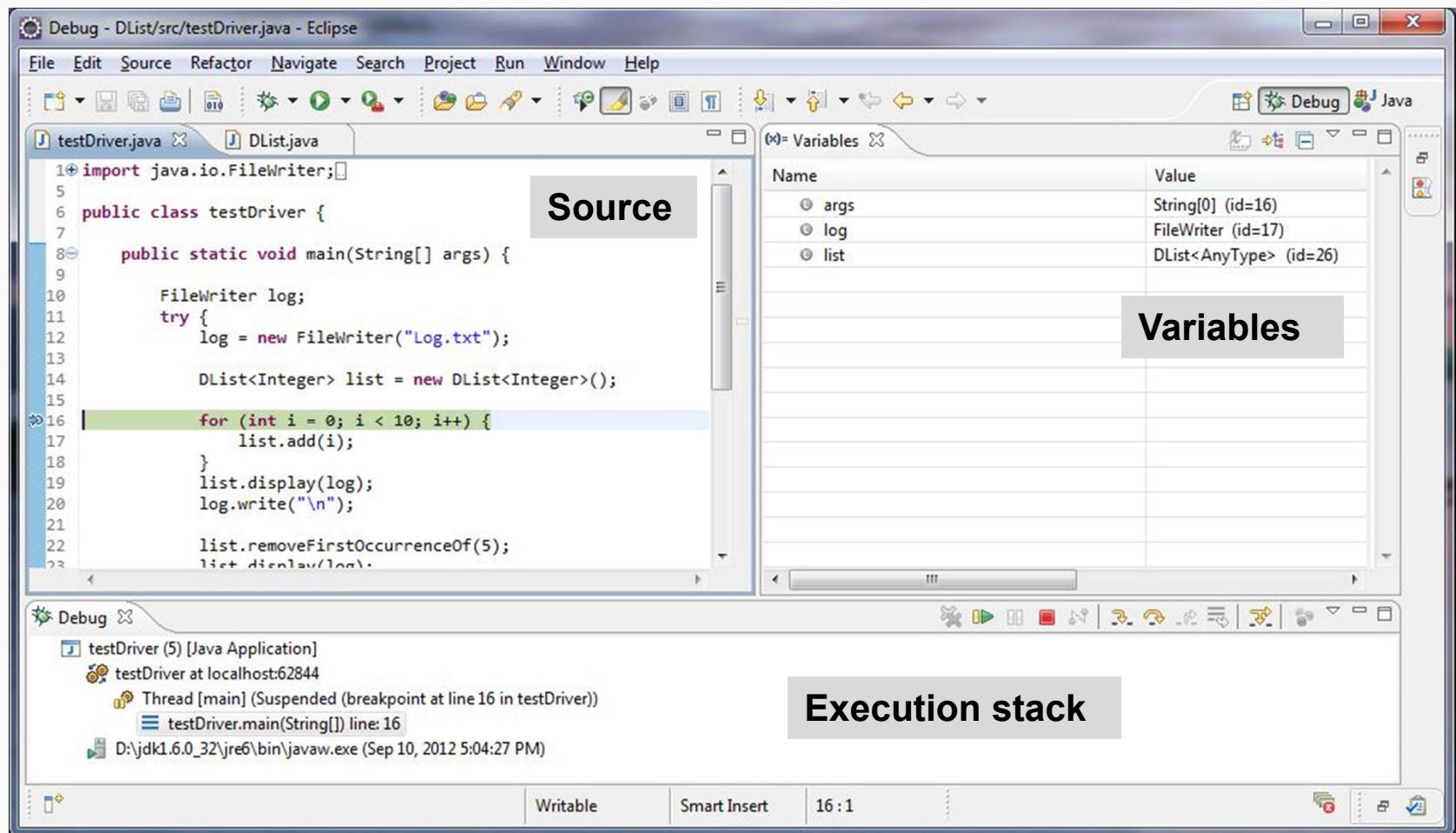
To set one, just double-click in the editor margin next to the selected line of code:



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.

# Using the Variables View

Debugging 16

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

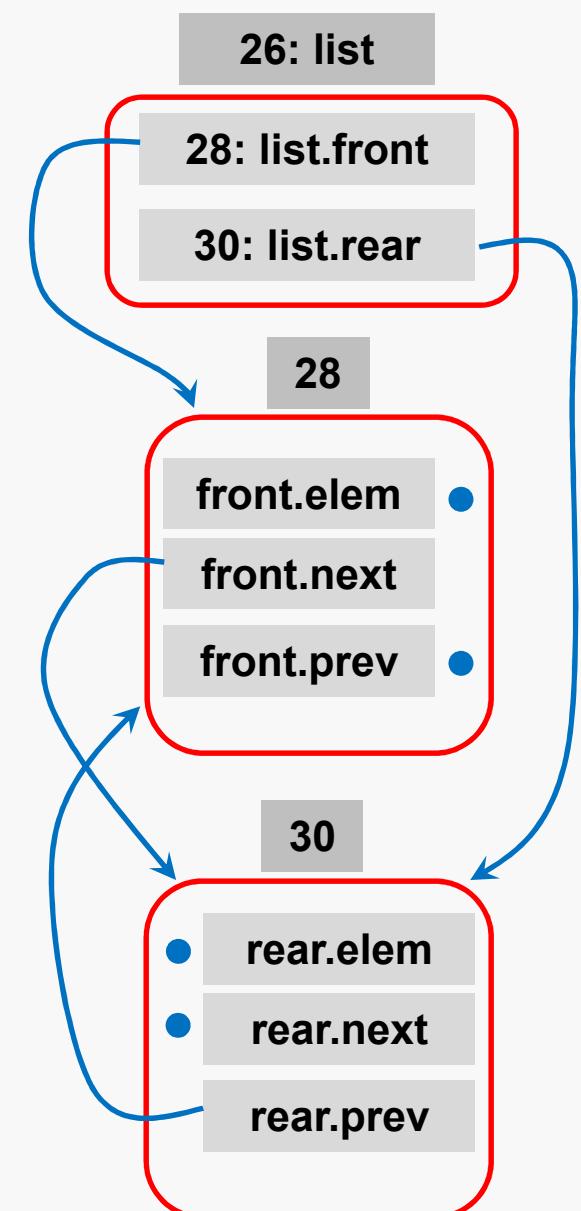
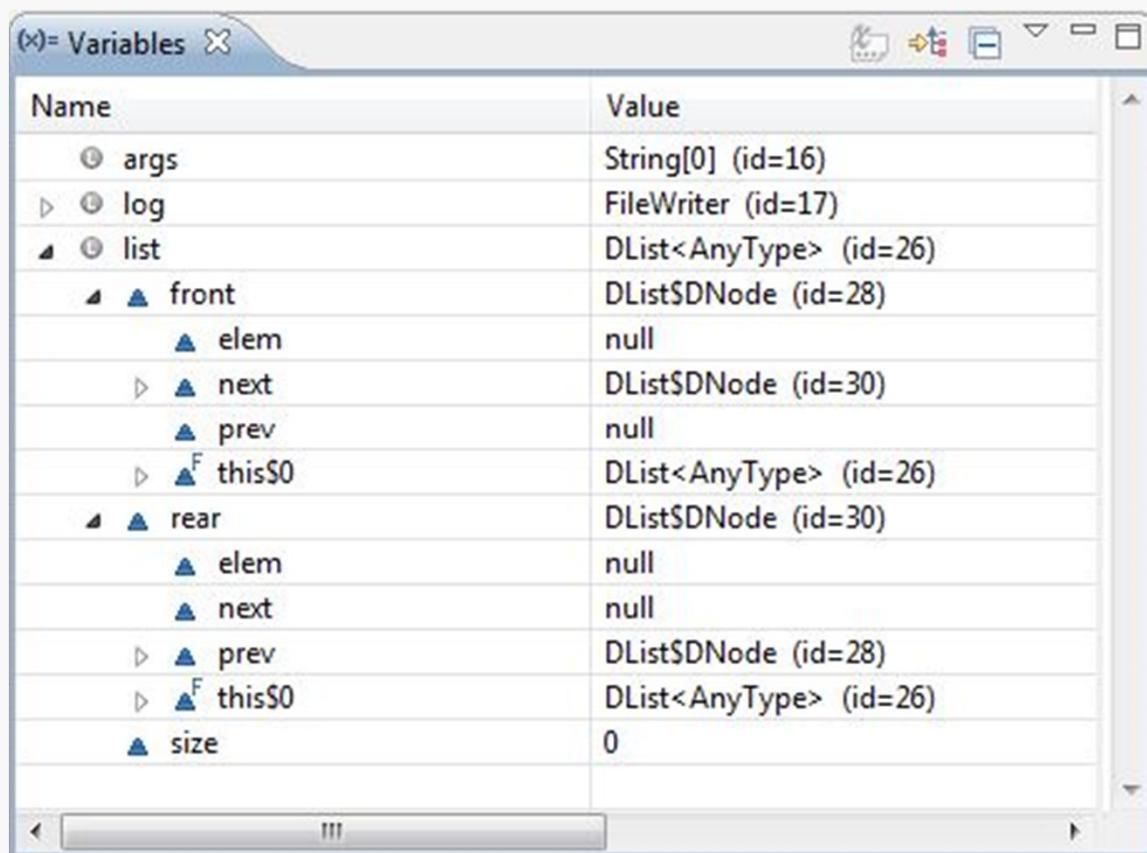
Name	Value
args	String[0] (id=16)
log	FileWriter (id=17)
list	DList<AnyType> (id=26)
front	DListSDNode (id=28)
rear	DListSDNode (id=30)
size	0

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

# Using the Variables View

Debugging 17

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.



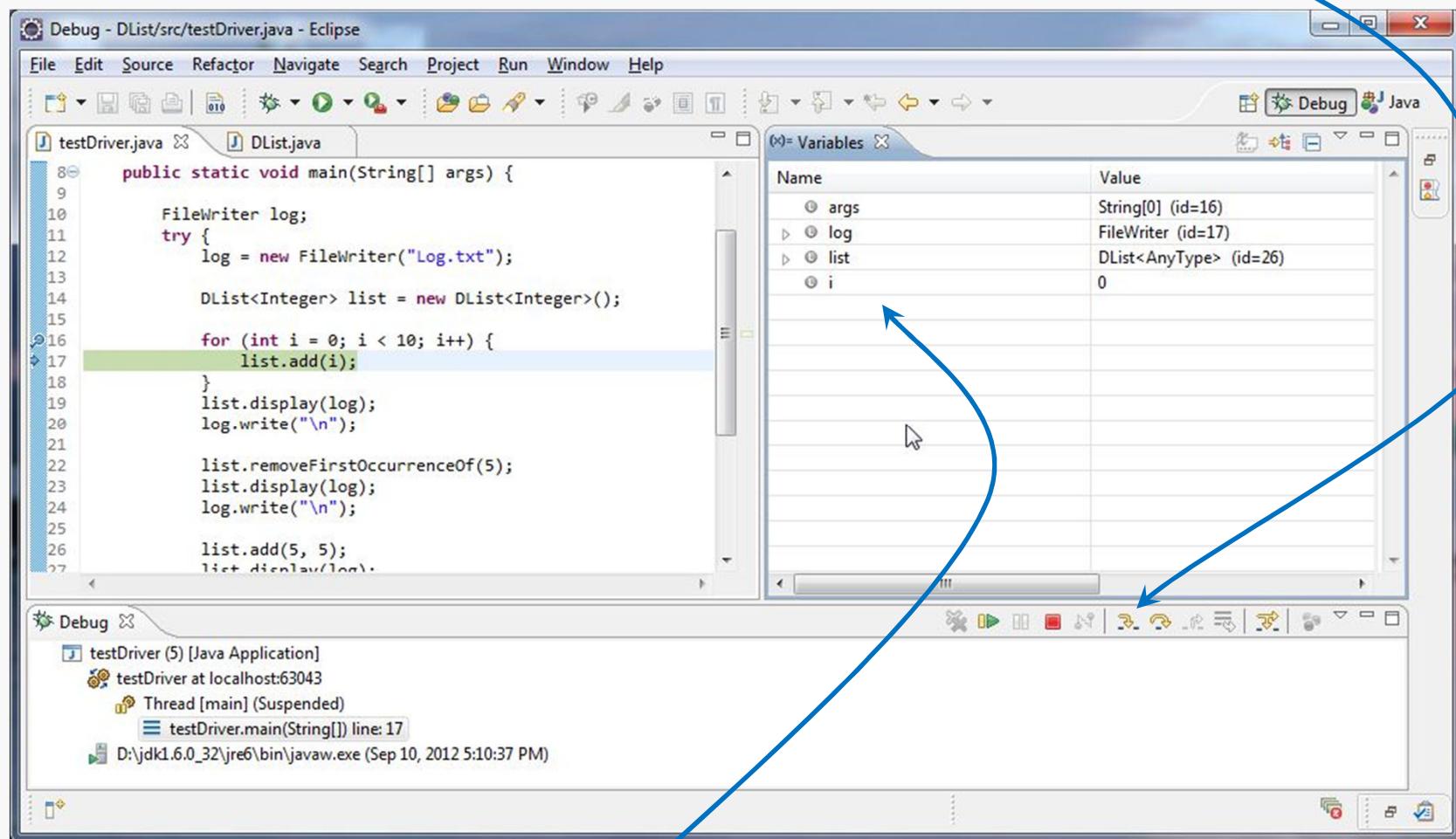
1    2    3    4    5    6    7    8    9    10

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

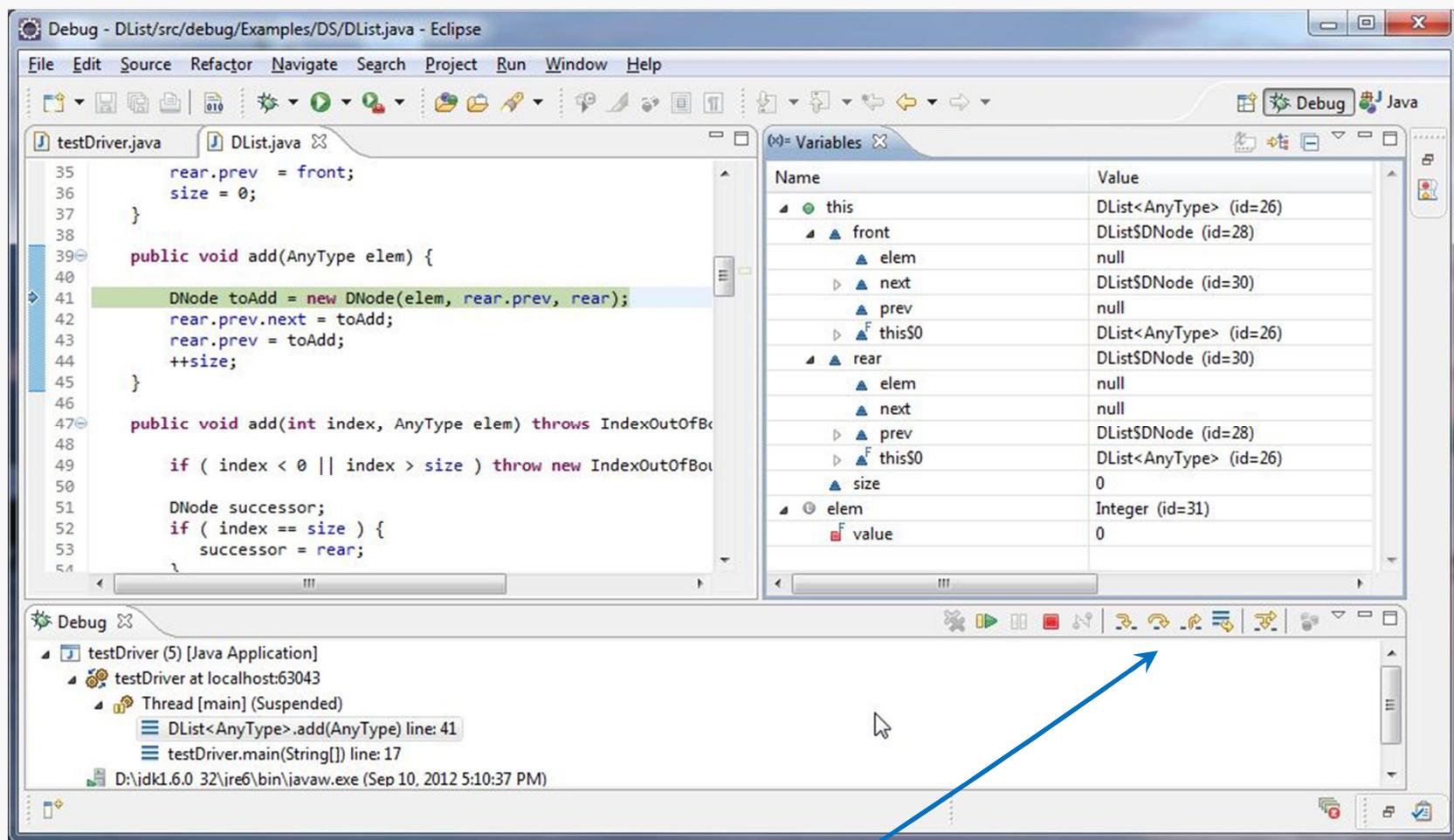
Debugging 19

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.

# Step-by-step Execution

Debugging 22

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

The screenshot shows the Eclipse IDE interface during a debug session. The code editor displays `DList.java` with the following code snippet highlighted:

```
        rear.prev = front;
        size = 0;
    }

    public void add(AnyType elem) {
        DNode toAdd = new DNode(elem, rear.prev, rear);
        rear.prev.next = toAdd;
        rear.prev = toAdd;
        ++size;
    }

    public void add(int index, AnyType elem) throws IndexOutOfBoundsException {
        if (index < 0 || index > size) throw new IndexOutOfBoundsException("Index " + index);
        DNode successor;
        if (index == size) {
            successor = rear;
        } else {
            successor = front.next;
            int i = 0;
            while (i < index) {
                successor = successor.next;
                ++i;
            }
        }
    }
}
```

The `Variables` view on the right shows the current state of variables:

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 interface during a debug session. The top bar displays "Debug - DList/src/debug/Examples/DS/DList.java - Eclipse". The left pane shows two files: "testDriver.java" and "DList.java". The code in "testDriver.java" is as follows:

```
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     } else {
55         successor = front.next;
56         int i = 0;
57         while (i < index) {
58             successor = successor.next;
59             ++i;
60         }
61     }
62 }
```

The right pane shows the "Variables" view. The variable "size" is highlighted in yellow, indicating it is the current focus. The table lists the following variables and their values:

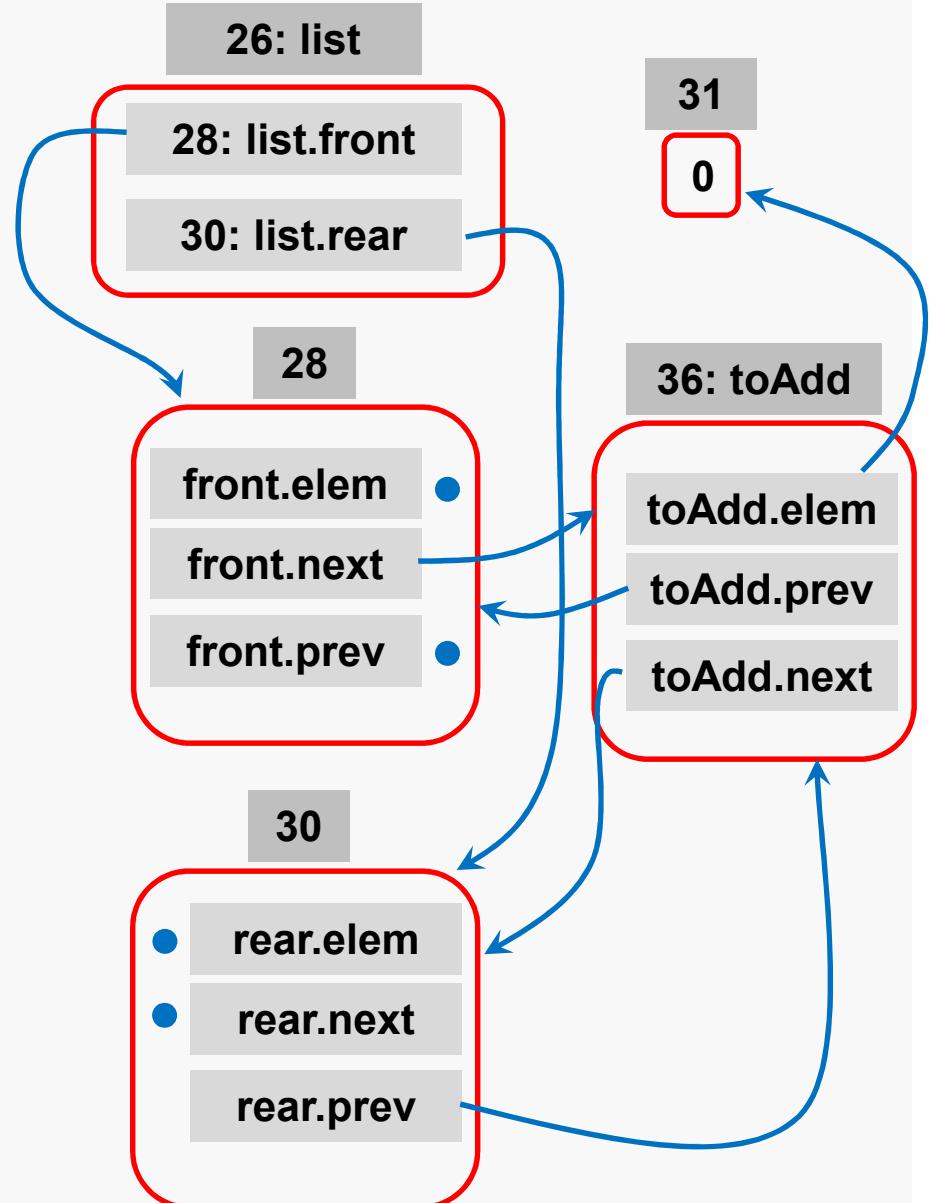
Name	Value
this	DList<AnyType> (id=26)
front	DList\$DNode (id=28)
elem	null
next	DList\$DNode (id=36)
prev	null
this\$0	DList<AnyType> (id=26)
rear	DList\$DNode (id=30)
elem	null
next	null
prev	DList\$DNode (id=36)
this\$0	DList<AnyType> (id=26)
size	1
elem	Integer (id=31)
value	0
toAdd	DList\$DNode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

# Checking the List Structure

Debugging 24

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)

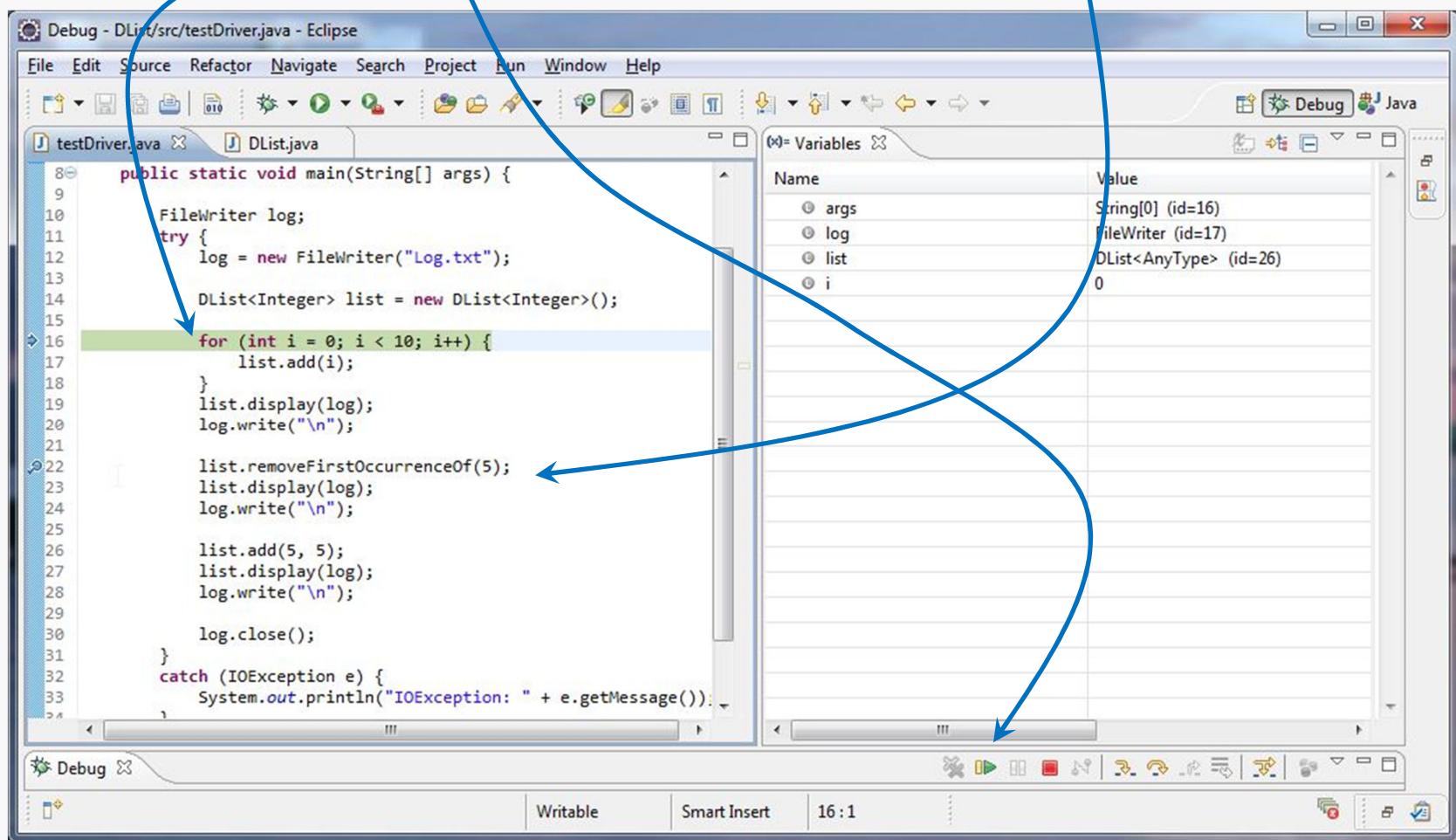
Well, that looks OK.



# Resetting Breakpoints and Resuming

Debugging 25

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

Debugging 26

Execution proceeds to the new breakpoint:

The screenshot shows the Eclipse IDE interface during a debugging session. The title bar says "Debug - DList/src/testDriver.java - Eclipse". The code editor displays the following Java code:

```
public static void main(String[] args) {
    ...
    try {
        log = new FileWriter("Log.txt");
        DList<Integer> list = new DList<Integer>();
        for (int i = 0; i < 10; i++) {
            list.add(i);
        }
        list.display(log);
        log.write("\n");
        list.removeFirstOccurrenceOf(5);
        list.display(log);
        log.write("\n");
        list.add(5, 5);
        list.display(log);
        log.write("\n");
        log.close();
    } catch (IOException e) {
        System.out.println("IOException: " + e.getMessage());
    }
}
```

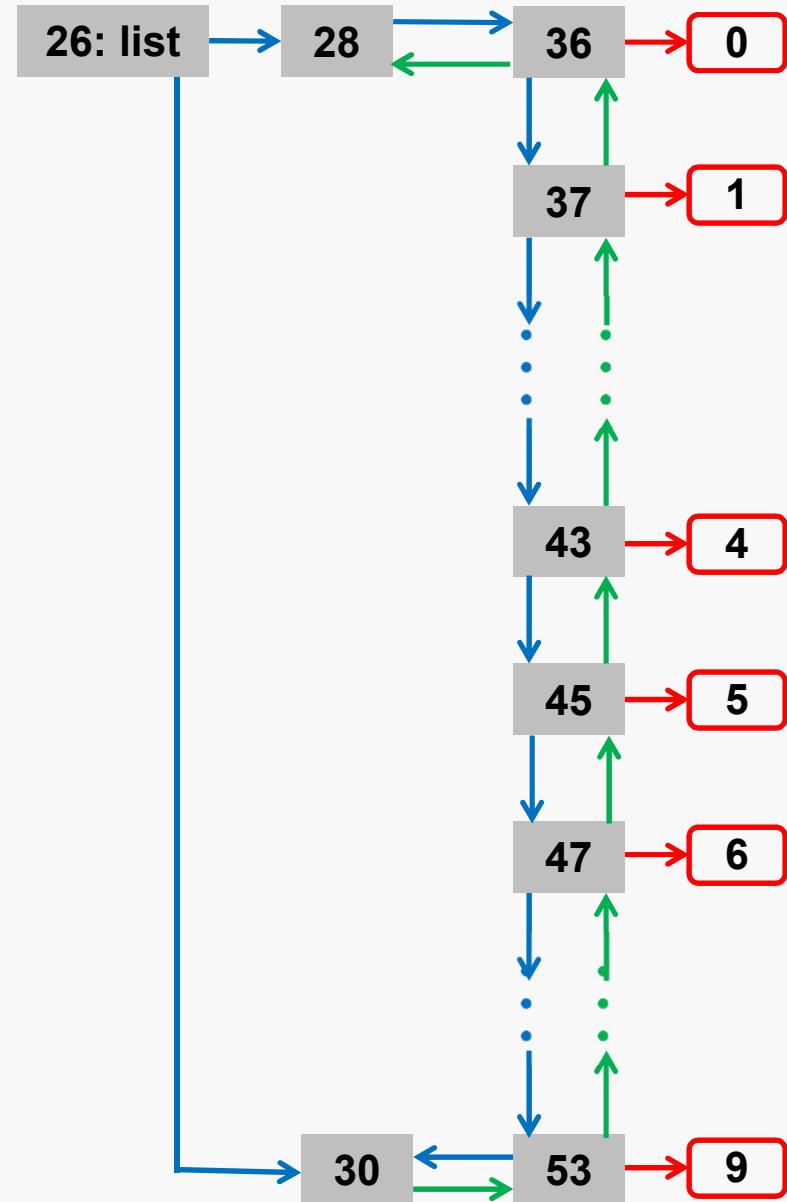
A blue arrow points from the text "Execution proceeds to the new breakpoint:" to the line "list.removeFirstOccurrenceOf(5);". The variables view on the right shows the current state of variables:

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	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=45)
prev	DList\$DNode (id=41)
this\$0	DList<AnyType> (id=26)
prev	DList\$DNode (id=39)
this\$0	DList<AnyType> (id=26)
prev	DList\$DNode (id=37)
this\$0	DList<AnyType> (id=26)
prev	DList\$DNode (id=36)



# Step Into removeFirstOccurrenceOf()

Debugging 28

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 interface during a debug session. The top menu bar includes File, Edit, Source, Refactor, Navigate, Search, Project, Run, Window, Help, and tabs for testDriver.java and DList.java. The code editor on the left displays Java code for a DList class, specifically the removeFirstOccurrenceOf method. A cursor is positioned at the start of the while loop. The 'Variables' view on the right lists local variables and their values. The current variable is highlighted, showing it is a DListNode object with an integer value of 5. Other variables listed include this (DList), front (DListNode), rear (DListNode), size (int), elem (Integer), current (DListNode), and current's elem (Integer).

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

# In removeFirstOccurrenceOf()

Debugging 29

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

The screenshot shows the Eclipse IDE interface during a debugging session. The code editor displays the `DList.java` file with the following code:

```
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;
}
```

The line `130 while ( current != rear ) {` is highlighted in green, indicating it is the current line of execution. The `Variables` view on the right shows the state of variables:

Name	Value
this	DLList<AnyType> (id=26)
front	DLList\$DNode (id=28)
rear	DLList\$DNode (id=30)
size	10
elem	Integer (id=46)
value	5
current	DLList\$DNode (id=45)
elem	Integer (id=46)
value	5
next	DLList\$DNode (id=47)
prev	DLList\$DNode (id=43)
this\$0	DLList<AnyType> (id=26)

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

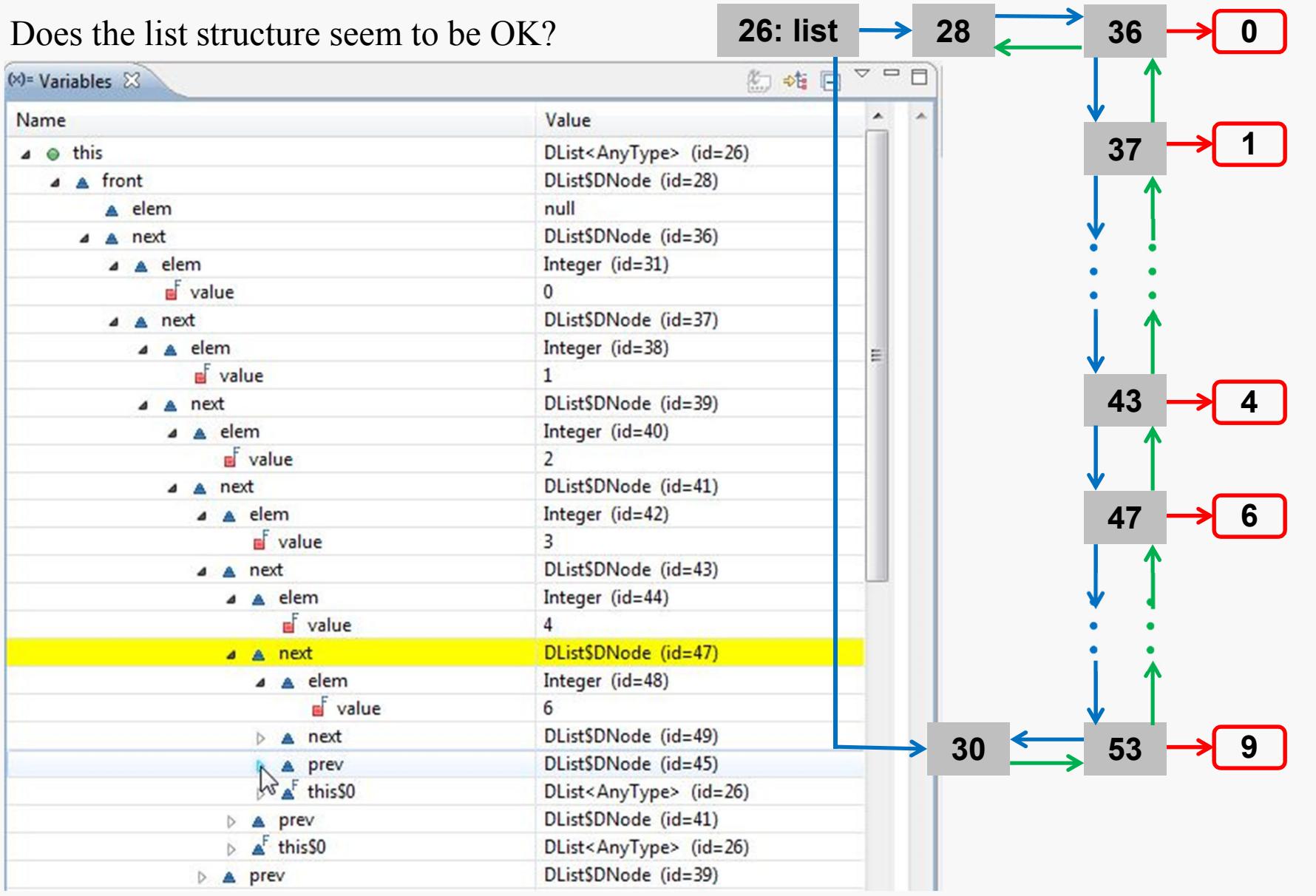
The screenshot shows the Eclipse IDE interface during a debug session. The left pane displays the code for `DList.java`, specifically focusing on the `removeFirstOccurrenceOf` method. The line of code being executed is highlighted in green: `return toReturn;`. The right pane is the `Variables` view, which lists the current state of variables. A yellow highlight is applied to the `next` variable of the `DList$DNode` object with `id=47`, which contains the value `6`. This indicates that the loop has just completed one iteration where it found a matching element and updated its `next` pointer.

Name	Value
<code>this</code>	<code>DList&lt;AnyType&gt; (id=26)</code>
<code>front</code>	<code>DList\$DNode (id=28)</code>
<code>elem</code>	<code>null</code>
<code>next</code>	<code>DList\$DNode (id=36)</code>
<code>  elem</code>	<code>Integer (id=31)</code>
<code>  value</code>	<code>0</code>
<code>  next</code>	<code>DList\$DNode (id=37)</code>
<code>    elem</code>	<code>Integer (id=38)</code>
<code>    value</code>	<code>1</code>
<code>    next</code>	<code>DList\$DNode (id=39)</code>
<code>      elem</code>	<code>Integer (id=40)</code>
<code>      value</code>	<code>2</code>
<code>      next</code>	<code>DList\$DNode (id=41)</code>
<code>        elem</code>	<code>Integer (id=42)</code>
<code>        value</code>	<code>3</code>
<code>        next</code>	<code>DList\$DNode (id=43)</code>
<code>          elem</code>	<code>Integer (id=44)</code>
<code>          value</code>	<code>4</code>
<code>          next</code>	<code>DList\$DNode (id=47)</code>
<code>            elem</code>	<code>Integer (id=48)</code>
<code>            value</code>	<code>6</code>
<code>            next</code>	<code>DList\$DNode (id=49)</code>
<code>              elem</code>	<code>Integer (id=50)</code>
<code>              value</code>	<code>7</code>
<code>              next</code>	<code>DList\$DNode (id=51)</code>
<code>              prev</code>	<code>DList\$DNode (id=47)</code>
<code>              this\$0</code>	<code>DList&lt;AnyType&gt; (id=26)</code>
<code>              prev</code>	<code>DList\$DNode (id=45)</code>

# List Details

Debugging 31

Does the list structure seem to be OK?

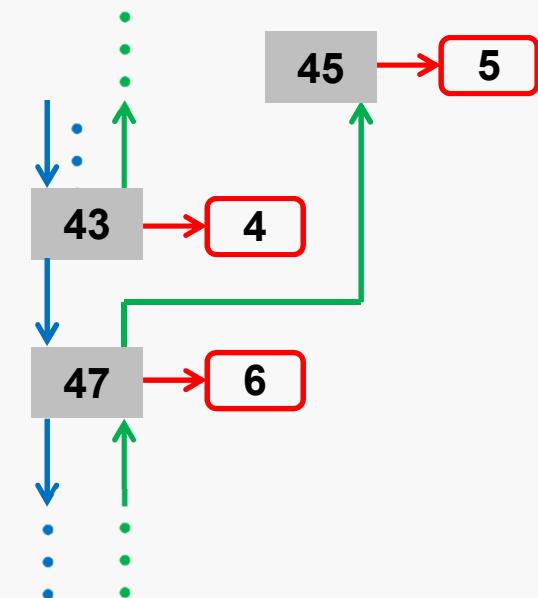


# Bugged List Structure (more detail)

Debugging 32

A careful examination indicates that something odd has happened:

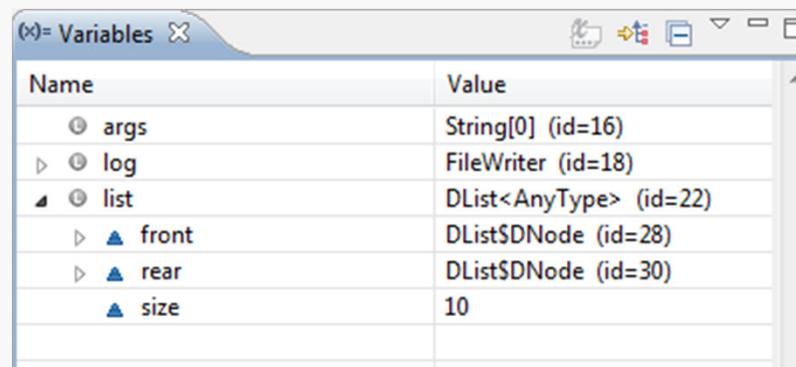
next	DList\$DNode (id=43)
elem	Integer (id=44)
value	4
next	DList\$DNode (id=47)
elem	Integer (id=48)
value	6
next	DList\$DNode (id=49)
prev	DList\$DNode (id=45)
elem	Integer (id=46)
value	5
next	DList\$DNode (id=47)



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



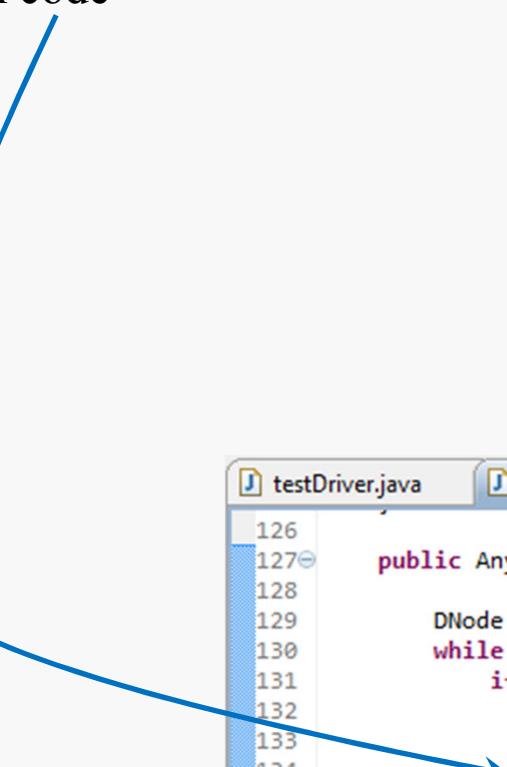
The screenshot shows the Eclipse IDE's Variables view window titled '(x)= Variables'. The table lists the following variables and their values:

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

# A Look at the Code

Debugging 34

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



```
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 }
```

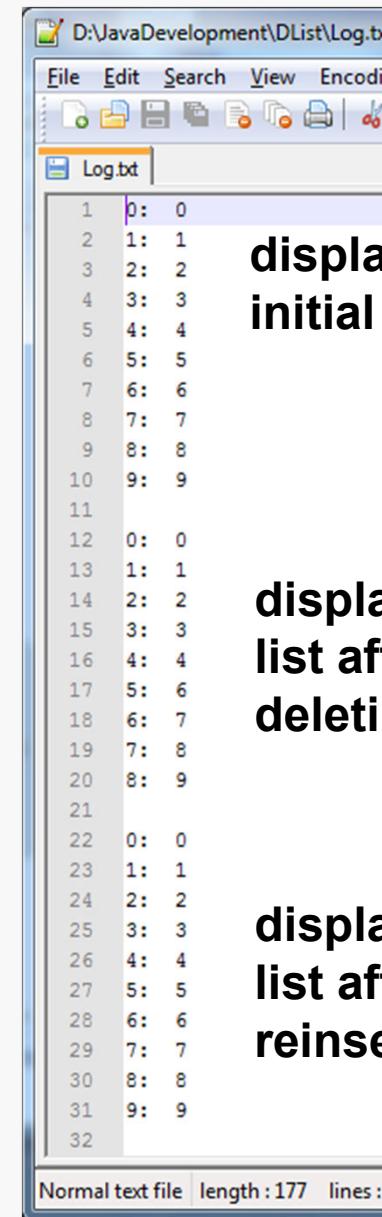


```
126
127     public AnyType removeFirstOccurrenceOf(AnyType elem) {
128
129         DNode current = front.next;
130         while ( current != rear ) {
131             if ( elem.equals(current.elem) ) {
132                 AnyType toReturn = current.elem;
133                 current.prev.next = current.next;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

## Testing Again

Debugging 35

Let's execute the modified program:



The screenshot shows a Windows Notepad window with the title 'D:\JavaDevelopment\DLList\Log.txt'. The window displays three distinct sections of a list, each preceded by a line number:

- display of initial list:** Lines 1 through 11 show a sequence from 0 to 9, each preceded by its index (e.g., 1: 0, 2: 1, ..., 10: 9).
- display of list after deleting 5:** Lines 12 through 20 show the list from 0 to 9 again, but the entry for index 5 is missing (5: is present but has no value).
- display of list after reinserting 5:** Lines 21 through 32 show the list from 0 to 9, with the entry at index 5 now having the value 5 (5: 5).

At the bottom of the window, it says 'Normal text file | length :177 lines :'. The entire window is labeled with bold text indicating the stages of list modification.

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

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

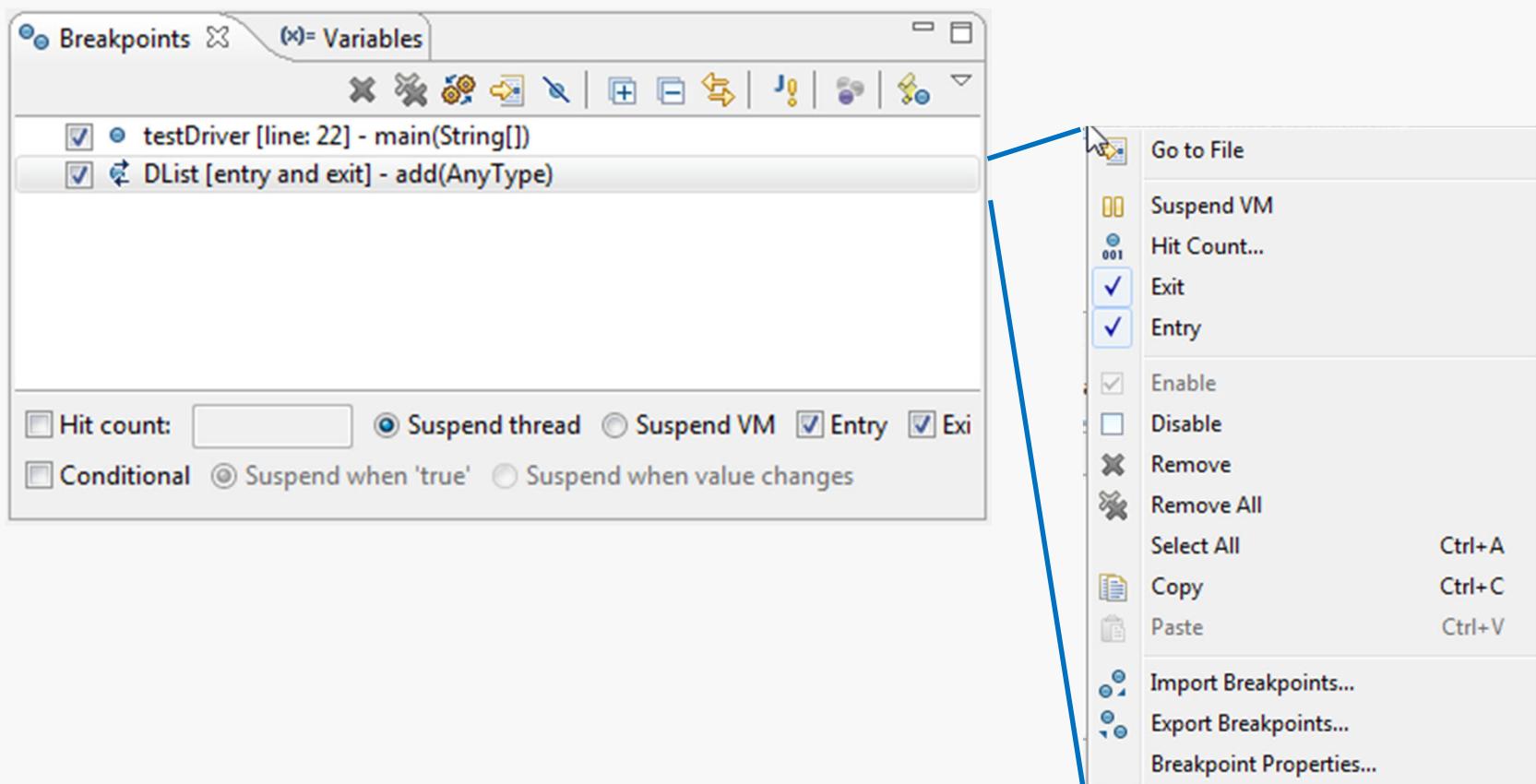
To set one, just double-click in the editor margin next to the method header:

```
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
```

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

# Exception Breakpoints

Debugging 39

