Chulalongkorn University
International School of Engineering
Department of Computer Engineering
2140105 Computer Programming Lab.

Name _____
Student ID _____
Station No. _____
Date _____

Lab 3 - Debugging

Objectives:

- Understand various debugging techniques.
- Learn to use Eclipse's Java Scrapbook to view, evaluate, and run Java expression.
- Learn to use Eclipse's Java debugger.
- Use debugger to control the execution of the program.

1. Scrapbook

The Java Development Toolkit (JDT) contributes a scrapbook facility that can be used to experiment and evaluate Java code snippets before building a complete Java program. Snippets are edited and evaluated in the Scrapbook page editor, with resultant problems reported in the editor.
From a Java scrapbook editor, you can select a code snippet, evaluate it, and display the result as a string. You can also show the object that result from evaluating a code snippet in the debuggers' Expressions View.

Creating a Java Scrapbook Page

The scrapbook allows Java expressions, to be run, inspected, and displayed under the control of the debugger. Breakpoints and exceptions behave as they do in a regular debug session.
Code is edited on a scrapbook page. A VM is launched for each scrapbook page in which expressions are being evaluated. The first time an expression is evaluated in a scrapbook page after it is opened, a VM is launched. The VM for a page will remain active until the page is closed, terminated explicitly (in the debugger or via the Stop the Evaluation button in the editor toolbar), or when a **`System.exit()`** is evaluated.
There are several ways to open the New Java Scrapbook Page wizard.

- Create a file with a **`.jpage`** extension
- From the menu bar, select **File > New > Other...**. Then select **Java > Java Run/Debug > Scrapbook Page**. Then click **Next**.

Once you've opened the New Java Scrapbook Page wizard:.

1. In the Enter or select the folder field, type or click **Browse** to select the container for the new page.
2. In the File name field, type a name for the new page as in Figure 1. The **`.jpage`** extension will be added automatically if you do not type it yourself.

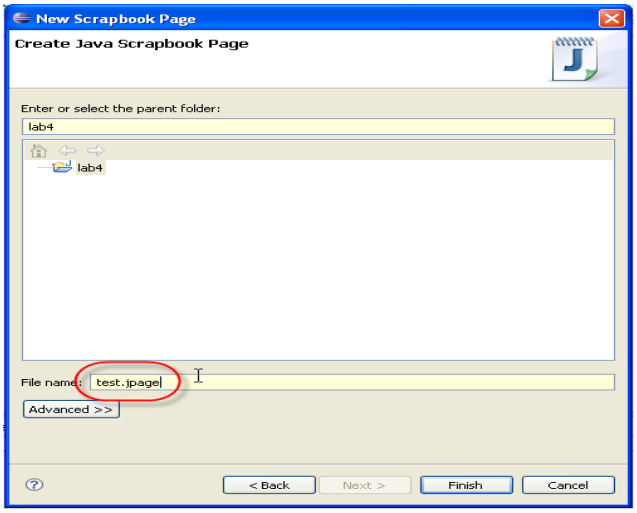


Figure 1 New Scrapbook Page

3. Click **Finish** when you are done. The new scrapbook page opens in an editor as in Figure 2.

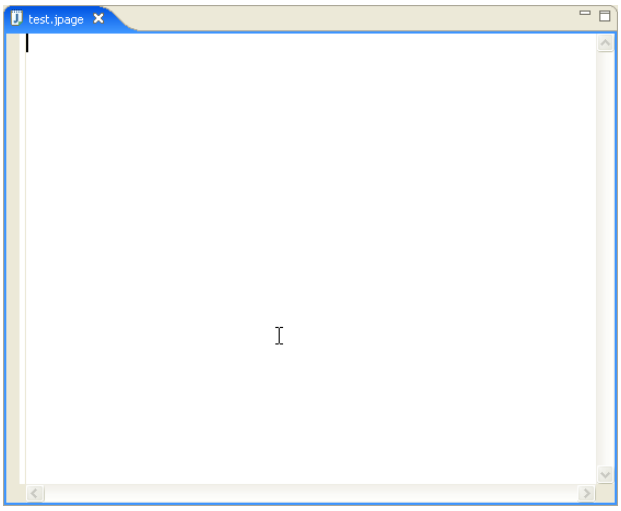


Figure 2 Scrapbook Editor

Inspecting the result of evaluating an expression

Inspecting shows the result of evaluating an expression in the Expressions view.

1. In the scrapbook page, either type an expression or highlight an existing expression to be inspected. Figure 3 shows an example: **`System.getProperties();`**

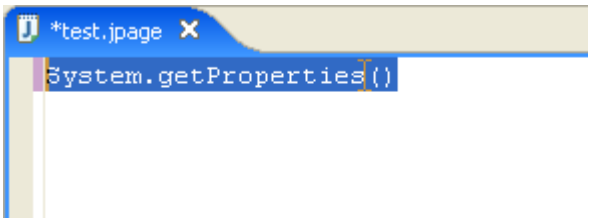


Figure 3 An expression in Scrapbook editor

2. Click the **Inspect** button in the toolbar (or select Inspect from the selection's pop-up menu) as in Figure 4.

Figure 4 Inspection Button

3. The result of the inspection appears in a pop-up as shown in Figure 5.
4. The result can be inspected like a variable in the debugger (for example, children of the result can be expanded).

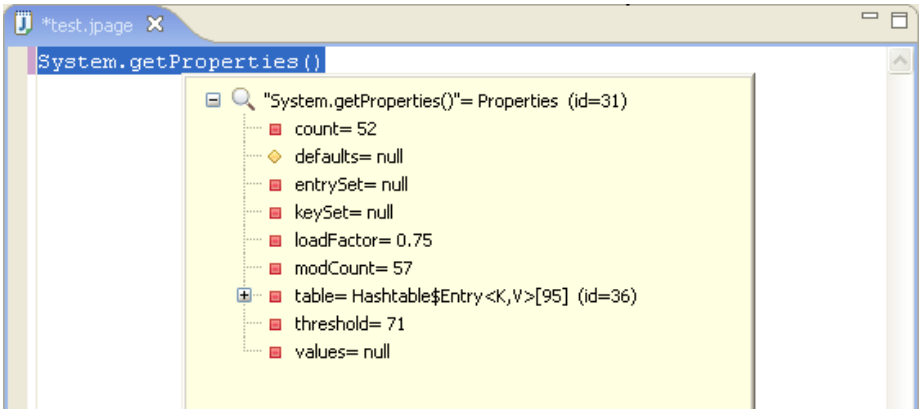


Figure 5 Inspection's Result

Displaying the result of evaluating an expression

Displaying shows the result of evaluating an expression in the scrapbook editor.

1. In the scrapbook page, either type an expression or highlight an existing expression to be displayed. For example: **`System.getProperties();`**
2. Click the **Display** button (as shown in Figure 6) in the toolbar (or select **Display** from the selection's pop-up menu.)

Figure 6 Display Button

3. The result of the evaluation appears highlighted in the scrapbook editor as shown in Figure 7. The result displayed is either
   - the value obtained by sending **`toString()`** to the result of the evaluation, or
   - when evaluating a primitive data type (e.g., an **`int`**), the result is the simple value of the result.

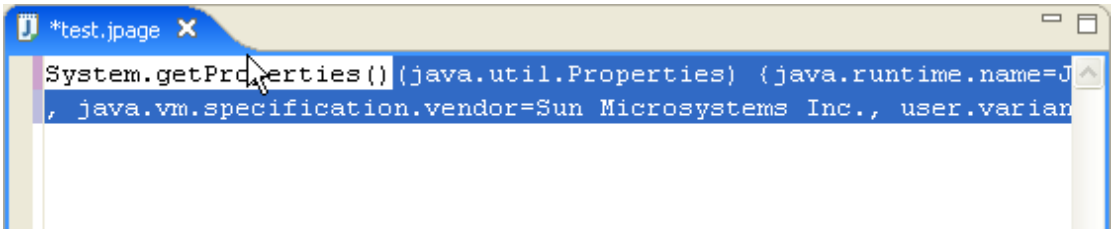


Figure 7 The result of the evaluation

For example:

- Type and highlight new **`java.util.Date()`** in the editor, and click **Display**. A result such as **`(java.util.Date) Tue Jun 12 14:03:17 CDT 2001`** appears in the editor.
- As another example, type and highlight **`3 + 4`** in the editor, and press **Display**. The result **`(int) 7`** is displayed in the editor.

Executing an expression

Executing an expression evaluates an expression but does not display a result.
If you select the expression to execute and click the Execute button in the toolbar (see Figure 8), no result is displayed, but the code is executed.

Figure 8 Execute Button

For example, if you type and highlight **`System.out.println("Hello World"),`** and click the **Execute** button, Hello World appears in the Console view, but no result is displayed in the scrapbook editor or the Expressions view.

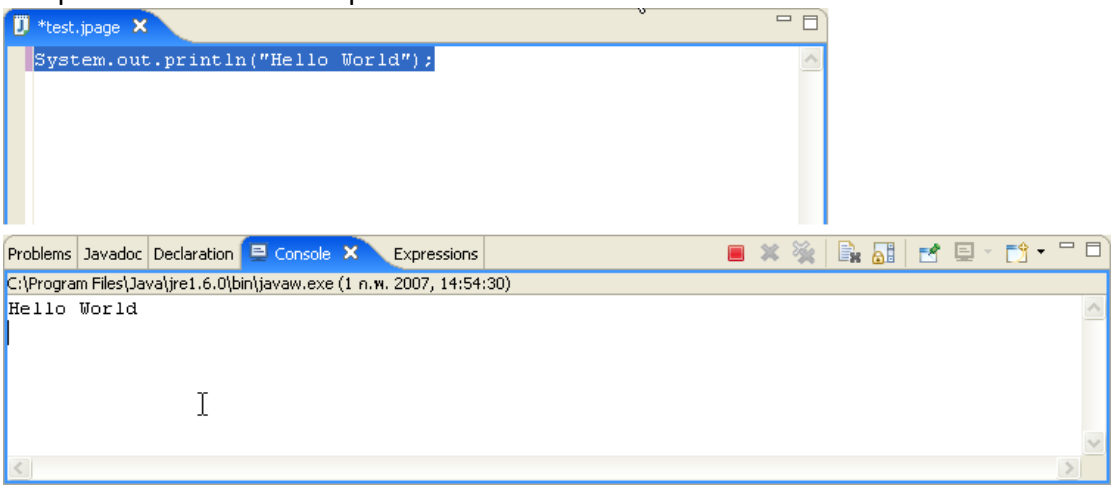


Figure 9 The result of execution in Console View

Your turn

1. Using Java Scrapbook in Eclipse to
   1.1. Evaluate the following expressions:
   - **`System.getenv("CLASSPATH")`**
   - **`System.getevn("ComSpec")`**
   - **`System.getenv()`**

   1.2. Execute the following statements

```
double s = 0.0;
for (int i = 0; i < 10; i++) {
    s += Math.random();
}
System.out.println("s = " + s);
```

2. Debugger

The Eclipse's Java Development Toolkit (JDT) includes a debugger that enables you to detect and diagnose errors in your programs running either locally or remotely. The debugger allows you to control the execution of your program by setting breakpoints, suspending launched programs, stepping through your code, and examining the contents of variables.

The Fundamentals of Debugging

When you debug Java, your basic activities are

- Stepping through the execution of your program with actions in the Debug view
- Following the source in the editor as it executes
- Managing your breakpoints from the editor and the Breakpoints view
- Examining variable values in the Variables view
- Evaluating expressions and viewing the results in the Expressions and Display views
- Following the output of your program in the Console view

Getting started with debugging is simple. You set a breakpoint (or breakpoints) in your code, start a debugging session, control execution of your code, and examine the state of your program as it runs.

Breakpoints

A breakpoint suspends the execution of a program at the location where the breakpoint is set.

Breakpoints can be enabled and disabled via the context menu in the Breakpoints View, or via the context menu in the Java Editor ruler.

- An enabled breakpoint causes a thread to suspend whenever the breakpoint is encountered. Enabled breakpoints are drawn with a blue circle [ ] and have a checkmark overlay once successfully installed. A breakpoint can only be installed when the class the breakpoint is located in has been loaded by the VM.
- A disabled breakpoint will not cause threads to suspend. Disabled breakpoints are drawn with a white circle [ ].

Breakpoints are displayed in the vertical editor ruler and in the Breakpoints view.

The simplest way to set a breakpoint is to double-click in the editor's marker bar on the line you want the breakpoint defined. You can also position the insertion cursor in a line and then press **Ctrl+Shift+B**

Adding Line Breakpoints

Line breakpoints are set on an executable line of a program.

1. In the editor area, open the file where you want to add the breakpoint.
2. Directly to the left of the line where you want to add the breakpoint, open the marker bar (vertical ruler) pop-up menu (**right-click**) and select **Toggle Breakpoint.** You can also double-click on the marker bar next to the source code line. A new breakpoint marker appears on the marker bar, directly to the left of the line where you added the breakpoint.

Removing Line Breakpoints

Breakpoints can be easily removed when you no longer need them.

1. In the editor area, open the file where you want to remove the breakpoint.
2. Directly to the left of the line where you want to remove the breakpoint, open the marker bar pop-up menu and select **Toggle Breakpoint**. The breakpoint is removed from the workbench. You can also double-click directly on the breakpoint icon to remove it.

Setting Method Breakpoints

Method breakpoints are used when working with types that have no source code (binary types).

1. Open the class in the **Outline View**, and select the method where you want to add a method breakpoint.
2. From the method's pop-up menu, select **Toggle Method Breakpoint**.
3. A breakpoint appears in the **Breakpoints View**. If source exists for the class, then a breakpoint also appears in the marker bar in the file's editor for the method that was selected.
4. While the breakpoint is enabled, thread execution suspends when the method is entered, before any line in the method is executed.

Method breakpoints can also be setup to break on method exit. In the Breakpoints view, select the breakpoint and toggle the **Exit** item in its context menu.

Method breakpoints can be removed, enabled, and disabled just like line breakpoints.

Applying Hit Counts

A hit count can be applied to line breakpoints, exception breakpoints, watchpoints and method breakpoints. When a hit count is applied to a breakpoint, the breakpoint suspends execution of a thread the n^{th} time it is hit, but never again, until it is re-enabled or the hit count is changed or disabled.

To set a hit count on a breakpoint:

1. Select the breakpoint to which a hit count is to be added.
2. From the breakpoint's pop-up menu, select **Hit Count**.
3. In the **Enter the new hit count for the breakpoint** field, type the number of times you want to hit the breakpoint before suspending execution.

   Note: When the breakpoint is hit for the n^{th} time, the thread that hit the breakpoint suspends. The breakpoint is disabled until either it is re-enabled or its hit count is changed.

Managing conditional breakpoints

An enabling condition can be applied to a line breakpoint such that the breakpoint suspends execution of a thread in one of these cases:

1. when the enabling condition is **true**
2. when the enabling condition changes

To set a condition on a breakpoint:

1. Find the breakpoint to which an enabling condition is to be applied (in the **Breakpoints View** or in the editor marker bar).
2. From the breakpoint's pop-up menu, select **Breakpoint Properties...**. The Breakpoint properties dialog will open.
3. In the properties dialog, check the **Enable Condition** checkbox.

4. In the **Condition** field enter the expression for the breakpoint condition.
5. Do one of the following:
6. If you want the breakpoint to stop every time the condition evaluates to **`true`**, select the **condition is 'true'** option. The expression provided must be a **`boolean`** expression.
7. If you want the breakpoint to stop only when the result of the condition changes, select the **value of condition changes** option.
8. Select **OK** to close the dialog and commit the changes. While the breakpoint is enabled, thread execution suspends before that line of code is executed if the breakpoint condition evaluates to **`true`**.

A conditional breakpoint has a question mark overlay on the breakpoint icon.

Starting a Debugging Session

To start a debugging session, select a Java program or a Java element containing the main method you want to debug, and select **Debug > Debug As > Java Application** from the menu or the toolbar. JDT executes the Java program, suspends execution prior to the line with a breakpoint you defined, and opens the Debug perspective. If you do not start execution with one of the Debug actions, execution will not suspend on breakpoints you have defined.

Controlling Program Execution

You control program execution from the Debug view with the following actions.

 **Step Over or F6** executes a statement and suspends execution on the statement after that.

 **Step Into or F5**, for method invocations, creates a new stack frame, invokes the method in the statement, and suspends execution on the first statement in the method. For other statements, like assignments and conditions, the effect is the same as **Step Over**.

 **Step Return or F7** resumes execution to the end of the current method and suspends execution on the statement after the method invocation, or until another breakpoint is encountered.

 **Resume** or **F8** causes execution to continue until the program ends or another breakpoint is encountered.

 **Terminate** stops the current execution without executing any more statements.

 **Run to Line** or **Ctrl+R** resumes execution until a specified line is executed.

Examining an Executing Program

When your program's execution is suspended in the Debugger, you can examine its execution state in the following ways.

- Manipulating variable values—examine variable values in the Variables view, and change a value by double-clicking it.
- Viewing field values—see the value of a field by hovering over it in the editor.

- Evaluating expressions—select and expression in the editor or the Display view, and then select Display or Inspect. The results are shown in the Display view and Expressions view, respectively.
- Viewing method invocations—see the series of nested method invocations that led to the current stack frame by selecting previous stack frames in the Debug view.
- Viewing program output—see program output in the Console view as you step through your program's execution.

Let start with debugging

Your turn

Follow the steps below to practice with the Eclipse's debugger.

1. Create new Java project called "**`lab3`**".
2. Import "**`debug.jar`**" into your project. If you don't know or remember how to do it, please refer to Lab 1. After import, there might be some errors because all files have been declared to be in **`debugging`** package (the concept of package will be introduce later). **Double-click** on file **`DebuggingExample.java`** in Package Explore View. You will notice that the first line has an error. Click anywhere within the first line and press **Ctrl+1** (quick fix). A list of possible ways to resolve the error is show in Figure 10. Select **Move 'DebuggingExample.java' to package 'debugging'** option by **double-click** on it.

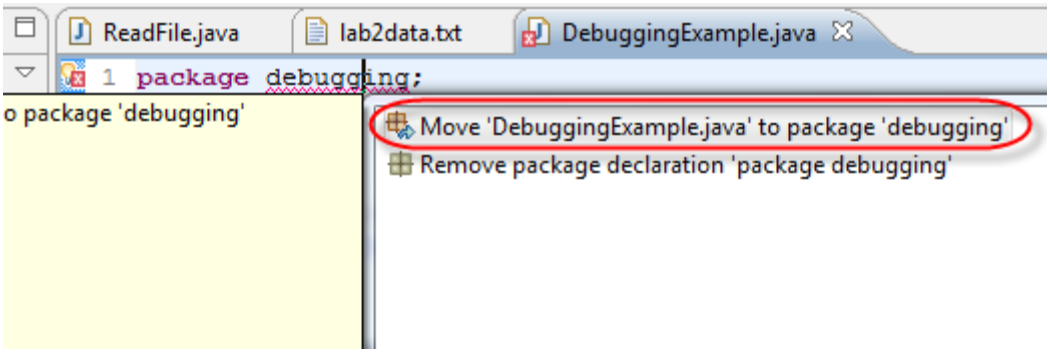


Figure 10 - Quick fix

3. A new package called debugging is created as shown in Figure 11. Click on **`NumberGenerator.java`** and drag it to package debugging. Repeat the same procedure for **`PrimeFactorialGenerator.java`** and **`PrimeNumberGenerator.java`**.

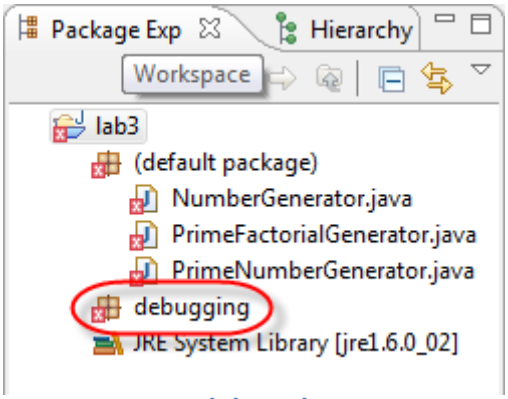


Figure 11 - Package **debugging** is created.

4. Edit the class **`DebuggingExample.java`** in the package **`debugging`** and set breakpoint on the first statement in the **`main`** method by **double-clicking** on the marker bar of the editor next to the line (See Figure 12)

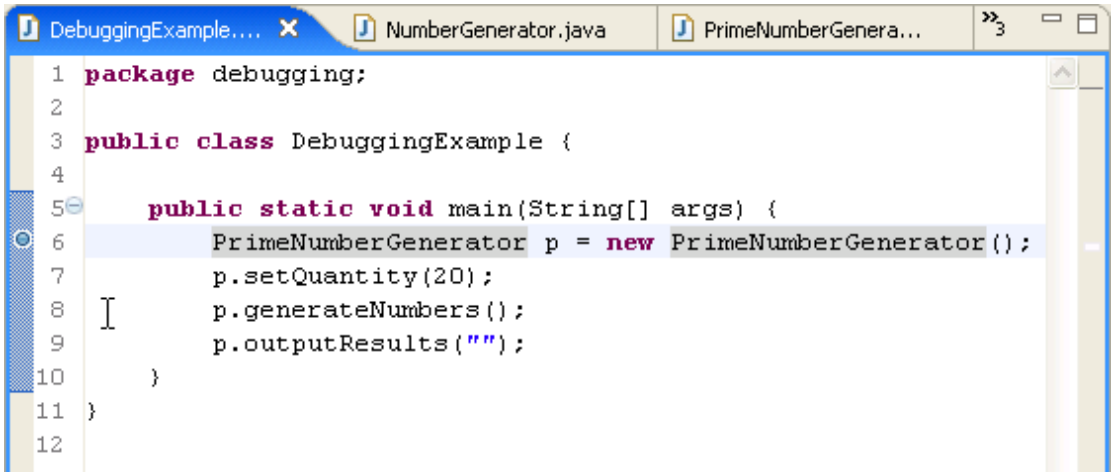


Figure 12 Setting a Breakpoint

5. From the Run pull-down menu select **Run > Debug As > Java Application**. The main method in the class **`DebuggingExample`** executes, the Debug perspective opens, and execution suspends before the line on which you defined the breakpoint, as shown in Figure 13.

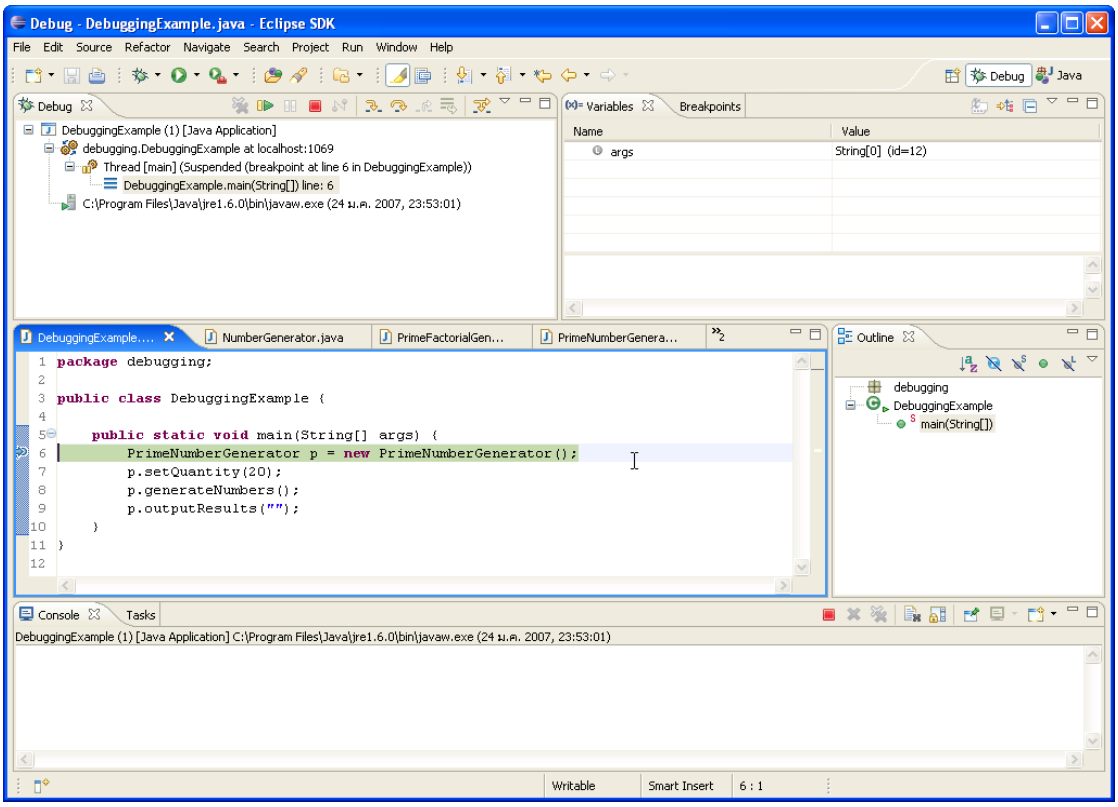


Figure 13 Debug Perspective

6. In the debug view, select Debug Target **`debugging.DebuggingExample`** (see Figure 14) and then select **Properties…** from the context menu (**right-click**). You see information on how the debug session was started, including the input parameters you defined in the launch configuration (see Figure 15). Select **OK**.

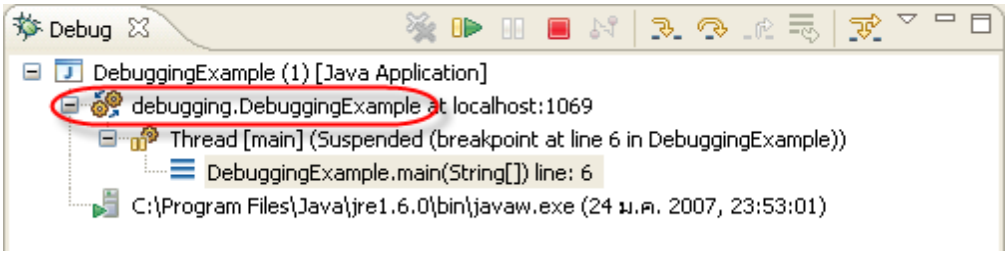


Figure 14 Debug View

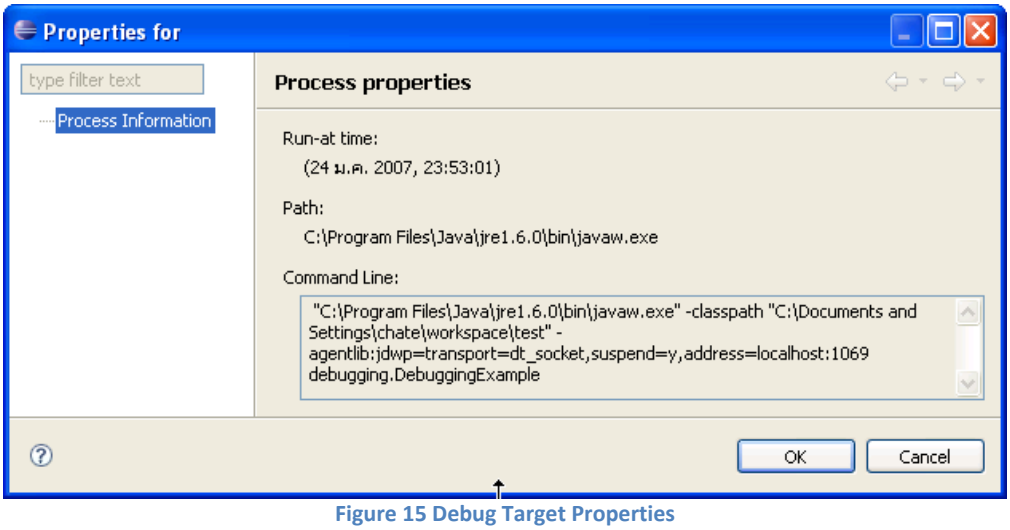


Figure 15 Debug Target Properties

7. In the next several steps you'll use debugging commands to control program execution.

   Select the current stack frame in the Debug view [ ], and then select **Step Over** or **F6** [ ]. One line executes and execution suspends on the next line. The variable **p** appears in the Variables view (see Figure 16). Write down the value of the variable **p** in the lab answer sheet.

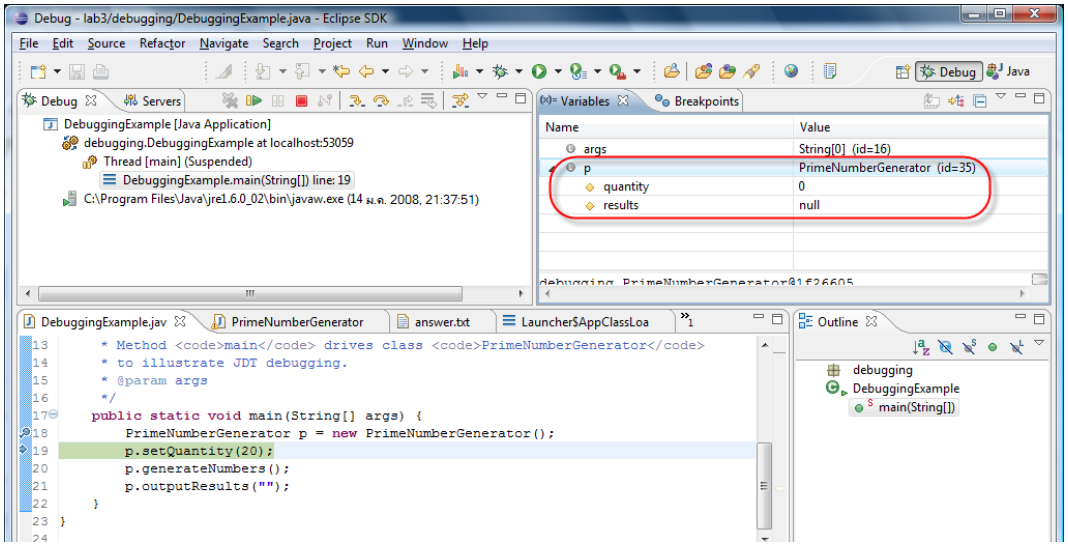


Figure 16 Step Over

8. Select **Step Into** [ ] or press **F5**. A new stack frame is created for the method invocation **`setQuantity()`**, the editor opens on **`NumberGenerator.java`**, and execution suspends on the first statement in the method (see Figure 17).

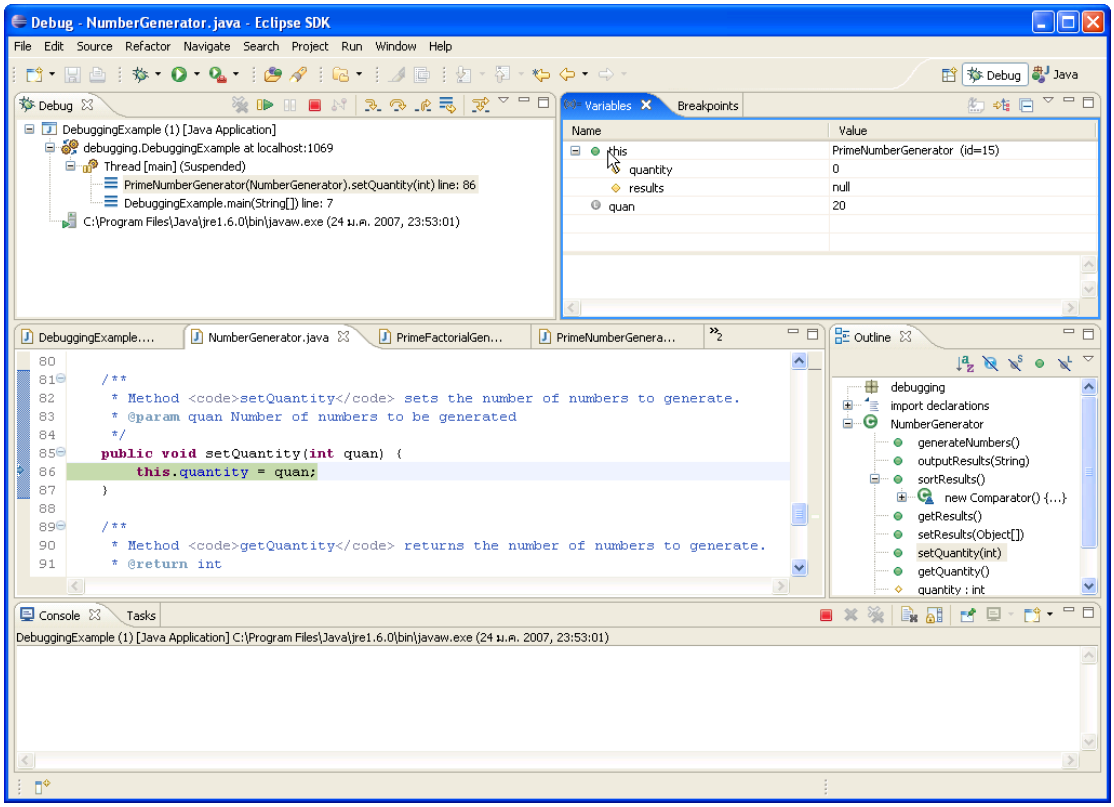


Figure 17 Step Into

9. Step Over this line and the next one to exit the method. The top stack frame is discarded and execution suspends on the statement following the one you just stepped into. Step into **`p.generateNumbers().`** Select **Step Return** [ ] or press **F7**. Execution resumes to the end of the method in the current stack frame, returns from the method execution, and suspends on **`p.outputResults("")`**, the statement following the method invocation.

10. Step Into **`p.outputResult(),`** Set a breakpoint on the line with the **`for`** statement and select **Step Return**. Execution suspends at the breakpoint rather that after the methods returns, because the breakpoint is encountered first.

11. Step over the **`for`** statement. Hover the cursor over the variable **`i`**. The value of the variables is displayed as shown in Figure 18.

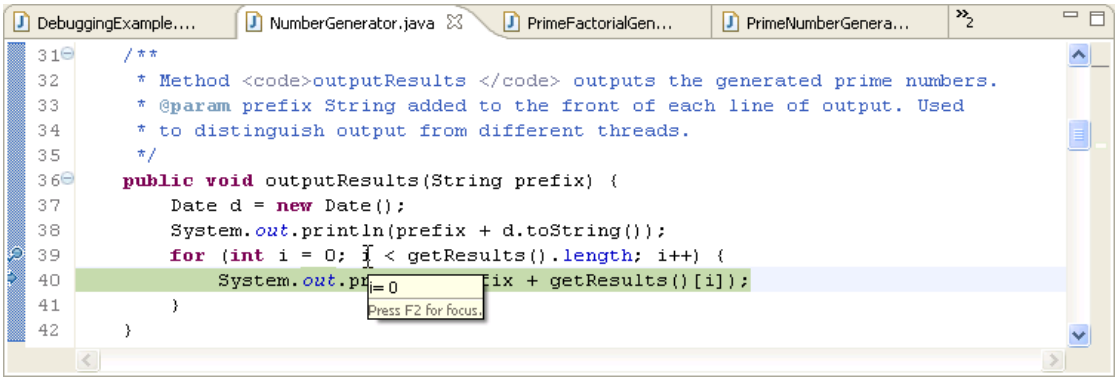


Figure 18 Hovering to View a Variable Value

12. Select **Run to Line** from the context menu or press **Ctrl+R**. Execution resumes and then suspends on the line that was selected, after an iteration loop. You did not have to define a breakpoint. You can verify this by the increase in value of variable **`i`**.

13. Select **Step Return** to complete the method **`outputResults`**. The output appears in the Console view. Select **Terminate** to stop execution or select **Resume** to continue execution to the end of the program. The status of your program in the Debug view shows it has terminated (see Figure 19). Remove the terminated entries in the Debug view with **Remove All Terminated** [ ].

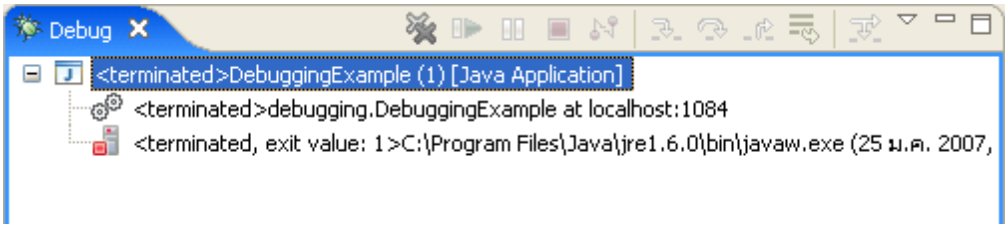


Figure 19 Terminated Debug Sessions

14. Now You'll see how to view and change variable values. Select **Run** or press **F11** to restart a debugging session on **`DebuggingExample`**. Step Over the first line and then Step Into **`p.setQuantity(20).`**
15. Switch to the Variables view and select **Show Detail Pane**. Successively select the variables and watch as the values display in the Detail pane in the bottom of the view. These are the values of the variables' **`toString()`** methods. You can provide useful debug information overriding this method in your own classes to display your

objects' state. Select the variable **quan**, select **Change Value** from the context menu, and enter a new value (see Figure 20). You can also double-click on a variable to change it. You cannot change a variable's value from the Detail pane.

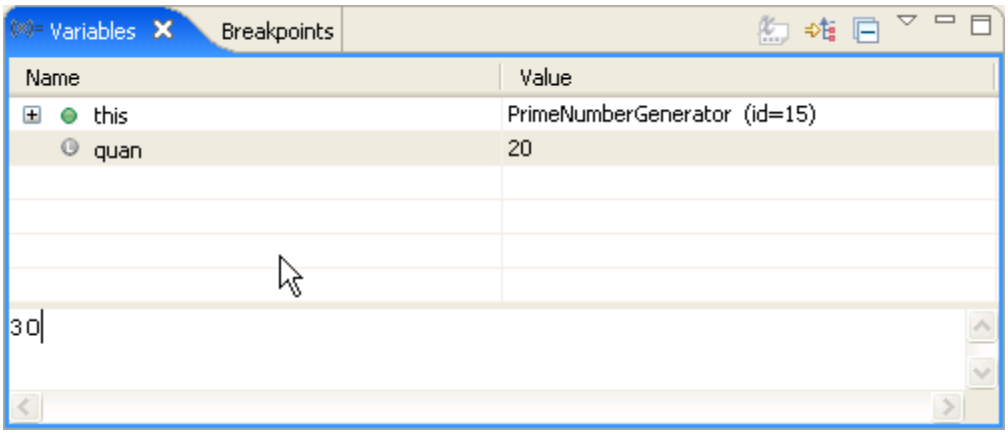


Figure 20 Changing a Variable's Value

16. Select Step Return, then Step Into on the line **`p.generateNumbers()`**. In the variables view, expand this and verify that the field quantity has the changed value.
17. With the variables view visible, Step Over lines to continue through iteration on the while loop. The colors of the entries in the Variables view change as the values change.
18. Breakpoints can have hit counts. To see this, set a breakpoint on the second line of the outer while loop, **`prime = true;`**. From the context menu on the breakpoint on the marker bar, select **Breakpoint Properties…** In the Java Line Breakpoint Properties dialog, select **Enable Hit Count** and set Hit Count to **5** (see Figure 21). Select **OK**. This will cause execution to suspend the fifth time the breakpoint is encountered.

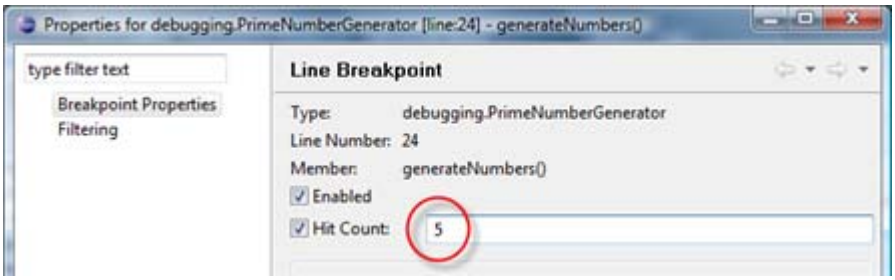


Figure 21 Setting a Breakpoint Hit Count

19. Hover over the breakpoint icon to verify its hit count (see Figure 22). Hover over the variable candidate in the editor and remember its current value.

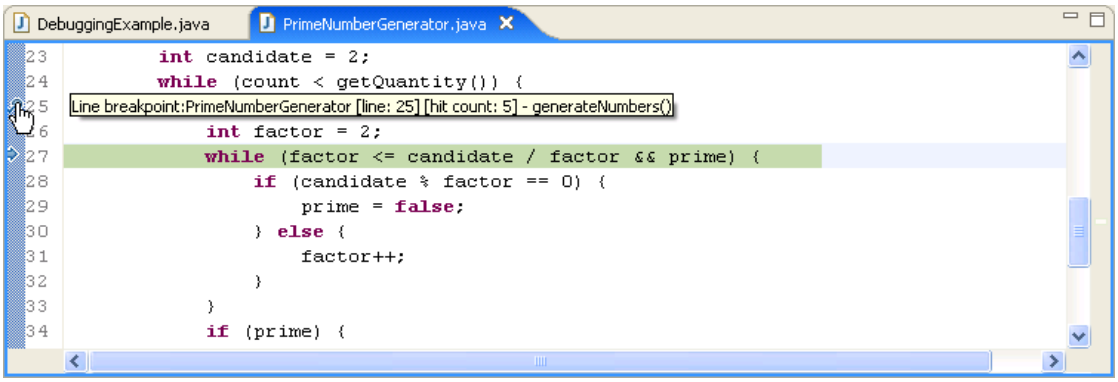


Figure 22 Viewing a Breakpoint Hit Count

20. Select **Resume**. Execution resumes and then suspends on the breakpoint after five iterations of the while loop. Hover again over the variable **`candidate`** to verify this; its value should be incremented by five (or four, depending on where you were in the loop). The breakpoint shows as disabled (a white dot appears in the left margin). Enable it for five more iterations by selecting **Enable Breakpoint** from the context menu. Select **Resume** again. Execution suspends on the same line after another five iterations. Hover over the variable **`candidate`** to verify this (write it down in the answer sheet task 2).

21. Close the editor on the class **`PrimeNumberGenerator`** and go to the Breakpoints view. You can quickly get back to the source where you've set a breakpoint by double-clicking on one in the Breakpoints view. This will open the associated source file and select the line containing the breakpoint. Do this for the disabled breakpoint.

22. Let's change this breakpoint to suspend execution when a condition (Java expression) evaluates to **`true`**. In the Breakpoint view, select the disabled breakpoint and then select **Properties…** from the context menu. In the Java Line Breakpoint Properties dialog, select **Enabled**, deselect **Enable Hit Count**, and select **Enable Condition** to make this a conditional breakpoint (see Figure 23). Enter **`candidate == 40`** for the condition and select **OK**.

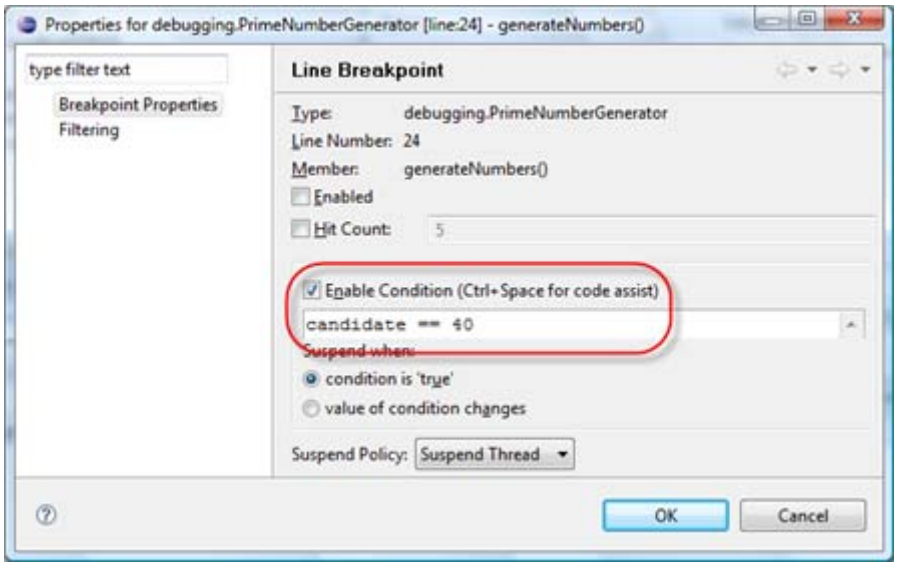


Figure 23 Setting a Breakpoint Condition

In the editor, the question-mark label decoration on the breakpoint indicates it is a conditional one. Hover over the breakpoint icon in the editor to see the condition expression.

23. Select **Resume**. Execution resumes and then suspends. Hover over the variable candidate in the editor to verify that its value is 40.
24. Finally, let's look at evaluating expression. Step Over lines until you are inside the inner while loop on the line, **`if (candidate % factor == 0) {`**. In the editor, select the **`candidate % facter == 0`** expression. From the context menu, select **Display** to evaluate the expression and show the result in the Display view (see Figure 24).

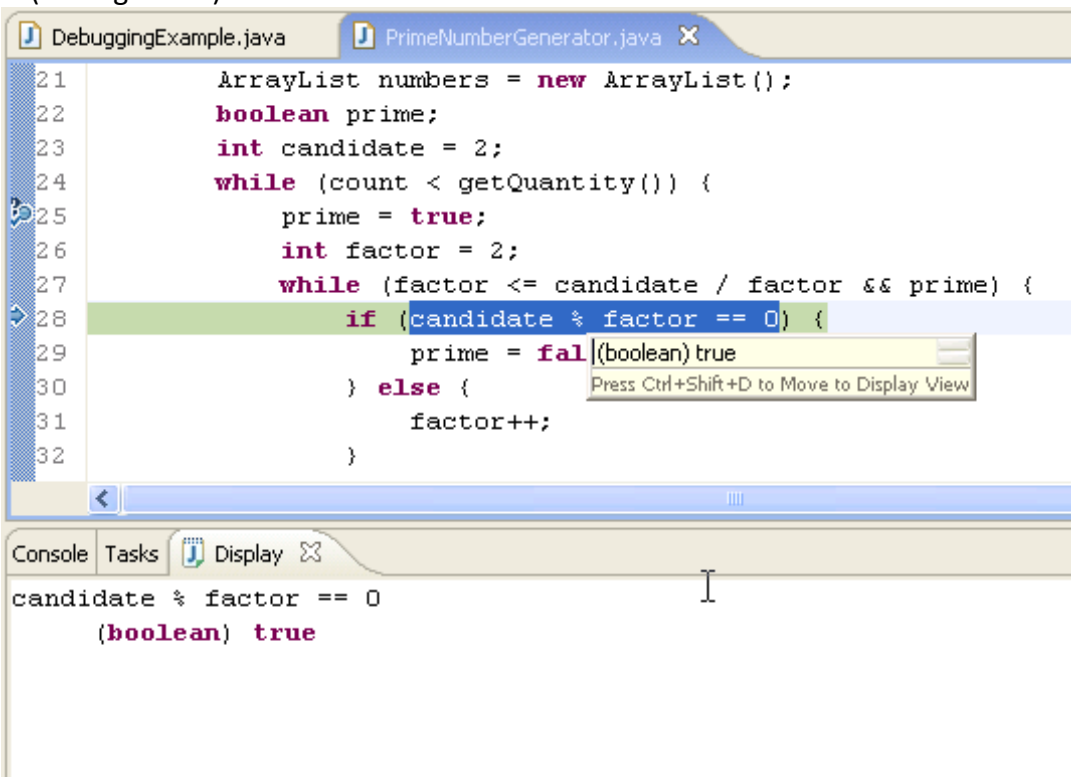
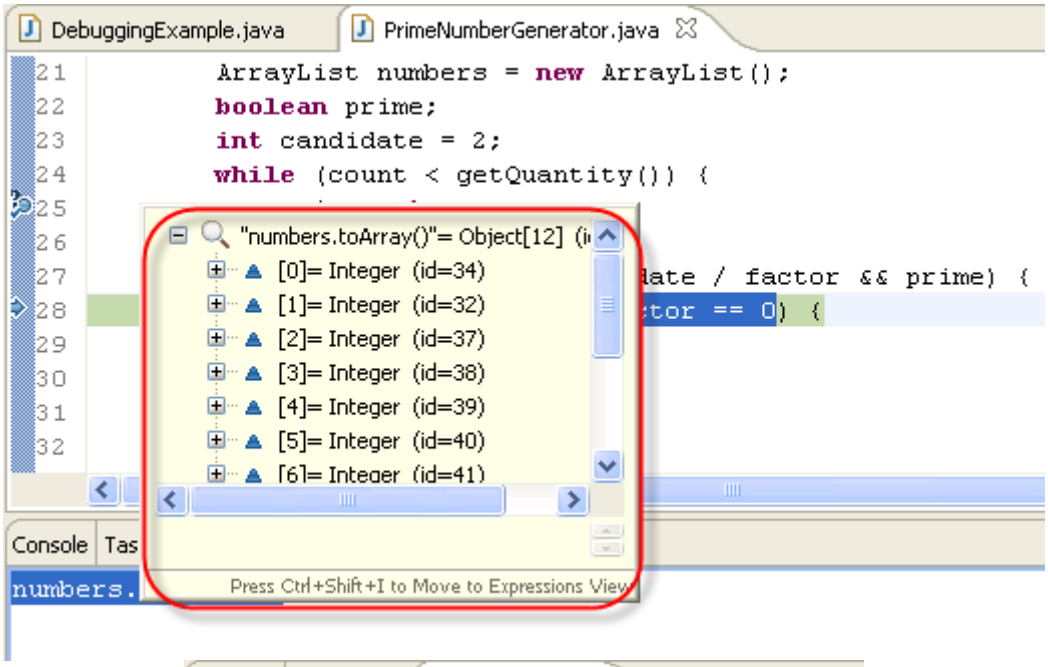


Figure 24 Displaying an Evaluated Expression

25. Enter an expression in the Detail pane of the Display view that can be evaluated in the context of the current stack frame, like **`numbers.toArray().`** Content assist is available here. Select the expression you entered and then select **Inspect Result of Evaluated Selected Text** to display the results in the Expression view. Select to display the Detail pane (See Figure 25).
26. In the Detail pane, enter **`numbers.get(1)`**, select it, and then select **Inspect** from the context menu. Another entry is added to the Expressions view with the result (see Figure 26).
27. Expend the **`numbers.toArray()`** entry in the Expressions view and modify the value of the first entry to a number that is obviously not prime, like 100. The result of evaluating **`numbers.toArray()`** returned an array of Integers, or more precisely an array of references to the Integers of the variable numbers. In the Expressions view, when you change a value to a referenced object, you change the value in the current stack frame. The point here is that with object references, you are not just changing a value in the Expressions view. While you make the change there, you are actually changing the value of an object in your executing program and altering its behavior.



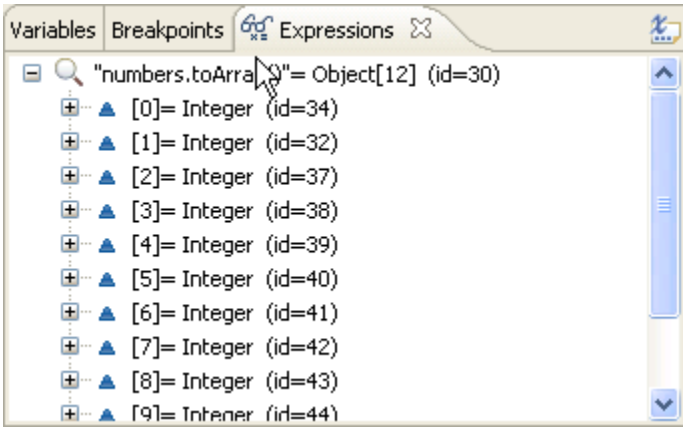


Figure 25 Inspecting an Evaluated Expression

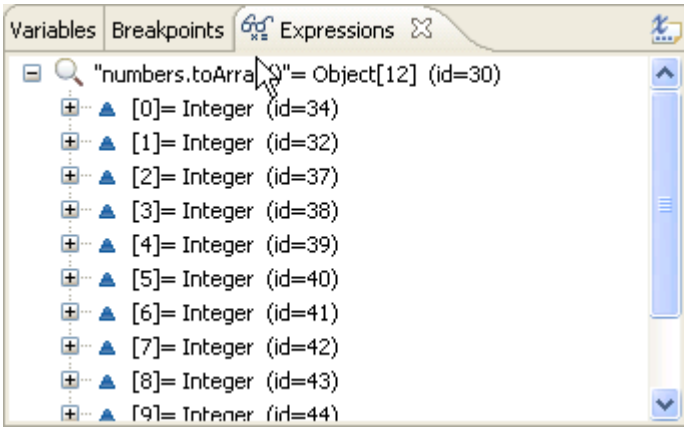


Figure 26 Evaluating an Expression in the Detail Pane

28. Select **Resume** to continue execution to the end of the program. Verify that your changed value appears in the Console view.
    When you're through, don't forget to close all open editors, terminate and remove existing debugging sessions, and remove your existing breakpoints.

Lab 3 Exercise

Your turn

1. Without changing anything in the source file, use debug to find the 500th prime number.
2. Find the smallest prime number which is greater than 10000.
3. How many prime numbers smaller than the answer from (2)?
4. Finish the rest of the remaining exercises you have not finished from Lab 1 and Lab 2.
5. (Optional) A *palindrome* is a word, phrase, number or other sequence of units (such as a strand of DNA) that has the property of reading the same in either direction where the punctuations and spaces are generally ignored. For example, "civic", "level", "Was it a cat I saw?", and "A man, a plan, a canal: Panama" are palindromes.

   Write a recursive Java method that returns **`true`** if the **`String`** passed to the method is a palindrome. Otherwise, it returns **`false.`**

Lab

Task	Description	Result	Note

1	`System.getenv("CLASSPATH")`
	`System.getenv("ComSpec")`
	`System.getenv()`		

2	Use Scrapbook to execute statements		
3	The value of variable **p**		
4	The value of variable **`candidate`** after five iterations twice.		
5	The 500th prime number.		
6	The smallest prime number which is greater than 10000.		
7	The number of prime smaller than the answer in task 4.		