

Exercise 4. Debugging Java code

What this exercise is about

In this exercise, you learn how to use the basic features of the Eclipse or Rational Application Developer debugger. You learn how to set breakpoints, view and modify variables, and step through code.

What you should be able to do

At the end of this exercise, you should be able to:

- Run an application in debug mode
- Set breakpoints to view and modify variables and expressions
- Step through the code using the debugger to quickly track down problems and bugs

Introduction

In the course of developing an application, debugging is an activity that is performed numerous times. The tool provides a debugger that helps you rapidly detect and diagnose errors in your program. Based on a client/server design, it can be used to debug programs running on a remote machine or locally on your workstation. In this exercise, you perform local debugging. All of the functions available for local debugging also work for remote debugging.

You create a program that echoes to the console. To exit, the user is expected to type the word `Quit`. However, the program does not work properly, and once started, the user is caught in an infinite loop. The debugger is invoked in order to help find the cause of the problem and correct the code.

The program uses `StringBuffers` to hold the user input and to set up the condition for exiting. `StringBuffer` comparison is the problem highlighted in this exercise. Unlike the `String` class, which has its own `equals()` method that compares characters within the strings, `StringBuffer` uses the `equals()` method inherited from `Object`. This method compares object references and returns `false` if two different instances of `StringBuffer` are compared.

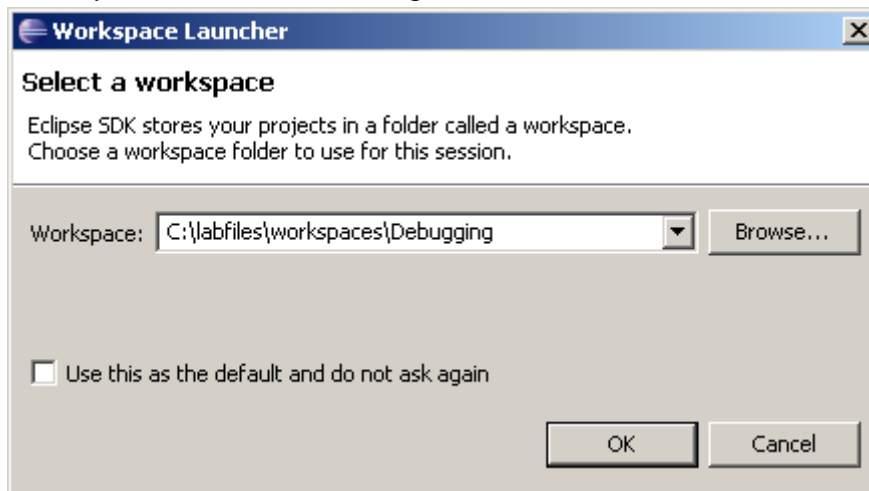
Exercise instructions

Part 1: Opening a new workspace

- ___ 1. Start Eclipse with a new workspace to hold the contents of this lab.
 - ___ a. From the Windows Desktop, double-click the Eclipse shortcut. Alternatively, find the Eclipse directory and double-click `eclipse.exe`. In a typical installation, Eclipse can be found at the following location:

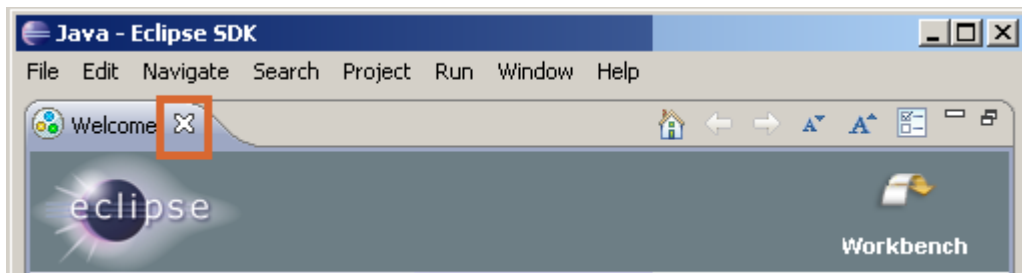
`C:\Program Files\eclipse\eclipse.exe`

- ___ b. When the dialog appears prompting for the location of the workspace that you want to open, enter the following: `C:\labfiles\workspaces\Debugging`



It is often a good practice to keep each of your software projects in its own separate workspace.

- ___ c. Click **OK**.
- ___ 2. When Eclipse opens, click the **X** at the upper left corner of the Welcome page to close it.





Note

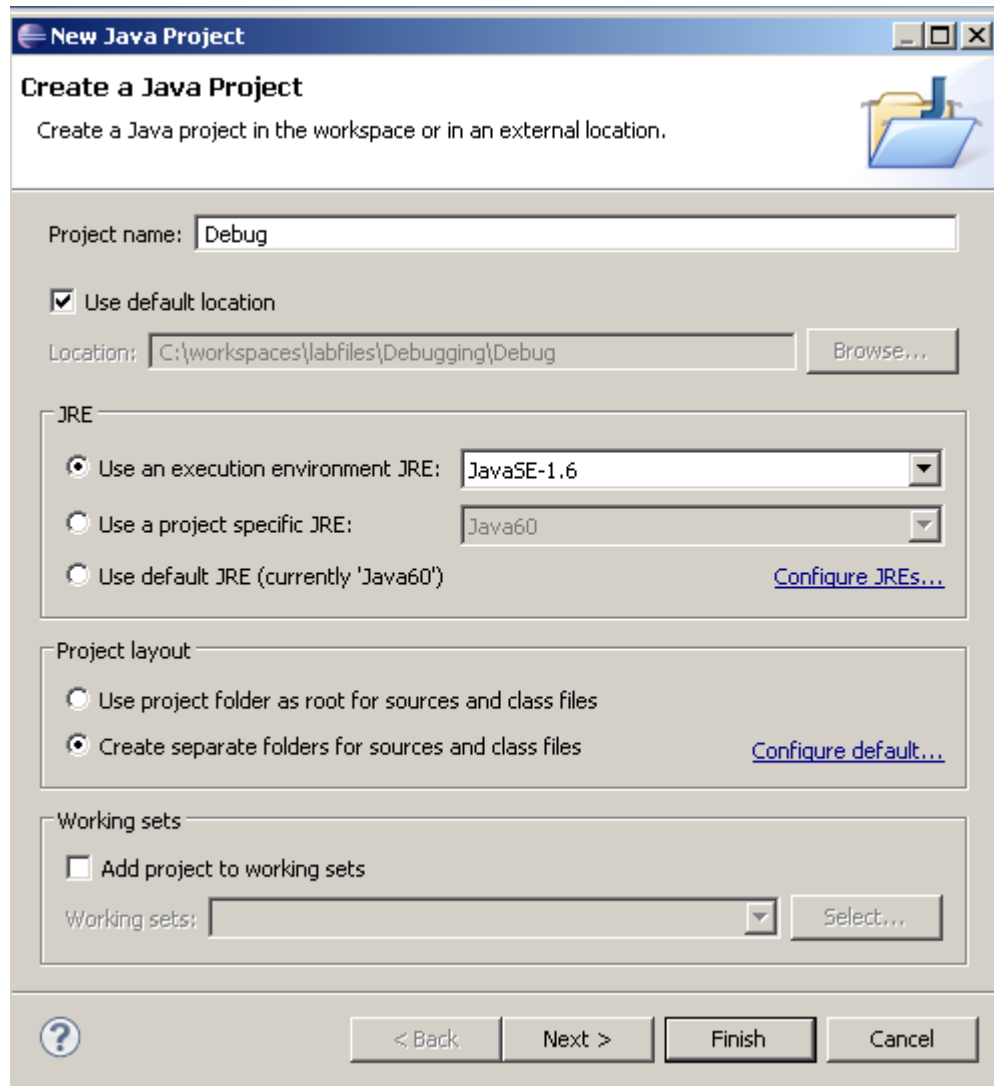
If you already have a workspace open in Eclipse, you can launch the workspace using the menu option **File > Switch Workspace**.

Part 2: Create a Java project, package, and class

You first create a Java project, package, and class for the application that you debug.

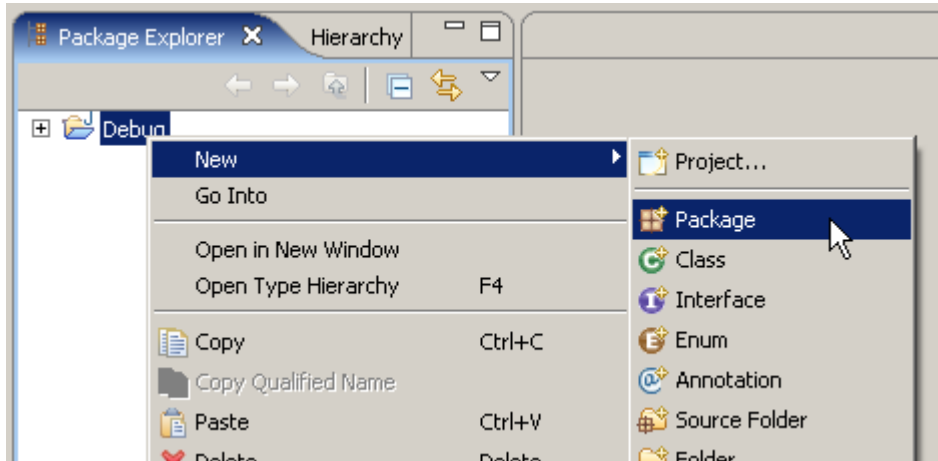
- ___ 1. Create a Java project.
 - ___ a. Switch to a Java perspective by selecting **Window > Open Perspective > Java** from the main menu.
 - ___ b. From the main menu, select **File > New > Java Project**.

- ___ c. In the New Java Project dialog that appears, enter `Debug` for the **Project name** field.

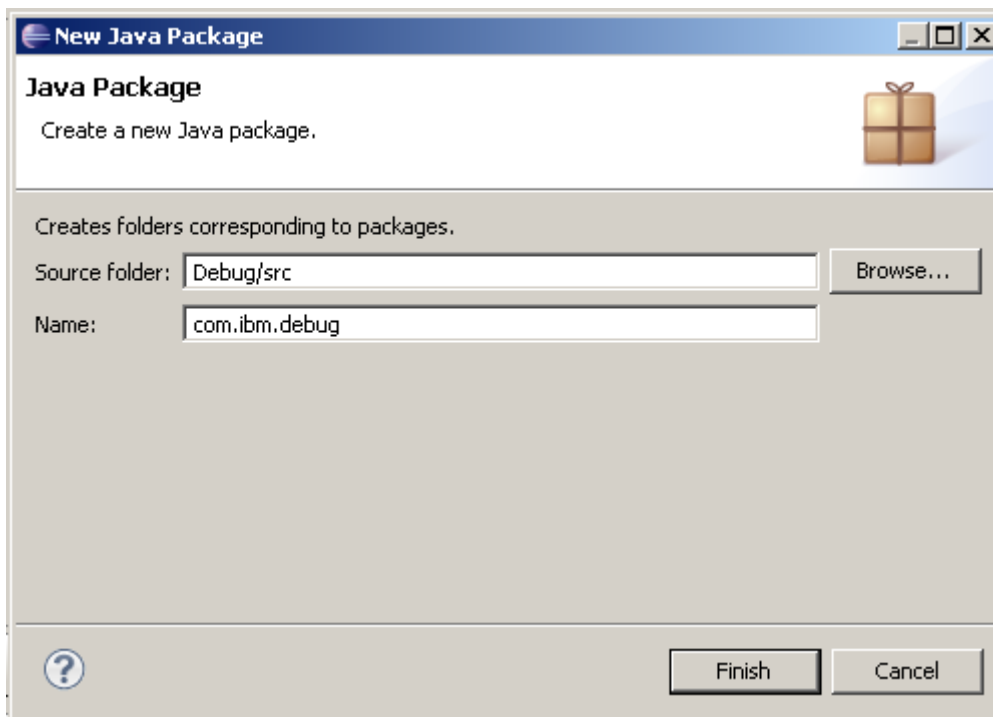


- ___ d. Click **Finish**.
- ___ 2. Create a package inside the project.
- ___ a. In the Package Explorer view of the Java perspective, select the newly created **Debug** Java project.

- ___ b. Invoke the pop-up menu by right-clicking the selection; then select **New > Package**.



- ___ c. The New Java Package dialog appears.
- ___ d. Type `com.ibm.debug` in the **Name** field.



- ___ e. Click **Finish**. A new Java package and dependent JAR files appear in the Package Explorer view.
- ___ 3. Create a class in the package.
- ___ a. In the Package Explorer view, expand the **Debug** project.
- ___ b. Right-click the `com.ibm.debug` package; then select **New > Class**.
- ___ c. In the New Java Class dialog, ensure that the value in the **Source Folder** field is `Debug/src` and the **Package** field is `com.ibm.debug`.

- ___ d. Enter `ExitCondition` in the **Name** field.
- ___ e. Select the `public static void main(String[] args)` check box to generate a main method stub.

New Java Class

Java Class
Create a new Java class.

Source folder:

Package:

☐ Enclosing type:

Name:

Modifiers: ☒ public ☐ default ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclass:

Interfaces:

Which method stubs would you like to create?

☒ `public static void main(String[] args)`
☐ Constructors from superclass
☒ Inherited abstract methods

Do you want to add comments? (Configure templates and default value [here](#))
☐ Generate comments

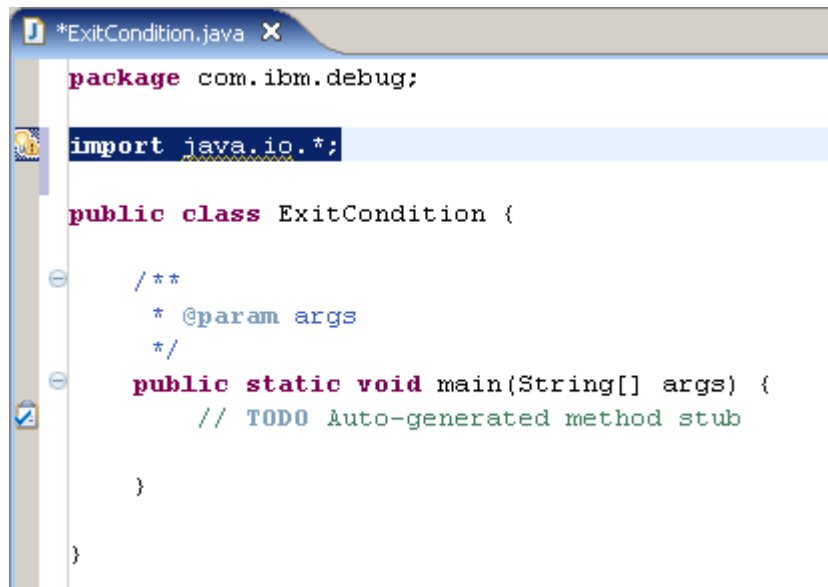
- ___ 4. Click **Finish**.

Part 3: Implementing and testing the `ExitCondition` class

The `ExitCondition` class with a main method stub now appears in a Java editor. You implement this class to respond to user input from the system. You also test the application to ensure that it performs as expected.

- ___ 1. Implement the main method.
 - ___ a. Add the following import statement to the class:

```
import java.io.*;
```



- ___ b. Update the main method according to the code that follows. Remember to update the method signature to throw an IOException exception. You may copy and paste the content from

C:\labfiles\Debugging\snippets\snippet01.txt.

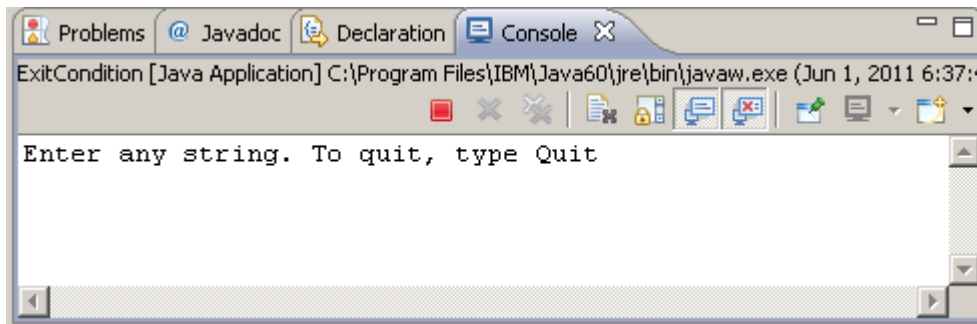
```

public static void main(String[] args) throws IOException {
    BufferedReader br =
        new BufferedReader(new InputStreamReader(System.in));
    StringBuffer s;
    StringBuffer exiting = new StringBuffer("Quit");
    while (true) {
        System.out.println("Enter any string. To quit, type Quit");
        s = new StringBuffer(br.readLine());
        if (s.equals(exiting)) {
            System.out.println("Goodbye");
            System.exit(0);
        } else {
            System.out.println("You typed \"" + s + "\"!");
        }
    }
}

```

- ___ c. Save ExitCondition.java (Ctrl+S). If any errors appear in the Problems view, fix them before continuing.
- ___ d. Run the class, then immediately click the **Terminate** icon on the Console view toolbar.
- ___ 3. Run the ExitCondition program.
- ___ a. Click **Run** to execute the application.
- ___ b. If the Console view is not open, you can open it by selecting **Window > Show View > Console** from the main menu.

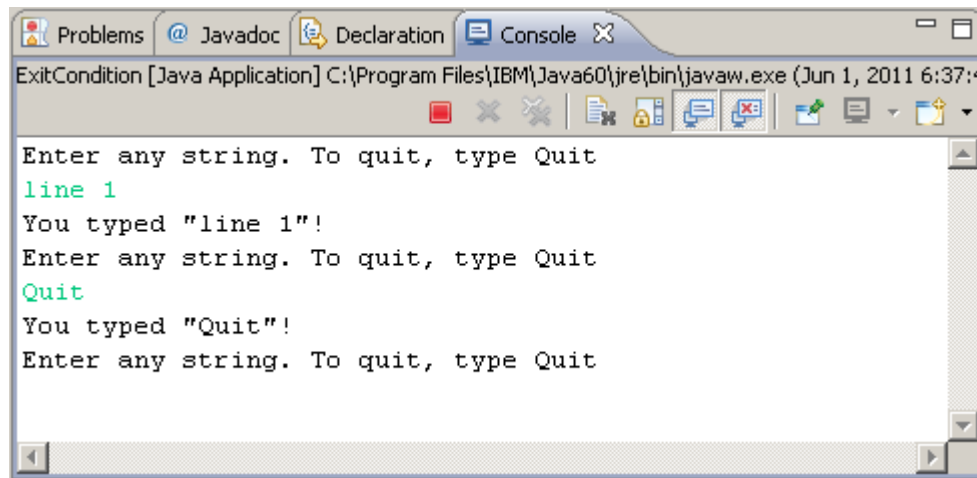
- ___ c. The Console view now displays a prompt from the ExitCondition application.



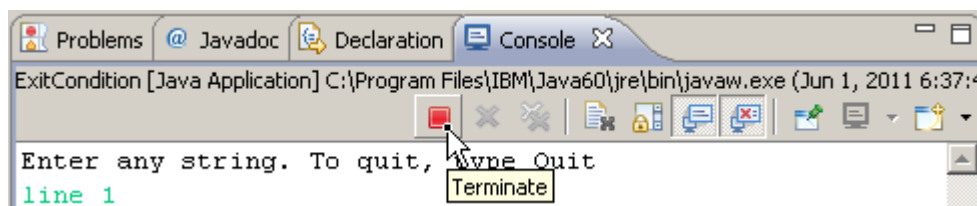
Ensure that the Console view has focus, by clicking its title bar.

Type `line 1` and press Enter. Notice that this application repeats your input in the console.

Type `Quit` and press Enter.



- ___ d. Typing `Quit` into the console did not terminate the application as expected. In the next part, you use the debugging tools to determine why the program did not respond to the word `Quit`.
- ___ 4. You need to force the program to terminate as the `Quit` command does not work. Click the **Terminate** icon on the Console view toolbar.

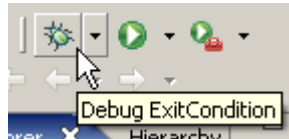


Part 4: Working with the debugger

To discover the cause of the problem, you can read over the code inside the workbench editor. A more efficient way is to step through the logic of the code, examining the

application as it is being executed. The Debug perspective provides various tools to check an application in real time.

- ___ 1. Start the application in Debug mode.
 - ___ a. In the Package Explorer view, expand **Debug > com.ibm.debug**.
 - ___ b. Select `ExitCondition.java`.
 - ___ c. Click the **Debug** icon from the main toolbar. You want to click the icon itself, not the down arrow to the right.



- ___ d. Switch to the Debug perspective by selecting **Window > Perspective > Open Perspective > Debug**.

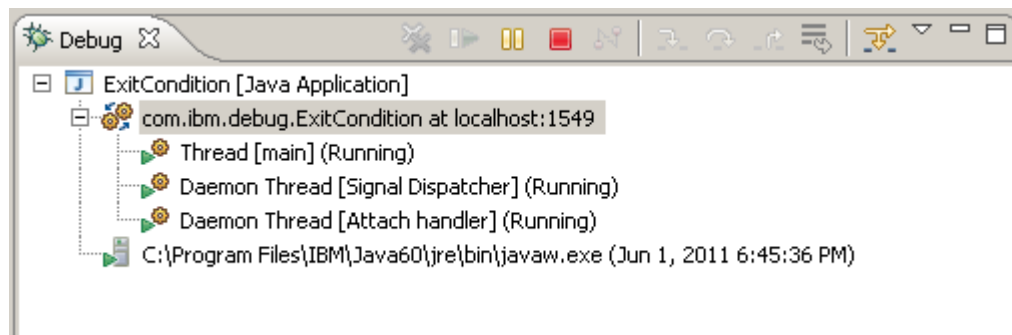


Note

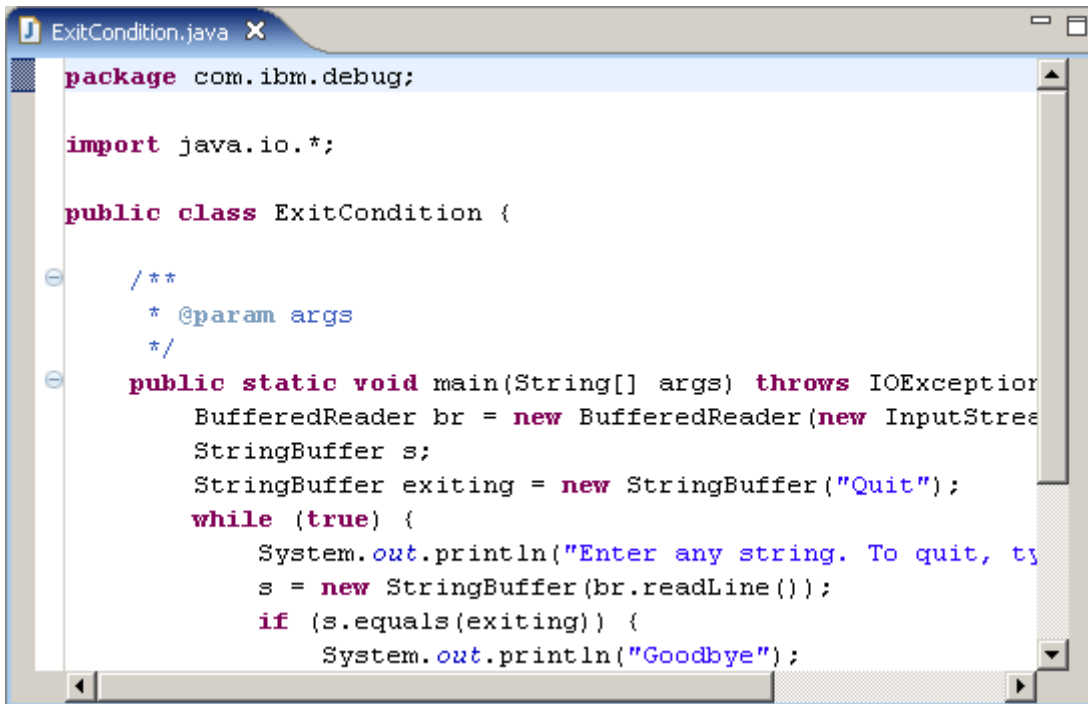
Before you leave the Java perspective, ensure that `ExitCondition.java` is open in the Java editor view.

The **Debug perspective** appears, displaying information about the running application.

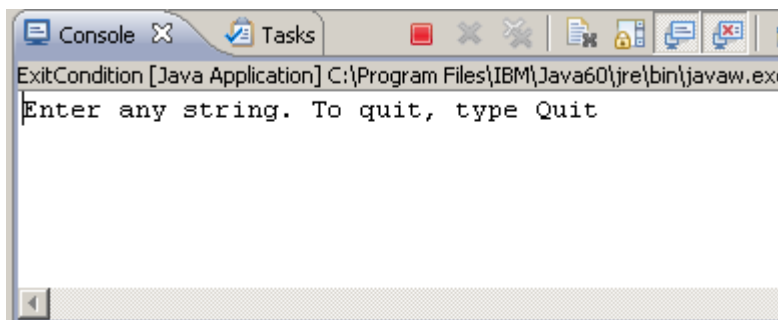
The **Debug view** displays all threads executing in the runtime environment. Currently, only the `ExitCondition` application is running.



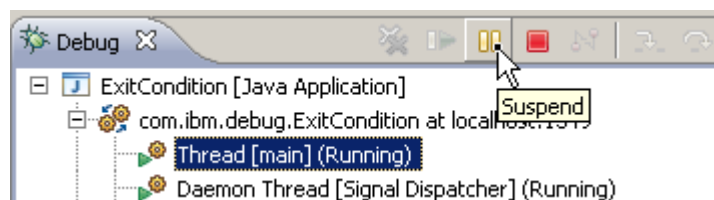
The **Java Editor** displays the code of the currently selected thread. You can set breakpoints and step through code using this editor.



The **Console** view shows any output from the application itself. You can also type in input to the system using this view.



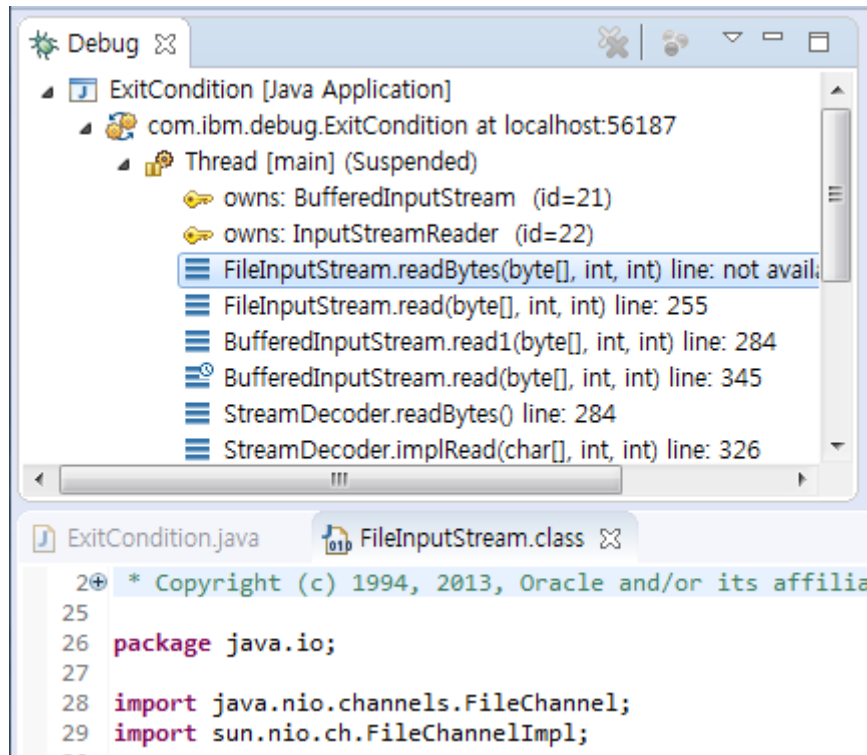
- ___ 2. Pause execution of the ExitCondition application.
 - ___ a. In the Debug view, select the thread labeled Thread [main].
 - ___ b. Click the **Suspend** button from the Debug title bar.



The Debugger suspends the main thread and attempts to open an editor on the class containing the last method executed. In your case, you were waiting on user

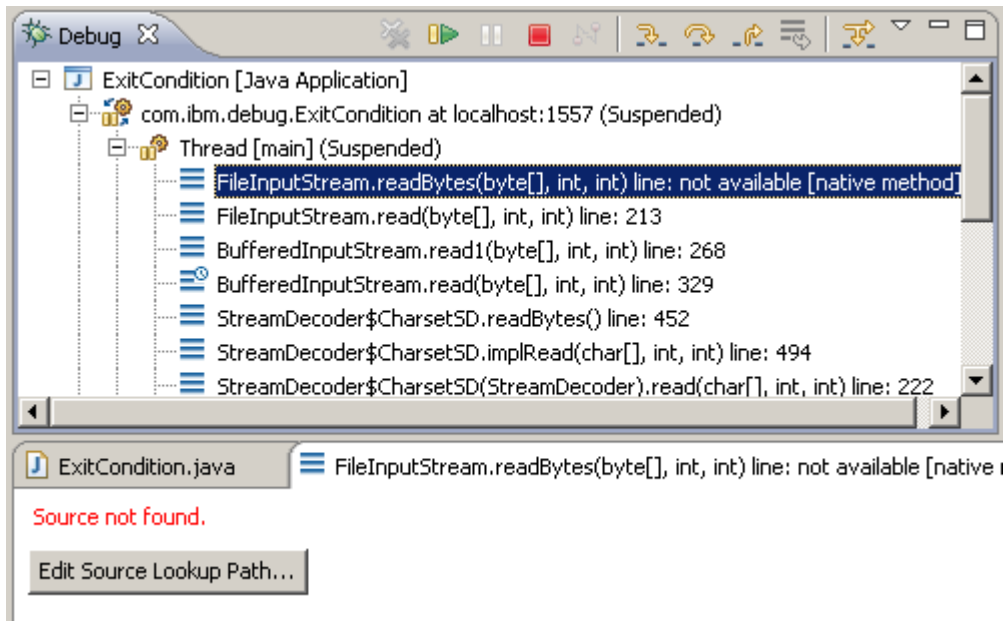
input, and the last method invoked was `readBytes()` in the `FileInputStream` class. Confirm this fact by examining the method stack trace.

- ___ 3. Examine the topmost element in the method stack trace.
 - ___ a. In the Debug view, if the suspended main thread is not already expanded, expand it by clicking the plus sign (+) to the left of `Thread [main]`.
 - ___ b. Select the topmost element immediately following `owns: InputStreanReader` in the method stack trace, `FileInputStream.readBytes(byte[], int, int, FileDescriptor)`. This method is the last method invoked before the thread was suspended.



For Eclipse:

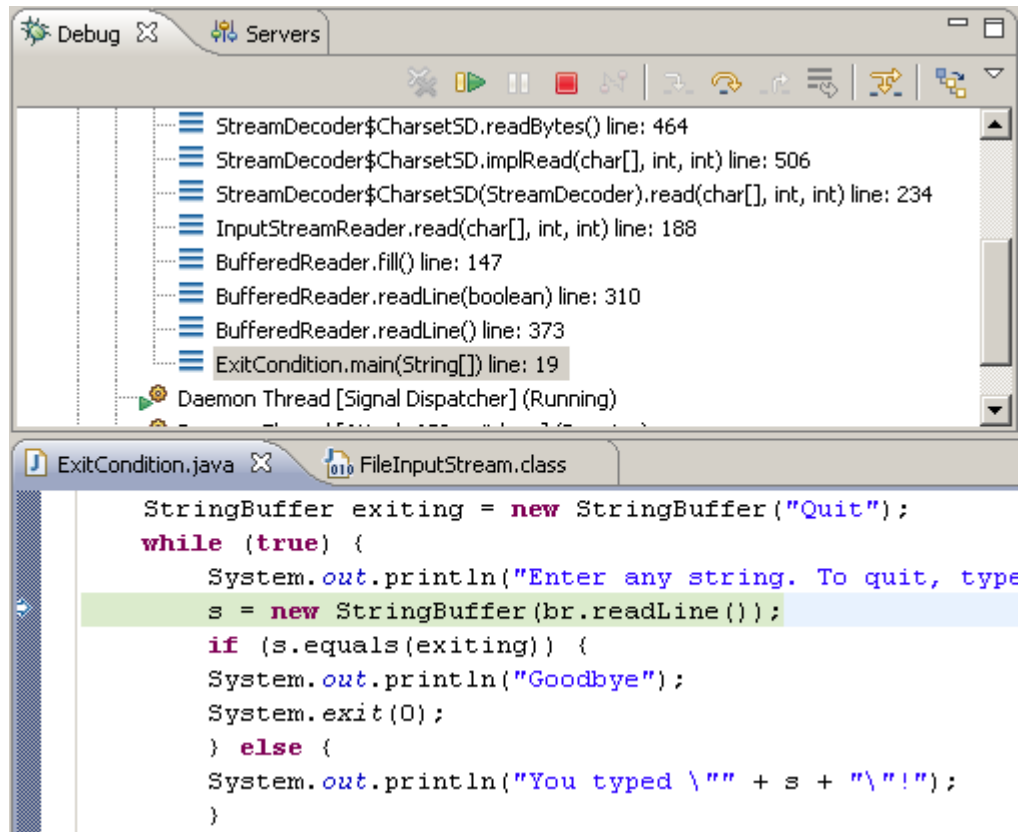
If you were running Eclipse with only the Java 1.8 JRE (not with the Java 1.8 JDK), you would discover that the `FileInputStream` class did not display, and you would see:



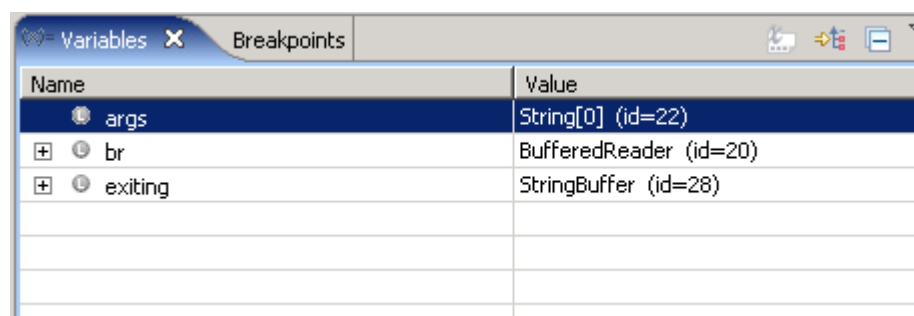
Making sure that you did not get the “Source not found” condition is the reason for changing the `ExitCondition` program earlier, to run with the Java 1.8 JDK, rather than the Java 1.8 JRE.

- ___ 4. Examine the last element in the method stack trace, where the `ExitCondition` class was executing before it was suspended.
 - ___ a. In the Debug view, select the bottom-most element under `Thread [main]` in the method stack trace, `ExitCondition.main(String[])`.
 - ___ b. `ExitCondition.java` is automatically loaded in the editor. The line with the `BufferedReader` object reading a line from Standard Input is marked by an arrow

in the left-edge ruler. This indication is expected, as the application is waiting for input in the Console view.



- ___ 5. Use the Variables view to the right of the Debug view to check all variables in scope.
 - ___ a. Make sure `ExitCondition.main(String[])` is still selected in the Debug view.
 - ___ b. Examine the first variable, `args`. Since no arguments were passed to the Java application, this String array has no elements to expand.

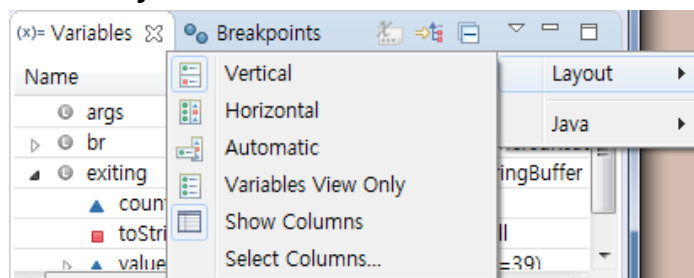


- ___ c. Examine the remaining variables, `br` and `exiting`. For variables other than primitive types and String objects, object fields can be examined by clicking the plus sign (+) beside the variable.
- ___ d. Expand **exiting > value** to examine the contents of the `value` array. It is a field under the `exiting` variable. Fields composed of objects can be further

expanded by selecting the plus sign (+). The `existing` `StringBuffer` object contains the value of `Quit`.

Name	Value
args	String[0] (id=22)
br	BufferedReader (id=20)
exiting	StringBuffer (id=28)
count	4
shared	false
value	char[20] (id=39)
[0]	Q
[1]	u
[2]	i
[3]	t

- ___ 6. Another way to inspect the string representation of an object is to invoke its `toString()` method.
 - ___ a. In the Variables view, select the **value** array.
 - ___ b. Click the **menu** button (downward triangle) on the Variables view toolbar, and make sure the **Layout > Vertical** is selected.



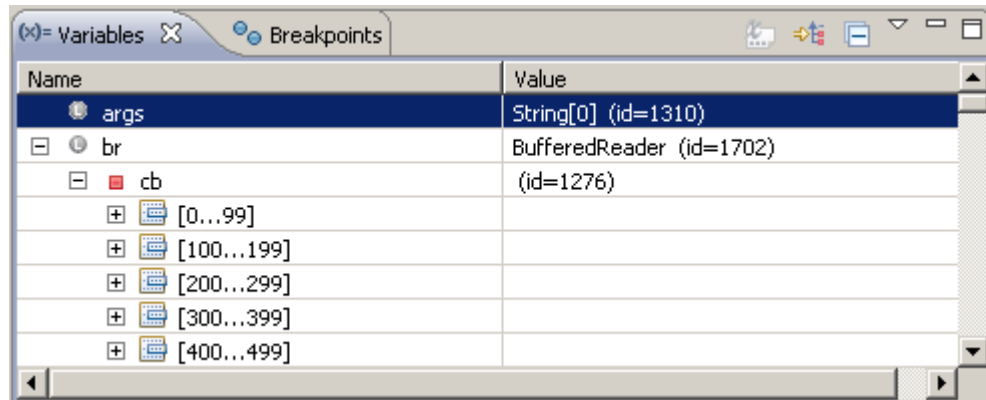
- ___ c. Locate the Detail pane that appears on the lower portion of the Variables view. It displays the result of calling the `toString()` method on the **value** array.

[illegible]

- ___d. Confirm that this result is equivalent to the **value** field.

You can change the location of the Detail pane by selecting **Layout > Vertical View Orientation** or **Horizontal View Orientation** after clicking the **menu** button. You can also toggle it off by selecting **Variables View Only**.

- ___ 7. While stepping through the lines of code in `ExitCondition`, you can examine the contents of the `br` `BufferedReader` object in real time.
 - ___ a. In the Variables view, expand the fields in the `br` `BufferedReader` object by clicking the plus sign (+) box.
 - ___ b. Expand the `cb` field. This array of char data type holds the contents of the `BufferedReader` object.



At the moment, there is no value to display. As you step through the code, expect values to appear.

Part 5: Stepping through the code

Using the four different step modes, and other commands, you can execute Java applications one line of code at a time. Using step modes allows you to examine, adjust variable values, and monitor the program execution path. Stepping through code is an essential practice for checking logic errors in programs.

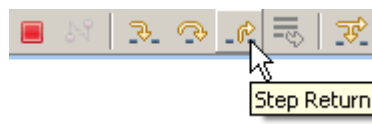
The **Step Into** button evaluates the current line of code, by dropping one level deeper into the call stack. For example, with the code stopped in the `ExitCondition` class at the `BufferedReader readLine()` method, selecting **Step Into** reveals the code being executed in the `BufferedReader` class, if available. A new editor appears with the source of the invoked class and execution is suspended at the first statement. The F5 function key is equivalent to the **Step Into** button.



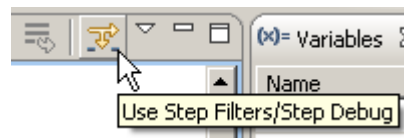
The **Step Over** button executes until the next line of code in the same level of the call stack. Step Over is a useful function if you are not concerned with the methods invoked by the current line of code. The F6 function key is equivalent to the **Step Over** button.



The **Step Return** button continues execution until control is returned to the next level higher in the call stack. For example, while stepping over lines of code inside the `readLine()` method in `BufferedReader`, selecting **Step Return** continues execution until control is returned to the `ExitCondition` class. Issuing a **Step Return** command in the main method causes the program to run until completion. The F7 function key is equivalent to the **Step Return** button.



The **Use Step Filters/Step Debug** button allows stepping through code that contains debug information (that is, the source code) while skipping code that does not. The currently selected line executes, and execution continues until reaching the next line with debug information. Use the button in the Debug view toolbar or press Shift + F5.



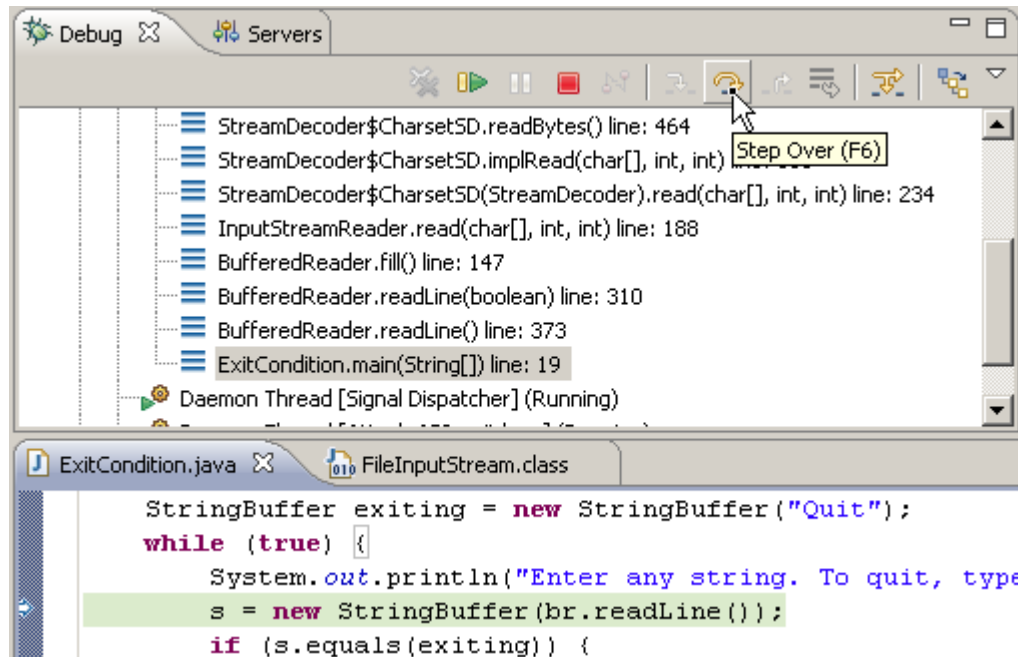
The **Run to Line** command allows execution to continue until reaching the currently highlighted line in the editor. This command is a useful alternative to continuously issuing the Step Over command or inserting a breakpoint to the code. It can be accessed by selecting **Run > Run to Line** from the workbench menu or by using the keyboard equivalent by pressing Ctrl+R.

___ 1. Resume program execution suspended at the line waiting for input from the user.

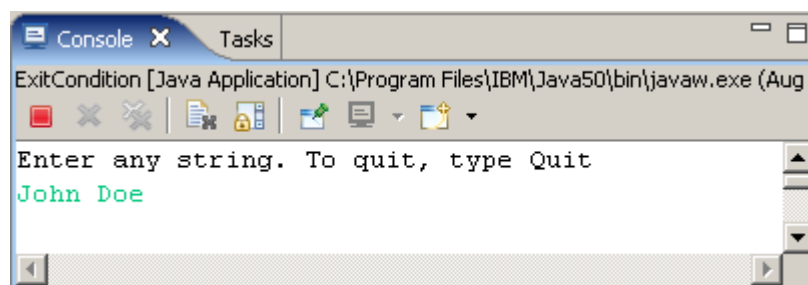
___ a. Confirm that the program is suspended at the following line in `ExitCondition`:

```
s = new StringBuffer(br.readLine());
```


- ___ b. Click the **Step Over** button once. This resumes operation with the application waiting for input in the console.

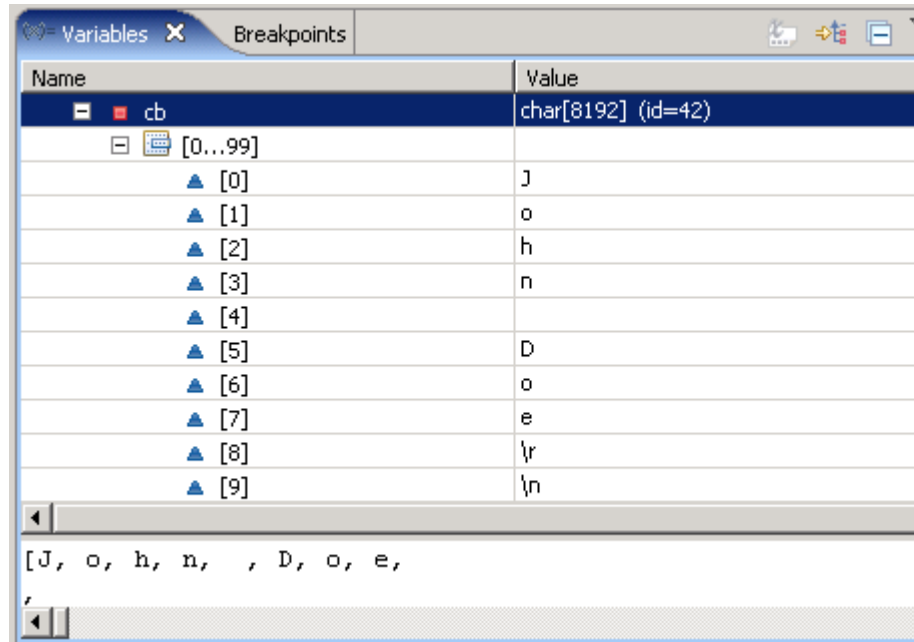


- ___ c. Select the Console view. Click the line under the Enter any string prompt.
- ___ d. Type John Doe into the Console view. Press Enter.



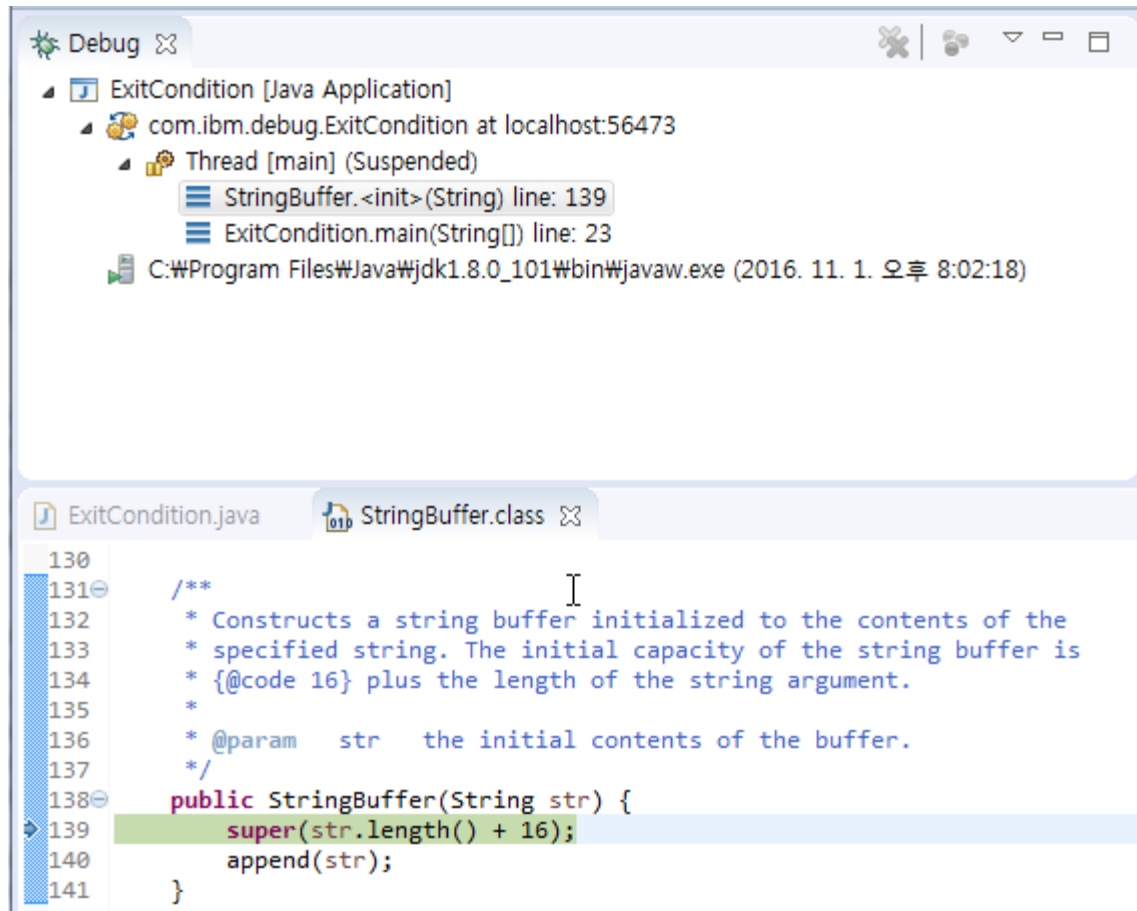
- ___ 2. Examine the variables currently in scope.
- ___ a. In the Variables view, expand the `br` element.

- ___ b. Expand the `cb` field of the char array. Examine the first 100 elements of the array by expanding the `[0..99]` element.

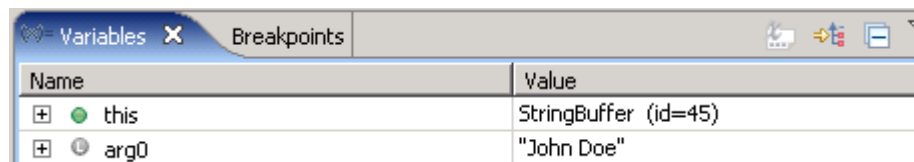


- ___ c. Notice that the `br` `BufferedReader` object contains the text that you have typed into the Console view.
- ___ 3. Examine the code evaluation using the Step Into command.
- ___ a. Click the **Step Into** button or press F5.

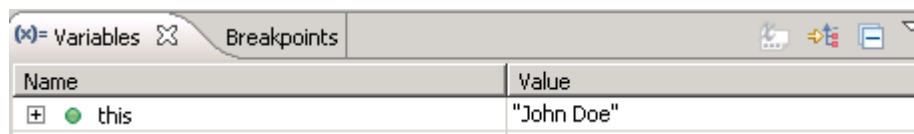
- ___ b. Look at the Thread [main] thread in the Debug view. One more level of execution has been added to the method call stack. You are now examining the StringBuffer class.



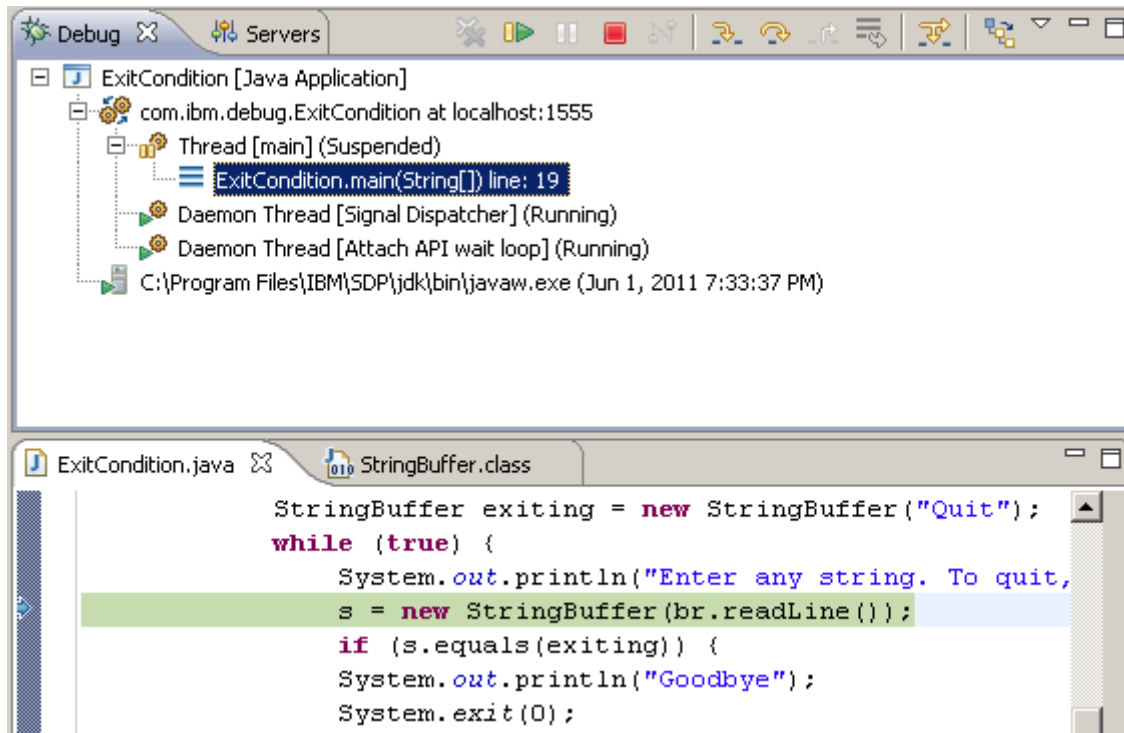
- ___ c. Notice that the Variables view now contains this StringBuffer object. As you move one level lower in the call stack, the variable scope changes.



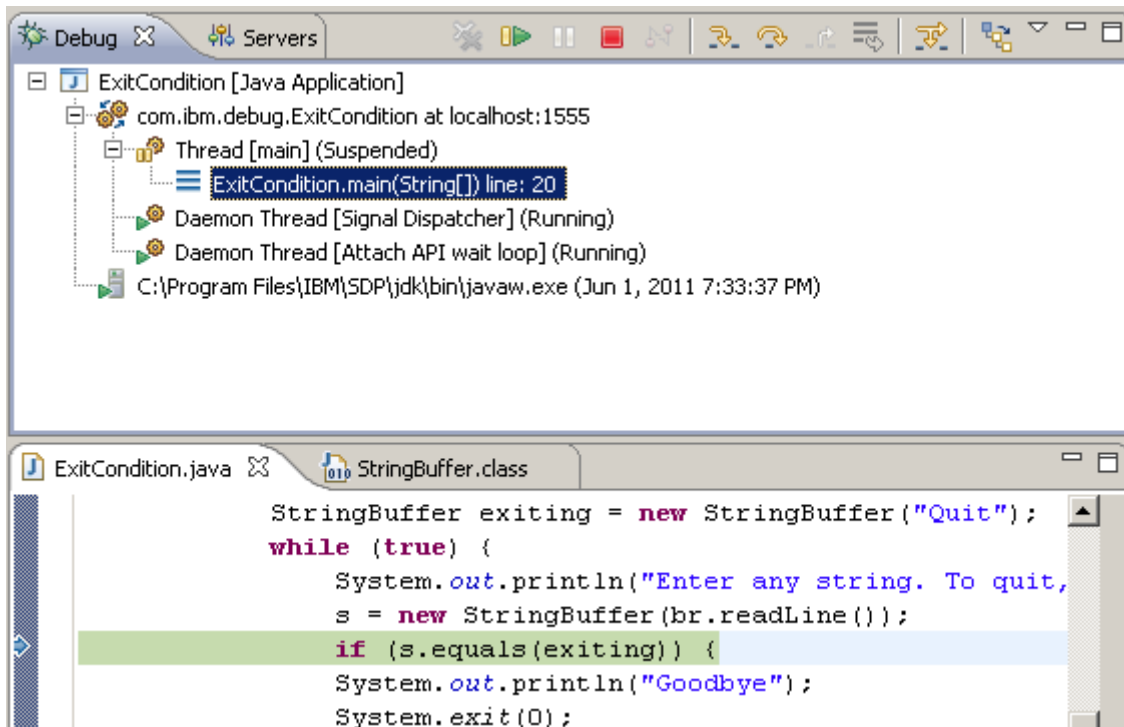
- ___ 4. Examine the code after another Step Into command.
- Click the **Step Into** button again to evaluate the `str.length()` method call at the current line.
 - Examine the Variables view. The object `this` now refers to the `string` object from the `str.length()` method call.



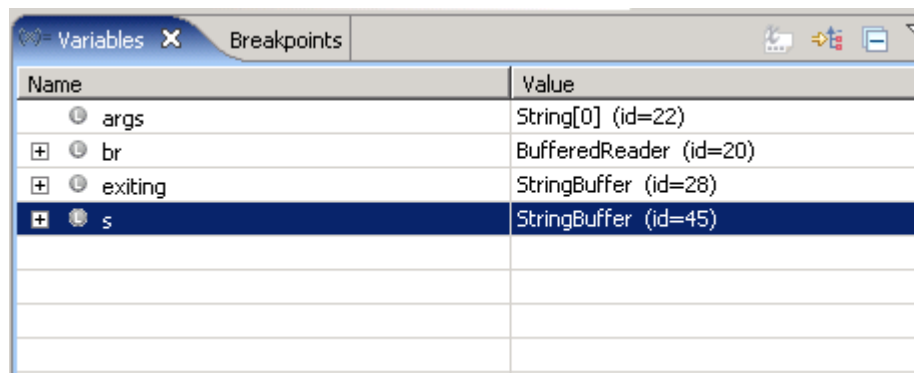
- ___ 5. Click **Step Return** or press the F7 button twice to return to the ExitCondition application level of scope. The Debug and Variables views have been updated to the corresponding scope.



- ___ 6. Examine the variables for comparing the user input to the exiting StringBuffer object.
- ___ a. Click the **Step Over** button to reach the line `if (s.equals(exiting))`.

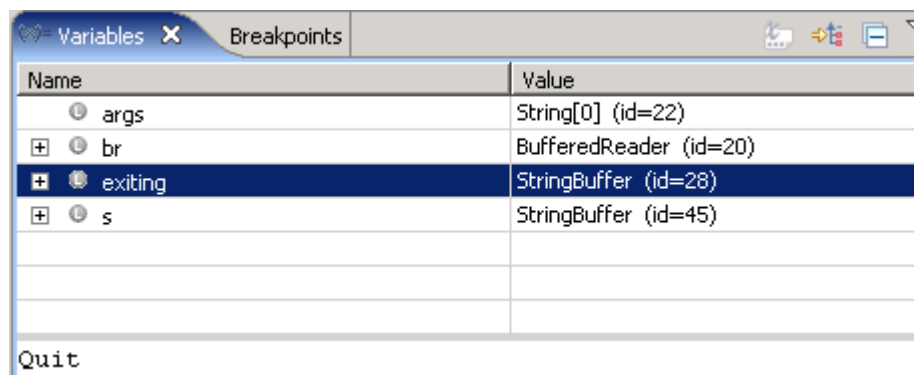


- ___ b. Examine the new variable `s` in the Variables view, created in the previous statement.
- ___ c. Confirm that the value of the variable `s` is `John Doe`. You can use the Detail pane to quickly check the String value assigned to each object.



Name	Value
args	String[0] (id=22)
br	BufferedReader (id=20)
exiting	StringBuffer (id=28)
s	StringBuffer (id=45)

- ___ d. Check the value for **exiting**. It is `Quit`, as assigned by your code.

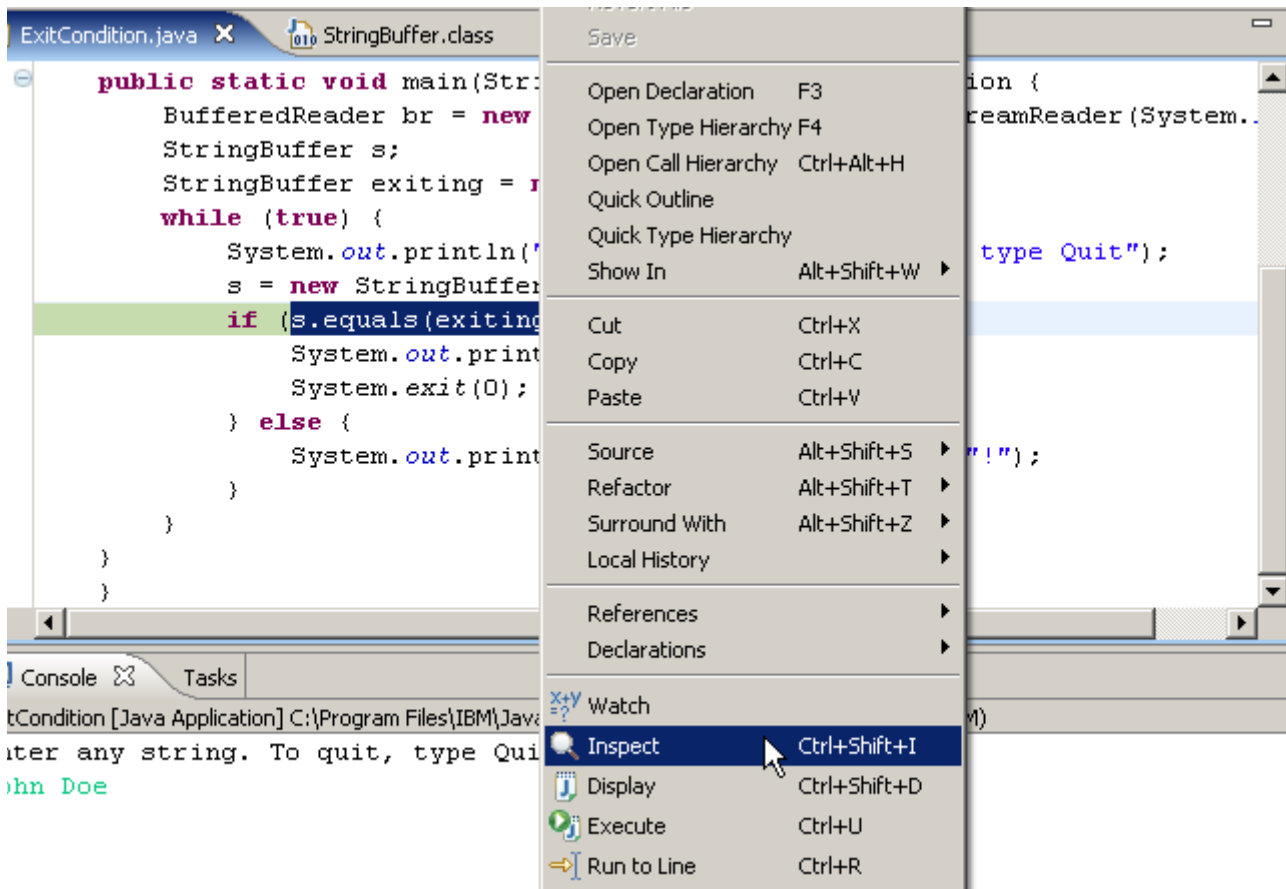


Name	Value
args	String[0] (id=22)
br	BufferedReader (id=20)
exiting	StringBuffer (id=28)
s	StringBuffer (id=45)

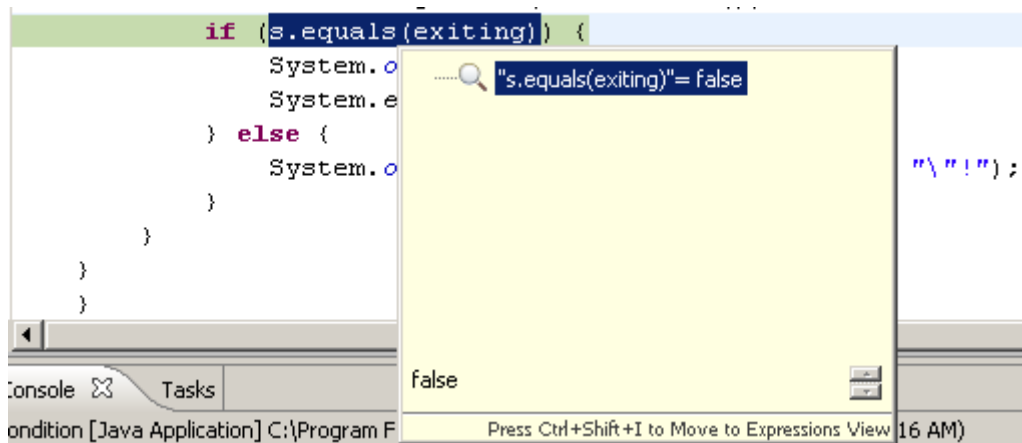
Quit

- ___ 7. Based on these values, you expect the `equals()` condition to return false. Verify this claim.
- ___ a. Highlight the expression `s.equals(exiting)` in the editor.

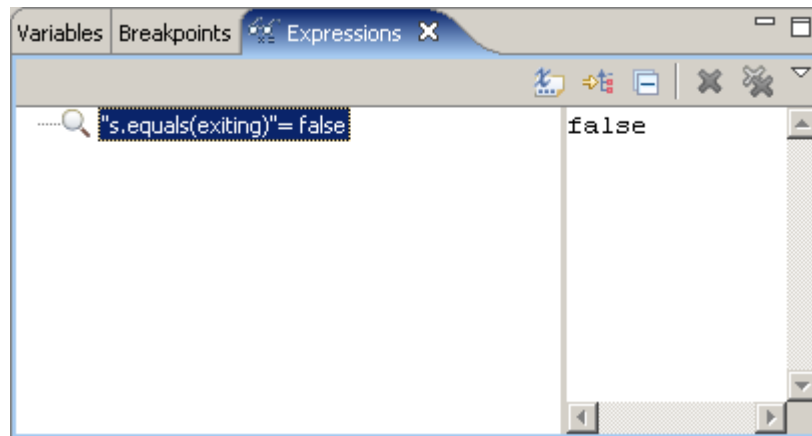
- ___ b. Invoke the pop-up menu by right-clicking the selection; then select **Inspect**.



- ___ c. A hover window opens and displays the result as false.



- ___ d. Press **Ctrl+Shift+I** to move the result to the Expression view.



- ___ e. Highlight the same expression in the editor again.

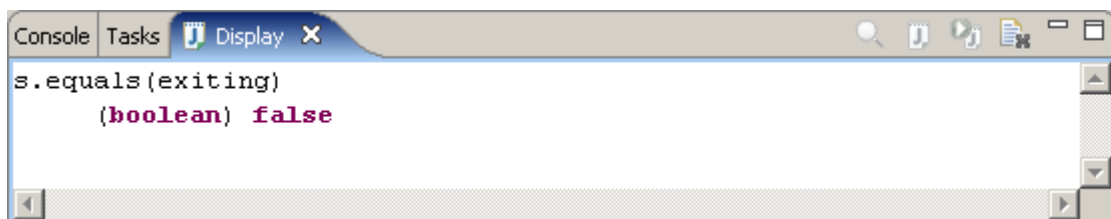
```
System.out.println("Enter any string. To quit, type Quit");
s = new StringBuffer(br.readLine());
if (s.equals(exiting)) {
    System.out.println("Goodbye");
    System.exit(0);
} else {
    System.out.println("You typed \"" + s + "\"!");
}
}
```

- ___ f. Right-click the expression and select **Display** from the pop-up menu.

- ___ g. This time, the same result is displayed as (boolean) false in a hover window.

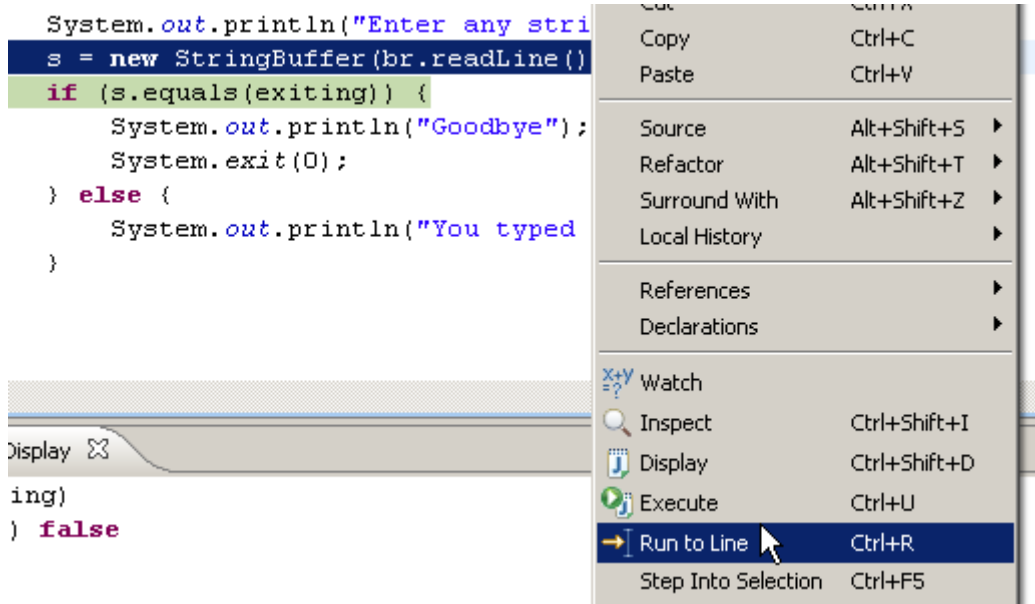
```
System.out.println("Enter any string. To quit, type ");
s = new StringBuffer(br.readLine());
if (s.equals(exiting)) {
    System.out.println((boolean) false);
    System.out.println("Press Ctrl+Shift+D to Move to Display View");
} else {
    System.out.println("You typed \"" + s + "\"!");
}
}
```

- ___ h. Press **Ctrl+Shift+D** to move the result to the Display view.

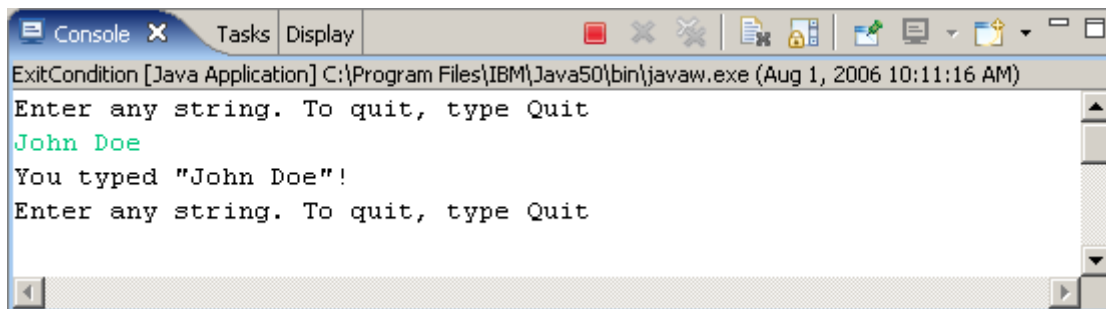


- ___ 8. You can now complete one iteration of the while loop, expecting to see the input echoed on the Console view.

- ___ a. In the editor, right-click the line `s = new StringBuffer(br.readLine());` and then select **Run to Line**.



- ___ b. The Console view now displays an echo of your input.

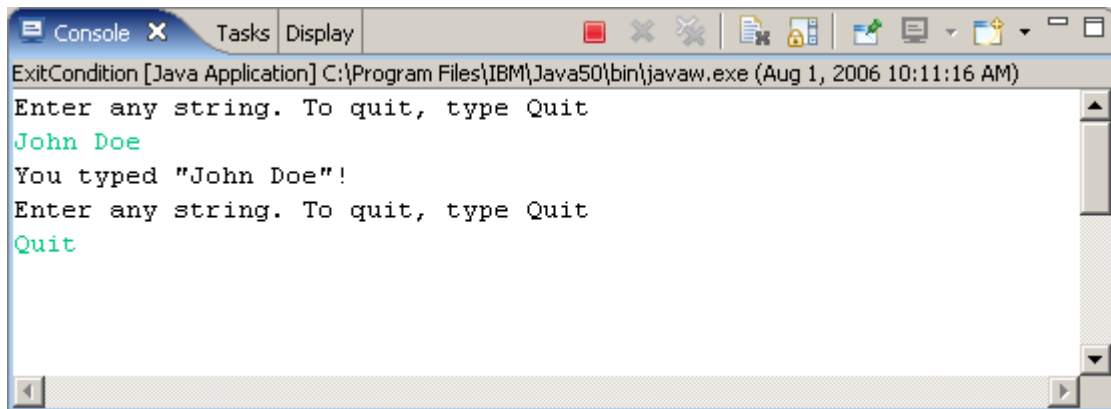


Part 6: Debugging the exit condition

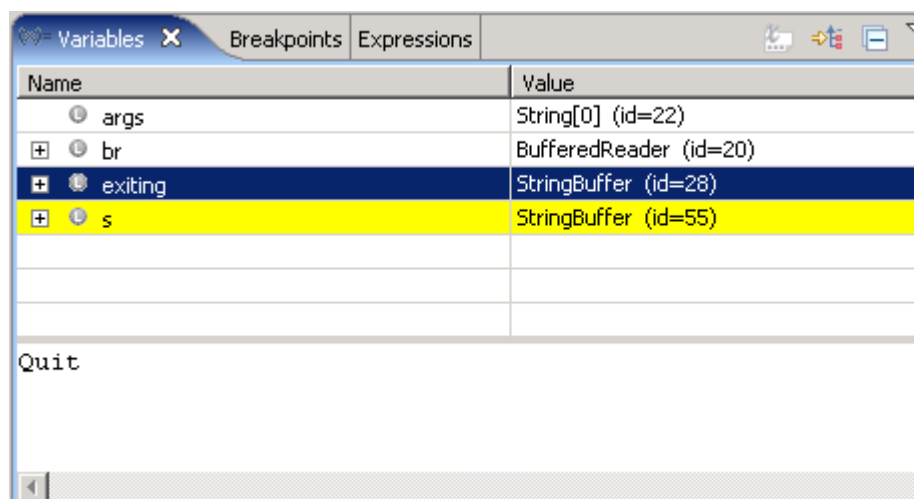
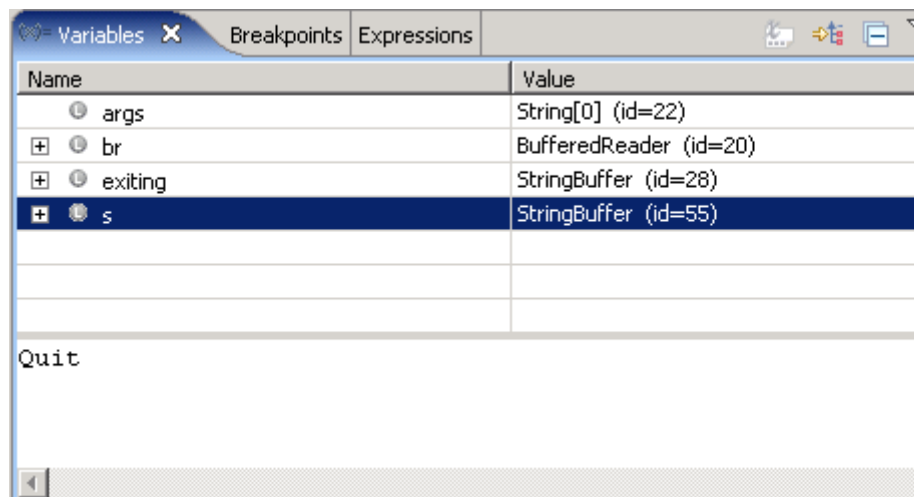
You now focus on the `s.equals(exiting)` statement in the `ExitCondition` class using the debugger. Specifically, you use the debugger to determine why the expression never evaluates to true.

- ___ 1. Determine why the `Quit` command does not terminate the application.
- ___ a. Click the Console view below the `Enter any string` prompt.

- ___ b. Type `Quit` and press Enter.

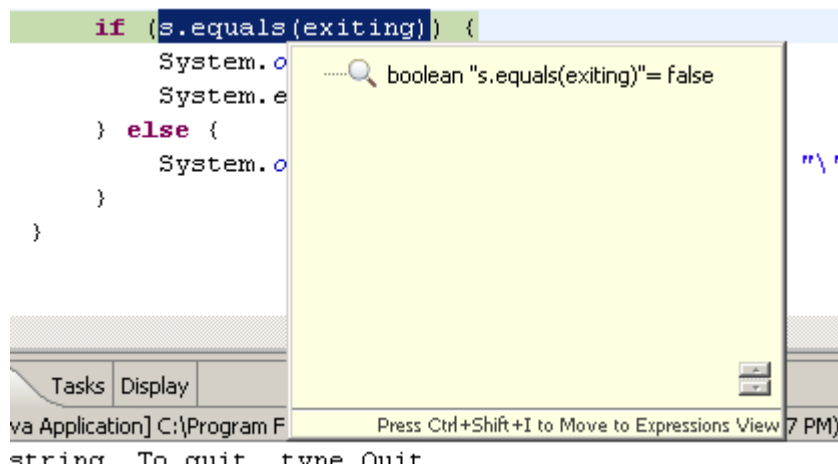


- ___ c. Click **Step Over**. The program is now at the `if (s.equals(exiting))` line.
- ___ d. In the Variables view, check the values for the variables `s` and `exiting`. They both have the value of `Quit`.



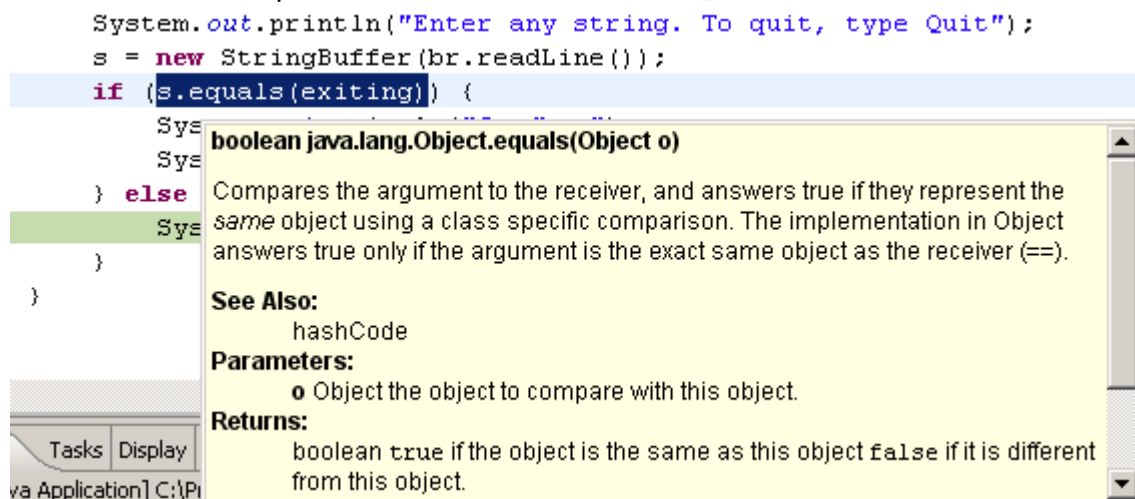
- ___ 2. Since both variables have the same string value, you expect `s.equals(exiting)` to evaluate to true. Use the debugging tools to test this claim.

- ___ a. Select the expression `s.equals(exiting)` in the editor.
- ___ b. Invoke the pop-up menu by right-clicking the selection. Select either **Display** or **Inspect** to evaluate the expression.
- ___ c. The expression evaluates to false.



- ___ d. Click the **Step Over** button and notice that the application now executes the else clause. It confirms that the `s.equals(exiting)` expression evaluates to false.
- ___ 3. Even though `s` and `exiting` appear to be equal, they are not being evaluated as equal. Why does the `equals()` method not find the string contents of the two objects to be the same? Use hover help to verify the class that implements this method.

- ___ a. Return to and highlight the `s.equals(exiting)` four lines up.
- ___ b. In the editor, position the cursor over the `equals()` method.

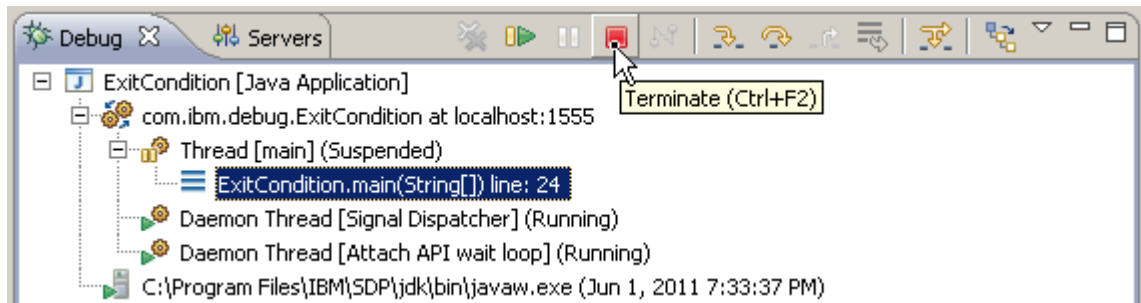


From the hover help message, you can see that the class implementing the `equals()` method is `Object`. Remember that the implementation of `equals()` in the `Object` class compares object references, not values. Since `s` and `exiting` represent two distinct `StringBuffer` objects, their object references do not have the same value.

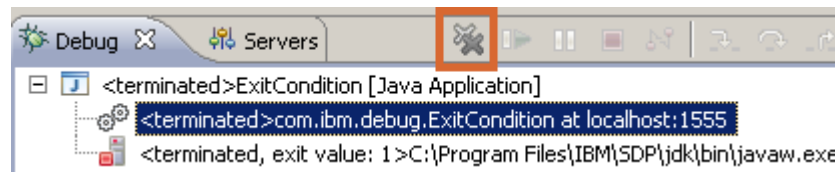
Part 7: Fixing the code and setting a breakpoint

You realize that the `StringBuffer` class does not override the `equals()` method inherited from `Object`. You cannot use it to compare string values. Instead, you use the `String` class because it overrides the `equals()` method, which compares the values stored in two strings.

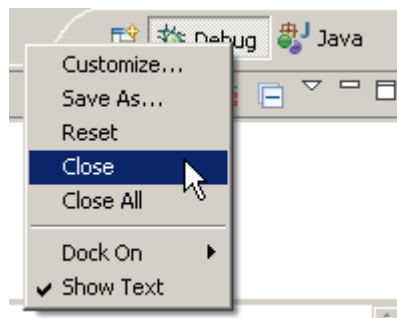
- ___ 1. Terminate the currently running instance of `ExitCondition`.
 - ___ a. Terminate the application by selecting **ExitCondition [Java Application]** in the Debug view and clicking the **Terminate** button on the Debug title bar.



- ___ b. Click the **Remove All Terminated Launches** button to clear the Debug view.

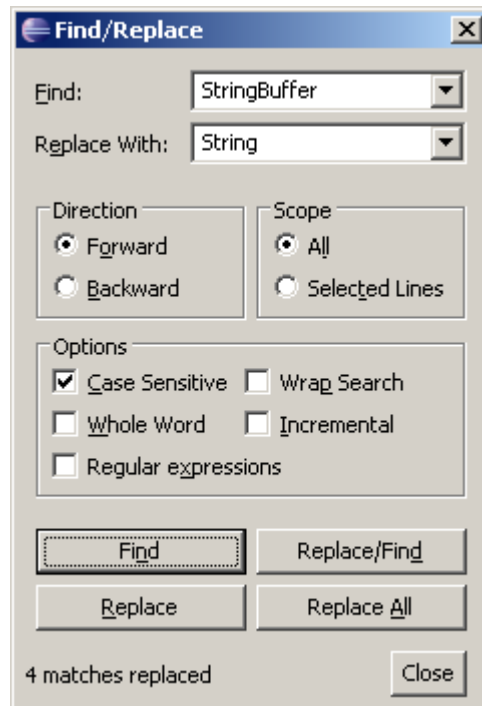


- ___ c. Close the Debug perspective by right-clicking the **Debug** icon in the perspective switcher toolbar and selecting **Close**.



- ___ 2. Change the `StringBuffer` objects to `String` objects in the `ExitCondition` class.
 - ___ a. Switch to the Java perspective (**Window > Open Perspective > Java**), if necessary.
 - ___ b. Open the `ExitCondition.java` file in a Java Editor view, if not already opened, by double-clicking the file in the Package Explorer view.
 - ___ c. Position the cursor at the beginning of the file.
 - ___ d. From the main menu, select **Edit > Find/Replace**.

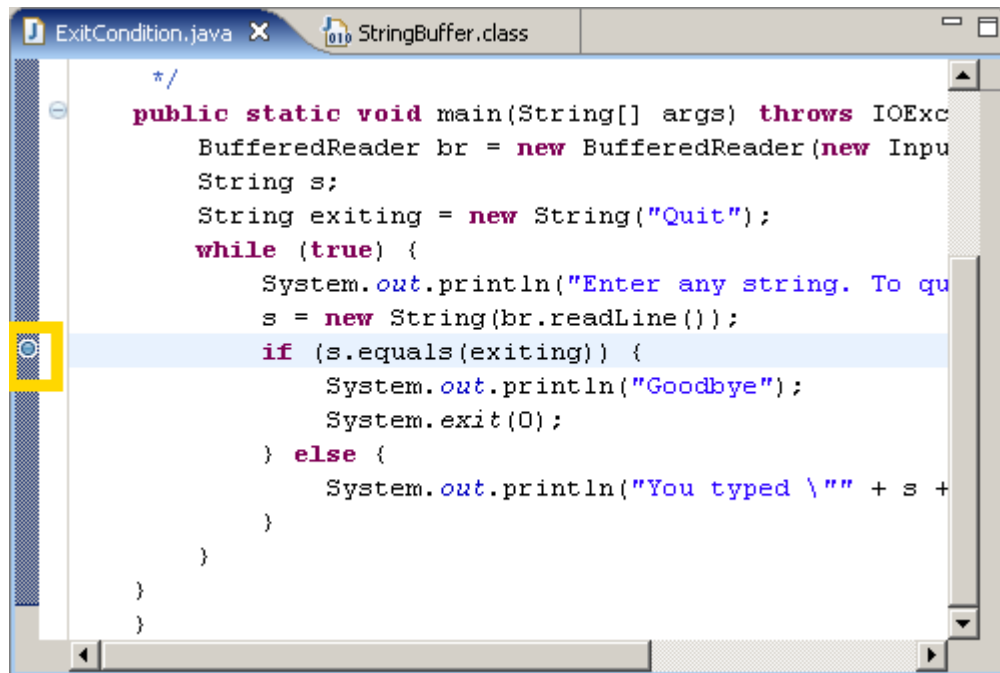
- ___ e. In the Find/Replace dialog, type `StringBuffer` in the **Find** field.
- ___ f. Type `String` in the **Replace With** field.
- ___ g. Select the **Case Sensitive** option.
- ___ h. Click **Replace All**. Confirm that four instances have been replaced.



- ___ i. Click **Close**.
- ___ 3. Save your changes (Ctrl+S).
- ___ 4. Set a breakpoint on the `if (s.equals(exiting))` line.
 - ___ a. In the Java Editor view containing `ExitCondition.java`, place the cursor on the line with `if (s.equals(exiting)) {`.
 - ___ b. Select **Run > Toggle Line Breakpoint** from the main menu. The breakpoint is indicated by a small blue dot in the gray margin to the left of the line.

**Note**

You can also double-click in the margin to the left of the line where you want the breakpoint set.



- ___ 5. You now test your updated ExitCondition application.
- ___ a. Click the **Debug** button on the main toolbar. Your program executes and pauses for user input.

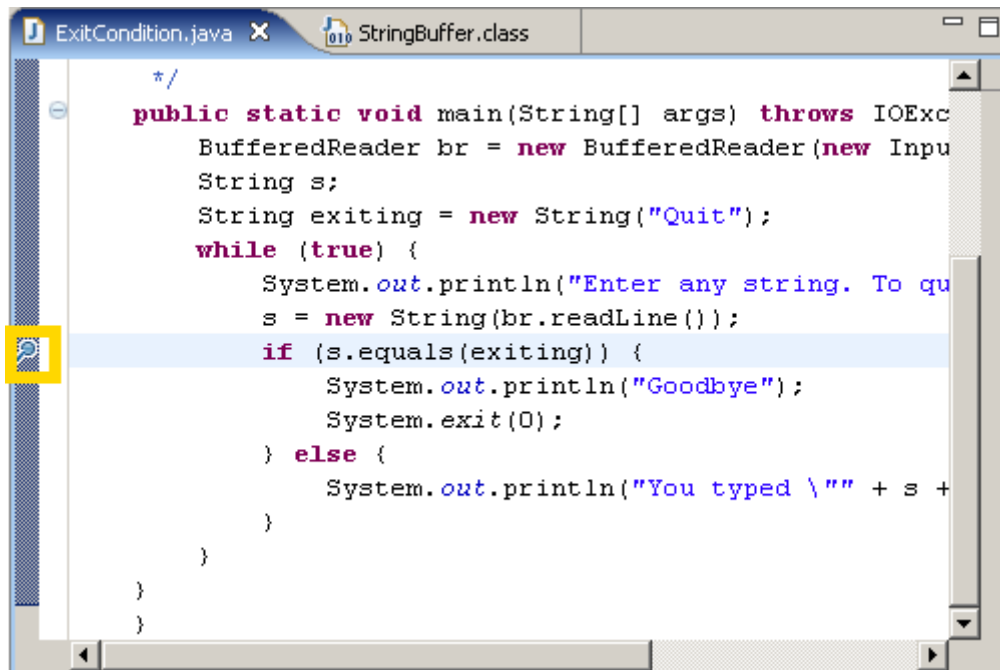


- ___ b. Open the Debug perspective by selecting **Window > Open Perspective > Debug** from the main menu.

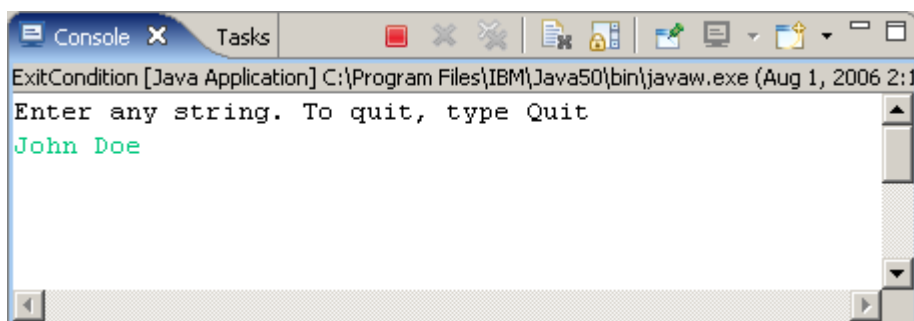


Note

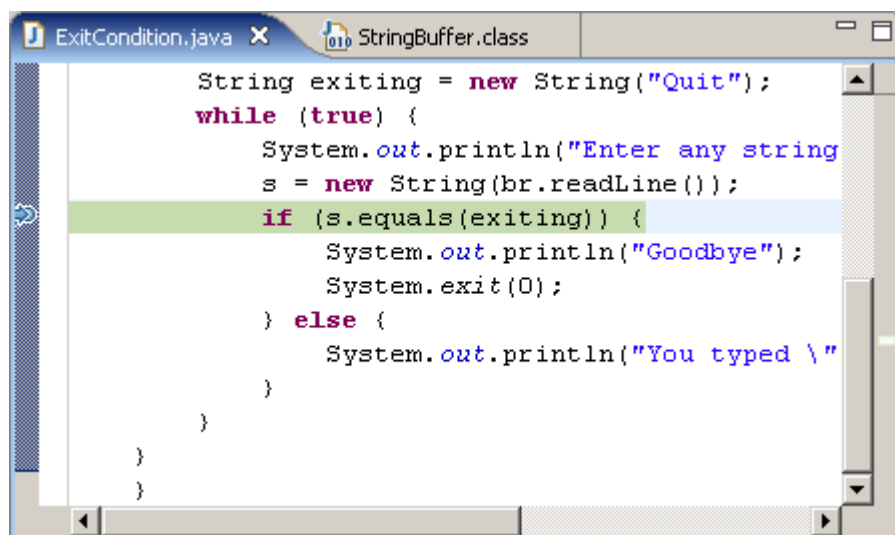
Notice in the editor view that the breakpoint symbol in the left margin now has a check mark. The icon in the margin indicates that the breakpoint is enabled and has been verified by the Java virtual machine.



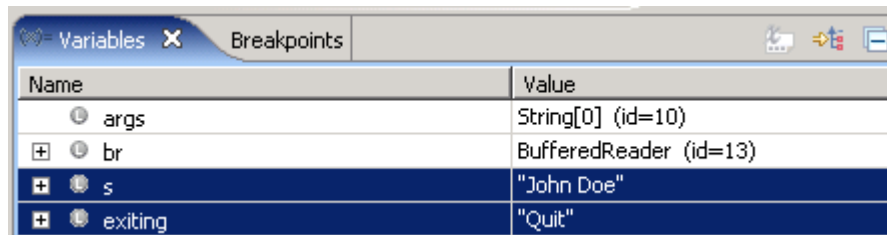
- ___ c. In the Console view, type `John Doe` and press Enter.



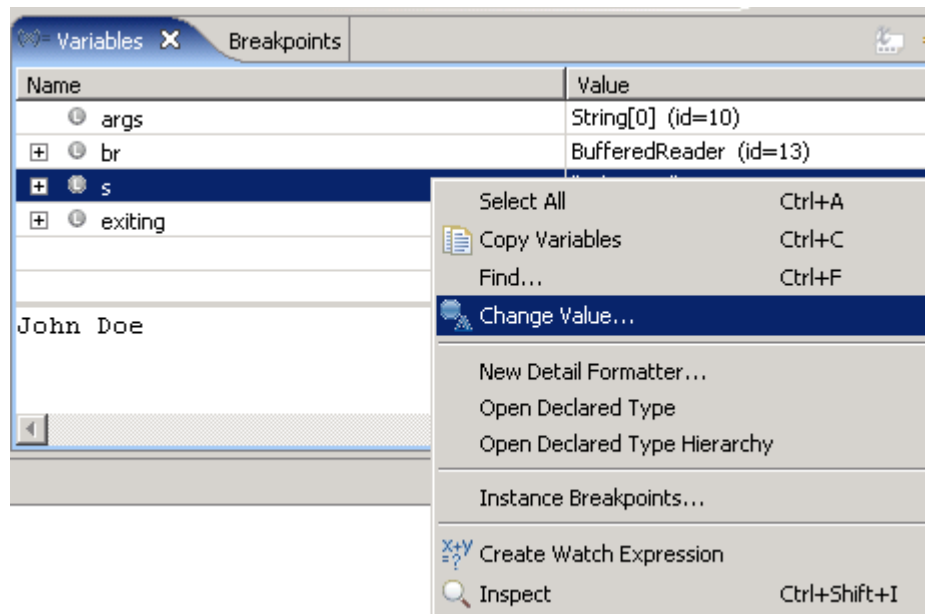
Execution is suspended at the line where the breakpoint has been set.



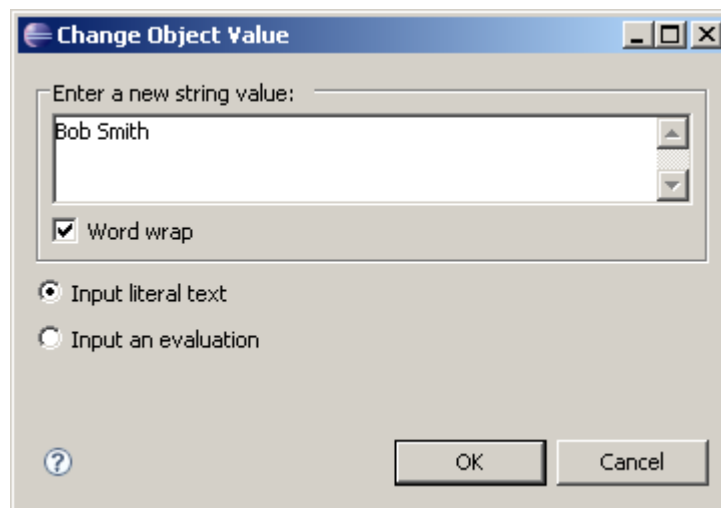
- ___ d. Notice the value of variables `s` and `exiting` in the Variables view. They are now String objects and have different values.



- ___ 6. To highlight the variable modification feature of the debugger, change the value of the `s` variable.
- ___ a. In the Variables view, select the variable `s`.
- ___ b. Invoke the pop-up menu by right-clicking the selection. Select **Change Value**.



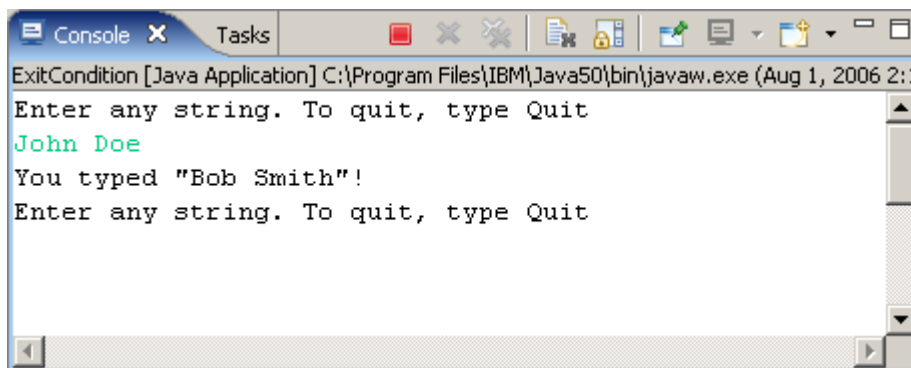
- ___ c. In the Change Object Value dialog, type `Bob Smith` in the text field and click **OK**. The value of `s` has now been changed.



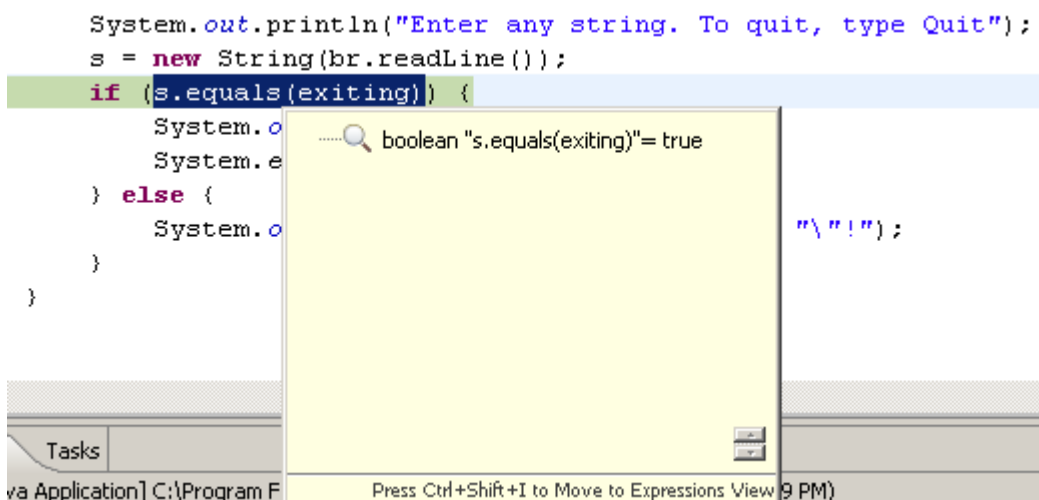
- ___ d. Click the **Resume** button in the Debug view toolbar to continue execution.



- ___ e. Observe the text in the Console view. Notice that the application reflects the new value for the `s` variable.
- ___ 7. Test the `Quit` command in the `ExitCondition` application.
- ___ a. Click the area below the input prompt in the Console view.
- ___ b. Type `Quit` and press Enter.
- ___ c. The execution suspends at the breakpoint on the line with the expression `s.equals(exiting)`. Notice in the Variables view that both `s` and `exiting` now have the same value of `Quit`.

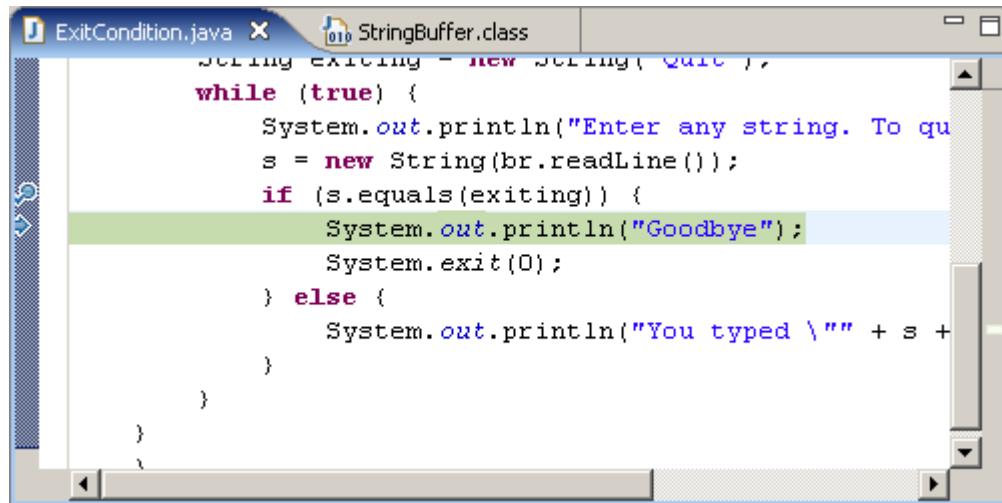


- ___ d. In the Java Editor, select the `s.equals(exiting)` expression. Invoke the pop-up menu and select **Inspect**.
- ___ e. Examine the hover window. Check that the statement evaluates to true.



- ___ f. Click the **Step Over** button.

- ___ g. Notice that the next statement highlighted in the editor view is in the if statement. This highlighting confirms that the equality check worked.

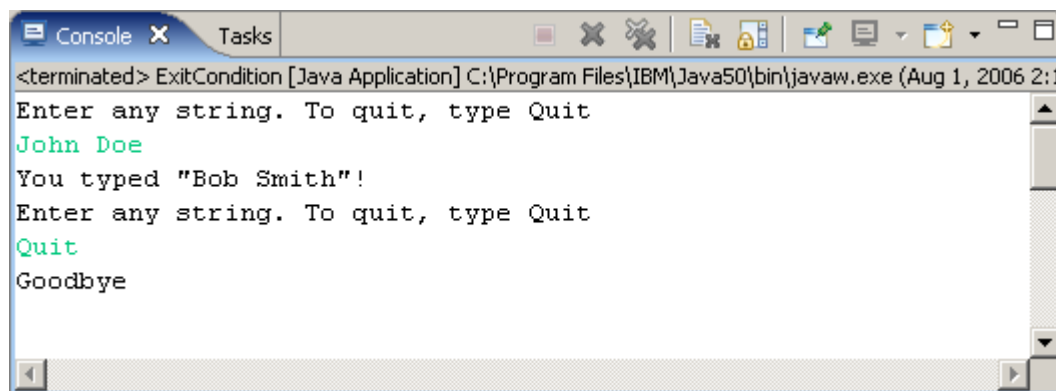


```

String exiting = new String( "Quit" );
while (true) {
    System.out.println("Enter any string. To qu
    s = new String(br.readLine());
    if (s.equals(exiting)) {
        System.out.println("Goodbye");
        System.exit(0);
    } else {
        System.out.println("You typed \"" + s +
    }
}

```

- ___ h. Click the **Resume** button in the Debug view to complete the execution of the program. The program terminates, and the Console view displays the expected Goodbye message.



```

<terminated> ExitCondition [Java Application] C:\Program Files\IBM\Java50\bin\javaw.exe (Aug 1, 2006 2:1
Enter any string. To quit, type Quit
John Doe
You typed "Bob Smith"!
Enter any string. To quit, type Quit
Quit
Goodbye

```

- ___ 8. Close the Debug perspective by right-clicking the **Debug** icon in the perspective switch toolbar, and select **Close**.

End of exercise

What you did in this exercise

In this exercise, you debugged an application running on your local machine. You learned how to launch your application in debug mode in order to diagnose and fix errors. You used the tools provided by the debugger to step through code, view and modify variables, and set breakpoints. You also experimented with the various views available on the Debug perspective.

Solution instructions

Solution files are provided for this exercise. However, most learning takes place as you complete the steps to debug your Java application; therefore, solution files are to be used as a reference only.

- ___ 1. Follow the directions in Appendix B to import the solution file to this exercise using **File > Import**, and then select **Import Existing Projects into the Workspace** from the Import wizard.
 - ___ a. Import the project from `labfiles\Debugging\solution\Debug`.
- ___ 2. Read through each part of the exercise to understand the code and to understand how the Debug perspective is used to debug a Java application.

