Using The NetBeans BBj IDE To Develop And Debug - Tutorial Part II

By John Schroeder

In the first part of this series, presented in the 2Q2002 Advantage e-zine, we showed how to use the editor in the NetBeans-based BBj® IDE to create a program that reads a Customer file and displays the results in a grid. Next, we used the editor and several new BBj objects to build our program. We ended the article with the code shown in **Figure 1**. (As we mentioned, this code is available on our web site, as well as on the CD in this edition.)

Figure 1. In this code sample from the Source Editor, the program copies the Customer file and moves the required data into the grid.

We then saved the program, using the right click in the editor area and selected Save. You can also save by clicking on the **Save** tool button, using the **File/Save** menu, or the keyboard shortcut, [CTRL]+[S]. Once the program is saved, you can run it by pressing **F6**, or by right clicking on the program name in the Explorer window and selecting **Execute**.

When we ran the program, the results were incorrect (**Figure 2**). All of the rows in the grid contained the same data! That's not quite what we wanted. Obviously there is a bug in the program, and it is the job of the BBj Debugger to help find it.

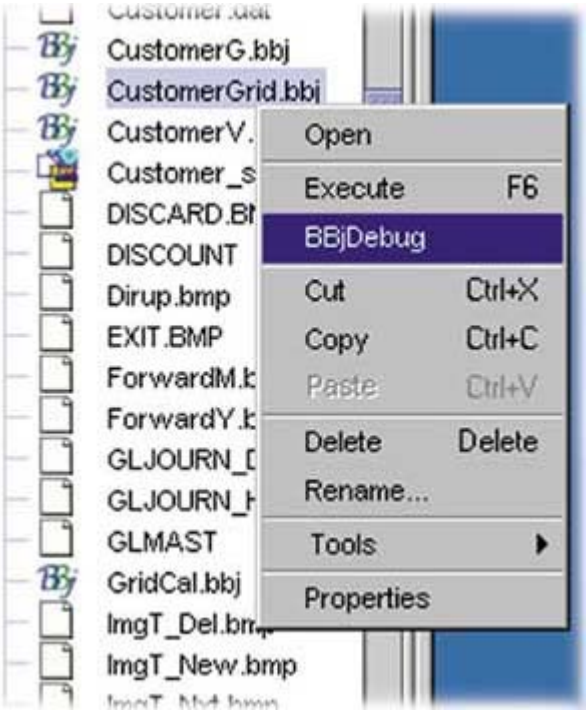
Figure 2. This grid displays the same Customer in each row.

Before going into the debugger, we can probe a bit further to narrow the area where the bug exists. The appearance of the main window is correct. If we double click on any cell in the grid, the proper information is displayed in a message box. If we click on the **Close** box, the program exits. By following these steps, we have ruled out the setup and event handler code for this problem. Now let's focus on the section of code where the Customer data is being put into the grid.

In PRO/5® we would probably put an ESCAPE in the program somewhere in the REPEAT/WEND loop between lines 14 and 29, where the grid is being built. (For line numbering, go to **Tools/Options/BBjDev/Editing/BBj Source/Line #s/True**). We would then save the program and run it. When the program hit ESCAPE, we'd step through the code, printing variables as we went, trying to isolate the bug. The BBj IDE includes the BBj Debugger, which makes this task a lot simpler.

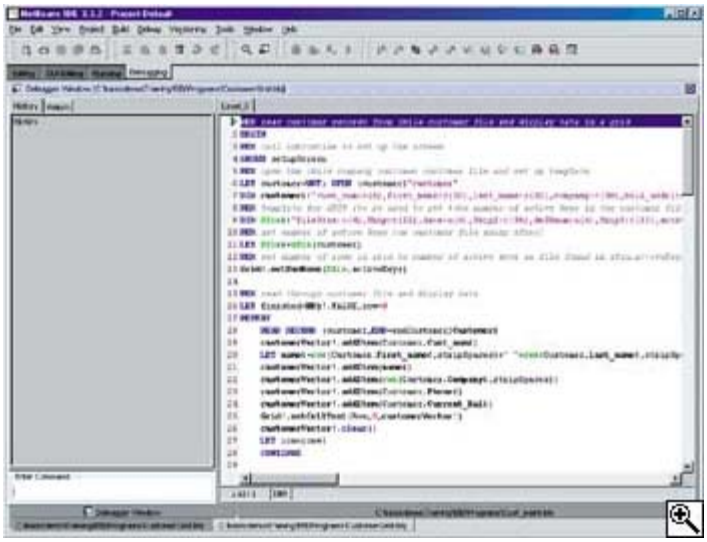
To start the BBj Debugger, right click on the program name (**CustomerGrid.bbj**) in the **Explorer** window and select BBjDebug (**Figure 3**). This loads the program into the BBj Debugger.

Figure 3. Select the BBjDebug function in the Explorer window.



As you can see in **Figure 4**, the debugger has three parts: the **History/Watch** windows are in the upper left, the command prompt box is at the lower left and the **Edit** window is on the right. The **Edit** window displays the text of the program, with the program pointer highlighting the current program line. In this case, it's line 1.

Figure 4. History/Watch windows are on the left, command prompt box is at the lower left and Edit window is on the right.



Unlike the editor, the BBj Debugger is linked to the BBj Interpreter. Errors in the code are highlighted; the code can be run, and commands entered and executed, just as in console mode. The command prompt box at the lower left of the debugger window is where console commands or step commands are entered. Think of it as the READY > prompt. The step command is the same in BBj as in PRO/5. A period, or "dot," followed by an optional number and the Enter key, causes the program to execute one line, or the specified number of lines, and come back to console mode, or READY. The single "dot," without a number, is referred to as single step mode. The **History** window shows all the commands entered in the command prompt as well as program lines executed in step mode.

The **Watch** window displays variables that have been selected for WATCHing. When a variable is WATCHed in a step mode, the value displayed is updated as it changes. This is convenient for observing the changes in variables, as the program executes one line at a time. There is no need to put PRINT statements into your code and save it before you can debug the program. (Nor is there any need to worry about what you left in the production version!)

The BBj Debugger supports the use of breakpoints in the code for stopping execution at a specific place. This is similar to adding an ESCAPE to the code, but with breakpoints there are two advantages: 1. The breakpoints are not part of the code, so you do not have to remember to remove them from a "production version." 2. The breakpoints can be saved in a separate file that is associated with the program being debugged. They can then be reused over the course of several debugging sessions. This is also useful if your application module consists of several RUN or CALLed programs. Each program in the module can have its own associated breakpoints and the entire sequence of programs can be debugged as needed.

Breakpoints can be set on any line(s) of the program. When a running program encounters a breakpoint, it stops with the program counter on the breakpoint line, and the command prompt is enabled. You can type any valid command or step through lines in the program(s) from this point. You can also type the following:

```
.watch (variable)
```

where variable is any initialized variable. This displays the variable name and its current value in the Watch window.

Let's apply this to the problem at hand. We know there is a problem with the code in the grid loading section somewhere between lines 14 and 29. Add a breakpoint at line 16, run the program, then single step through the code and observe the behavior. To set a breakpoint, right click anywhere on the line in question and select "Toggle Breakpoint." You can also set the breakpoint by putting the cursor on a line and pressing **Shift+F8**. A red ball displays in the margin, and the line is highlighted in red, as shown in **Figure 5**.

Now type "run" at the command prompt. The program will run until it comes to a breakpoint (Line 16) and then stop, waiting for a console command.

Figure 5. At a breakpoint (Line 16), a red ball displays in the margin, and the line is highlighted in red.

Since we are loading the customerVector! to build a row of data in the grid, let's watch how this object changes as we step through the program. To do this type **.watch customerVector!** (**Figure 6**), then click on the **Watch** tab at the top of the **History/Watch** window. You can see that the vector is empty (**Figure 7).**

Figure 6. Type .watch customerVector!

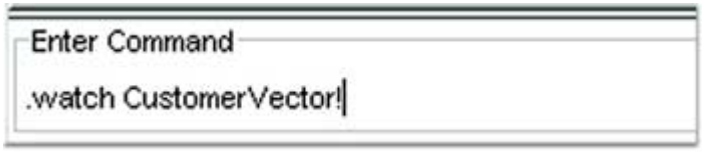
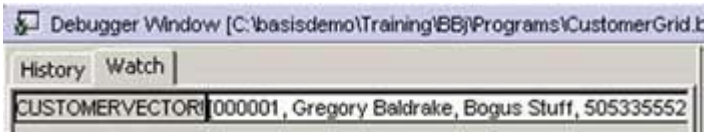


Figure 7. Click on Watch tab. The vector is empty.

Now, at the command prompt, single step for a few lines until the customerVector! changes. You can see how the vector changes as each field from the file is added.

Figure 8. The customerVector! changes as each field from the file is added.



Let's step a few times through the loop to see how the customerVector! behaves. As we step through, it appears as if the vector is not changing (**Figure 8**). However, if you look at the scroll bar at the bottom of the **Watch** window, you can see it has changed. It indicates that there is more to see in the vector if we scroll to the right (**Figure 9**). It also seems as if every record in the file is being added to the vector. How will this affect the display?

Figure 9. Scrolling the bar at the bottom of the Watch window to the right indicates the vector has changed.

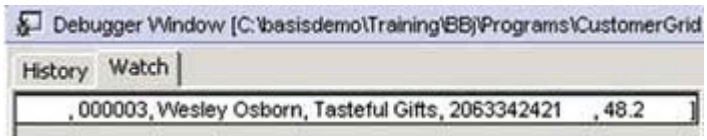
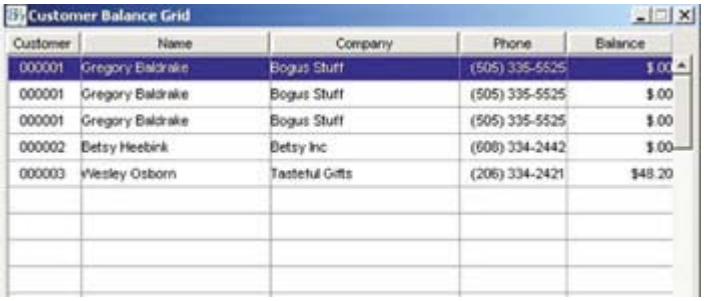


Figure 10. The console command makes the window visible.

We can use a console command to make the window visible (**Figure 10**). Then, we observe as we continue to single step. The command is shown in the command prompt. We use the setVisible() method of the BBjWindow object to make the grid visible as we fill it with data (at the command window under the **History** tab).

Figure 11. The top three rows contain the first Customer's information, while the last two contain the second and third Customers, respectively.



Looking at the grid (**Figure 11**), we see that the top three rows contain the first Customer's information, while the last two contain the second and third Customers, respectively.

Let's single step again and see where it takes us. Stepping through the loop once more, we see that the first Customer now occupies the first four rows of the grid, and that the fourth Customer has been added to the last row (**Figure 12**). As we go through the loop, we add another Customer to the vector, and move the entire vector into the next available row in the grid. Running through the entire file, we eventually would have a vector that contained every record in the Customer file. Each time we moved the vector into the grid, we would place the data from the first Customer into the next row number.

Figure 12. The first Customer occupies the first four rows of the grid. The fourth Customer has been added to the last row.

Subsequent rows are filled with the data from the remaining Customers in the vector. Each time we move down a row and insert the vector into the grid, we replace the data in that row with the first Customer's data, fill the next several rows with the data from the second Customer, then the third Customer, etc. Since there is a fixed number of rows in the grid, by the time we move the vector into the last row of the grid, the first Customer's data overlays whatever is there. So the grid is now filled entirely with the first Customer's data. By using the BBj Debugger's breakpoints, watch window, and console command access, we have pinpointed the problem. Now what do we do about it?

There are a few possible solutions: First, we can continue to build the vector until it has every record in the file and hold off moving the vector into the grid until this is complete. Then, we can insert the vector into the top row of the grid, filling the entire grid with the needed data. An alternative would maintain the loop logic as it is, inserting the vector into the next row as we read a Customer record. In this case, however, we would have to clear out the contents of the vector, after we moved it into the grid, so that only one Customer existed in the vector at any one time. Thus, only one Customer record would be moved into the grid at a time, and we solve the problem. Clearing the vector is simple. The BBjVector object has a method called clear() that wipes the vector clean. If we insert the line:

```
customerVector!.clear()
```

at the beginning or the end of the loop, we will have fixed the bug (**Figure 13**). We can do this in the debugger window, inserting this line between the row increment and the CONTINUE statement. To do this, place the cursor at the end of the row increment line, press [Enter] and type in the line. Now click on the margin beside line 23 to remove the breakpoint.

Figure 13. Insert CustomerVector!.clear() as in line 23.

When you run the program, you get the results shown in **Figure 14**. These are indeed the expected results, with all of the records in the Customer file displayed, one per row, in the grid.

Figure 14. When you run the program, you will get the correct Customer data in each row.

Customer	Name	Company	Phone	Balance
000001	Gregory Baldrake	Bogus Stuff	(505) 335-5525	$.00
000002	Betsy Heeblink	Betsy Inc	(608) 334-2442	$.00
000003	Wesley Osborn	Tasteful Gifts	(206) 334-2421	$48.20
000004	Harry Chuckie	Long Island Inc	(619) 782-9214	$.00
000005	Stephanie McIntyre		(209) 210-7793	$.00
000006	Dave Strum	FishLand	(301) 272-9987	$9.00
000007	Dr. Jane Booker	Greencastl Moose Farm	(717) 337-0198	$.00
000008	Cyndy Sikes	Gourmet Giftware		$60.70
000009	Krista Nugent	Utopian Cattle	(505) 773-2913	$.00
000010	Ms. Jane Ghast	Massage Land	(414) 956-3332	$19.99
000011	Karen Dennison	Rexall Drugs	(505) 273-7751	$.00
000012	Rhonda Frenchi	Bing Bong Gift co	(505) 299-3321	$.00
000013	Ernie Augustine		(505) 265-1563	$.00
000015	Amy Alcott	Misha Gifts	(505) 267-9971	$.00

You can see how the BBj Debugger makes it easier to pinpoint and solve a problem in code. With a full view of the code, combined with breakpoints, the **History/Watch** window, and the Command prompt all in one integrated tool, you now have considerably more debugging power at your fingertips to track down and solve problems. Coupled with powerful editing features, the new BBj IDE propels developing and debugging Business BASIC applications to a new plane.