[Skip to content](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch06.html" \l "container)

* [*Safari*](https://www.safaribooksonline.com/home/)
* [Recommended](https://www.safaribooksonline.com/r/)
* [Queue](https://www.safaribooksonline.com/s/)
* [Search](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch06.html)
* [Expand Nav](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch06.html)
  + [Recent](https://www.safaribooksonline.com/recent/)
  + [Topics](https://www.safaribooksonline.com/t/)
  + [Highlights](https://www.safaribooksonline.com/u/p253751/)
  + [Settings](https://www.safaribooksonline.com/u/)
  + [Feedback](mailto:feedback@safaribooksonline.com)
  + [Sign Out](https://www.safaribooksonline.com/accounts/logout/)
  + [Settings](https://www.safaribooksonline.com/u/)**10** days left in your trial. [Subscribe](https://www.safaribooksonline.com/subscribe/).
  + [Feedback](mailto:feedback@safaribooksonline.com)
  + [Sign Out](https://www.safaribooksonline.com/accounts/logout/)

[Software Test Engineering with IBM Rational Functional Tester: The Definitive Resource](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch06.html)

* [Search in book...](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch06.html)

Top of Form

Bottom of Form

* [Toggle Font Controls](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch06.html)
* [Share this](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch06.html)
  + [Twitter](https://twitter.com/share?url=http://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html&text=Software%20Test%20Engineering%20with%20IBM%20Rational%20Functional%20Tester%3A%20The%20Definitive%20Resource&via=safari)
  + [Facebook](https://www.facebook.com/sharer/sharer.php?u=http://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html)
  + [Google Plus](https://plus.google.com/share?url=http://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html)
  + [Email](mailto:?subject=Safari:%20Chapter%204.%20XML%20and%20Rational%20Functional%20Tester&body=http://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html%0D%0Afrom%20Software%20Test%20Engineering%20with%20IBM%20Rational%20Functional%20Tester%3A%20The%20Definitive%20Resource%0D%0A)

[Prev](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch05.html)

[Chapter 5. Managing Script Data](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch05.html)

[Next](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html)

[Chapter 7. Managing Script Execution](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html)

**Chapter 6. Debugging Scripts**

**Chip Davis, Daniel Chirillo, Daniel Gouveia**

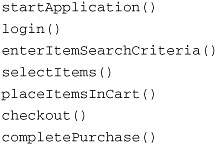
*Soon after you begin creating your scripts, you might find yourself in a situation where a script you develop isn’t doing what you want it to do and your first debugging tool—watching script execution and reading the RFT log—does not provide enough useful information for you to pinpoint the root cause of the problem.*

*Eclipse and Visual Studio have powerful, integrated debuggers. You might want to take advantage of the many features offered in both. We limit our discussion here to the critical features you need to start using your IDE’s debugger. If you’ve never used a debugger before, you might feel intimidated at first. That’s normal. We encourage you to practice using the debugger; in many cases, it can save you the tremendous time it costs to identify the causes of problems.*

**Common Debugging Concepts**

This chapter shows the Eclipse and Visual Studio debuggers. It begins with an explanation of some concepts that are common to debugging in both environments.

To make describing these concepts easier, refer to the following theoretical script that consists of these method calls:



Imagine that your script has a problem. Before looking at Eclipse and Visual Studio debuggers in action, defining some key terms that are used in both development environments is helpful. In the discussion of these terms in the following sections, the method names in this theoretical script are referenced.

**Breakpoints**

A *breakpoint* is a statement in your script before you want to pause. A script can have any number of breakpoints. You can put them in methods that are called by scripts and to which the source code is available. Typically, you put breakpoints where you have problems or suspect a problem might occur.

**Stepping**

*Stepping* though a script refers to executing the script one statement at a time. As described in this chapter, there are two variations on a definition of a *single* statement.

**Step Over**

The term *step over* can sometimes be misleading to people who are new to using a debugger. To the untrained ear, step over might be interpreted as “skip this statement” or “step over, i.e., don’t execute this statement.” This is not the case. To step over a statement means to execute that statement (or, the first method that’s called in a *single* statement that contains multiple method calls) in its entirety. For example, if you put a breakpoint on the line that contains the call to login(), when you run the debugger, Rational Functional Tester pauses before executing login(). When it pauses, you have the opportunity to examine the current values of variables. You can also execute the next statement, login(). Stepping over login() executes all of login(), which internally, consists of multiple method calls. You step *over* a method call when you’re not interested in examining what occurs *within* that method.

**Step Into**

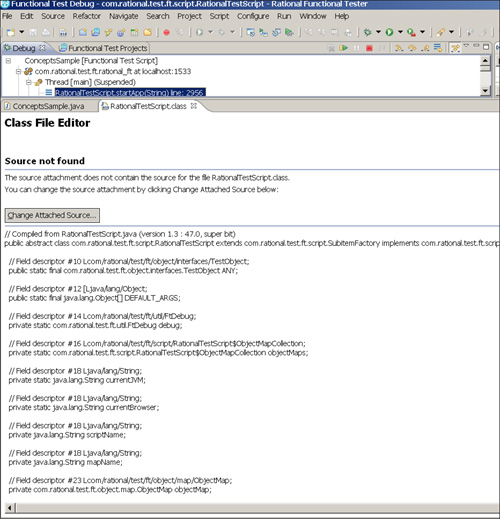
What if you have a breakpoint on the line in your script that calls login() and you suspect that the source of your current problem might be related to something that occurs within the login() method? In that case, if you have the source to login(), then, if you *step into* (instead of stepping over) it, you are brought into the source of the login() method. You then see the first statement in login(). At this point, you can choose to step over or into methods called in the login() method. After the last statement is executed, you are brought back to your script, and execution is paused at enterItemSearchCriteria().

What happens if you step into a method to which you don’t have the source? For example, what if you try to step into a Rational Functional Tester method such as the following?

startApp( "ClassicsJavaA" )

In Visual Studio, the effect is that of a step over. You get no complaints; Rational Functional Tester simply executes the method in its entirety. In Eclipse, if you step into a method to which you don’t have the source, you see what’s shown in [Figure 6.1](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch06.html#ch06fig01).

**Figure 6.1** Eclipse’s Source not found class file



Do not worry. If this happens, all you need to do is execute a *step return* (discussed in the next section).

**Step Return**

If you step return, Rational Functional Tester executes the rest of the current method (pausing at any breakpoints along the way that might be set) and brings you back (*returns*) to the method from which you stepped into the current method. After you’ve returned, you can continue stepping though the code.

**Debugging in Eclipse**

Rational Functional Tester provides a rich debugging environment in the Eclipse shell. You have the ability to set breakpoints, step through individual lines in your script, set hit counts for loops, and change variables at runtime. Working with these capabilities is easy, and they provide you with the necessary information to quickly debug any issues in your scripts. This section introduces you to the Rational Functional Tester’s debugging environment in the Eclipse development environment.

Before any debugging begins, the first thing you need to do is set at least one breakpoint. You set a breakpoint by following these steps:

**1.** Find the line of code in your script at which you wish to pause execution.

**2.** Right-click in the vertical editor ruler (located on the far left of the editor).

**3.** Select **Toggle Breakpoint**.

Alternatively, you can set a breakpoint by clicking on the desired line and selecting **Run** → **Toggle Breakpoint** from the RFT menu.

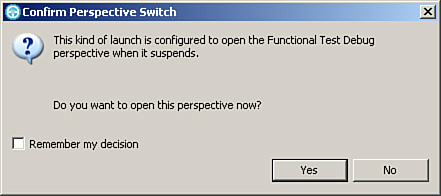
Doing this places a small blue dot in the gray column on the left side of the script. [Figure 6.2](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch06.html#ch06fig02) shows the process of setting a breakpoint.

**Figure 6.2** Setting a breakpoint



After your breakpoint is set, you can begin the process of debugging your scripts. This requires you to run your script in “debug” mode. You do this by selecting **Script > Debug**. Alternately, you can press the **Shift + F11** keystroke combination. In either case, your script starts as it usually does. When the breakpoint in your script is hit, you receive a dialog box asking you to switch to the **Functional Test Debug** perspective (see [Figure 6.3](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch06.html#ch06fig03)). Click the **Yes** button and Rational Functional Tester switches to the debug perspective.

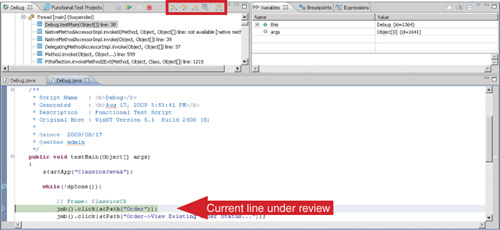
**Figure 6.3** Confirm Perspective Switch dialog box



The Functional Test Debug perspective contains multiple views. One view that you work with often is the Debug view. This view is used with the Script editor, usually found in the middle of the Functional Test Debug perspective. The Script editor displays the contents of your script, using a blue arrow and highlighting to reveal the line that is currently under review. The idea is for you to watch this view as you utilize the step commands in the Debug view to execute individual statements in your script. This helps you pinpoint which line in your script is causing the issue.

[Figure 6.4](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch06.html#ch06fig04) highlights Script and Debug views. The callout arrow shows how the debugger flags the current line under review, using a small blue arrow (on the left side) and highlighting. The box at the top of the figure highlights the buttons that engage the step commands. These commands are discussed in the introduction to this chapter. They enable you to step into, step out of, and step over individual lines of scripts. The Eclipse debugger also provides a means to set up step filters, helping to expedite the debugging process.

**Figure 6.4** Functional Test Debug perspective—Debug and Script views



While using the debugger, you control the statements you want to step into/over/return from using the buttons in the Debug View:

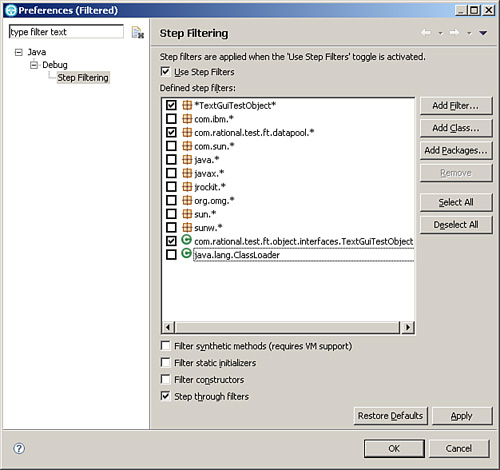
• The first button in the box is the Step Into command. Clicking this button lets you step into the method the current line of code is trying to execute. You can also engage the Step Into command by pressing the **F5** key. This lets you dive deeper into what a particular line in your script does. For instance, if a line calls out to a method to which you have the source code, the Step Into command lets you jump into that method and step through each line of code, analyzing its behavior.

• The second button in the box is the Step Over command. You can either click this button or press your **F6** key to engage the functionality. As discussed earlier in the chapter, stepping over a line of code enables it to execute normally and move on to the next statement in your script. You use this command when you are confident that a certain line isn’t the problem. It prevents you from diving deeper into what the method does. Using the example in the preceding paragraph, you use the Step Over command if you are confident that your method works correctly. This enables the line in your script that calls out to the method to execute without actually entering into the method.

• The third button in the box is the Step Return command. This enables you to jump out of a method that you stepped into. It is useful when you dive deep into Rational Functional Tester’s base methods or even the base methods of any Java classes that you are using. You click on this button to return back to the point in your debug session where you clicked the **Step Into** button.

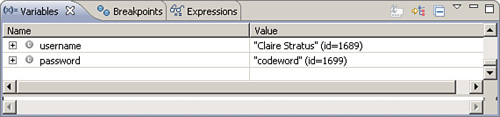
• The fifth button in the box is the Step Filters command. This is an advanced capability that enables you to filter out certain elements that you do not wish to step through. You use this if you know that your script issue doesn’t lie within a certain class or package. There are two steps to follow to use Step Filters. The first is to click on the **Step Filters** button. This tells the debugger to filter out the classes, packages, and user-specified filters listed in the Step Filtering preferences. The second step is to specify the filters. You do this by right-clicking the Debug view and selecting the **Edit Step Filters** option. This opens the Step Filtering preferences dialog box (shown in [Figure 6.5](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch06.html#ch06fig05)). If you click the **Add Filter** button, you can specify a pattern to filter. For instance, if you didn’t want to step through any part of your script that dealt with text fields, you could specify **\*TextGuiTestObject\*** as your pattern. The Add Class button lets you select a specific class to skip over when stepping through script. For instance, instead of specifying a pattern, you can simply type TextGui in the class field and it displays the TextGuiTextObject in the list box. You can then select that class, adding it to the filter list. Lastly, the **Add Package** button lets you omit an entire set of classes. Perhaps you don’t want to deal with stepping through Rational Functional Tester’s datapool classes. You can type com.rational.test.ft.data. This displays the packages that match your search string, com.rational.test.ft.datapool being one of them. You select this package, adding it to the filter list. These are displayed as the checked boxes in [Figure 6.5](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch06.html#ch06fig05). Ultimately, you use Step Filters to hone in on the problem by skipping over the pieces you know aren’t part the issue, such as the com.rational.test.ft.datapool package and the TextGuiTestObject class. The goal of this capability is to expedite your debugging session.

**Figure 6.5** Step Filters preferences—Step Filtering list



The Functional Test Debug perspective also offers a Variables view (see [Figure 6.6](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch06.html#ch06fig06)). You can use this view with the step commands. This view enables you to see what the current values of your variables are. When you execute a line in your script that modifies a variable, you can watch to see if the value was set correctly. This becomes a valuable view when your script (or custom method) deals with setting and passing variables. You can watch values being assigned to your variables, looking for proper assignments.

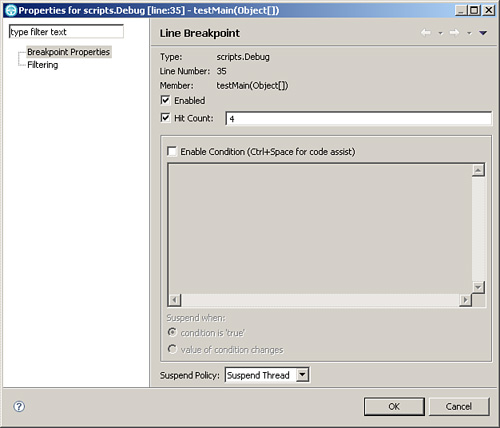
**Figure 6.6** Functional Test Debug perspective—Variables view



Sometimes issues occur within a loop. For instance, you may have a while loop that parses a file of usernames, line-by-line, and enters them into your application. You notice that toward the end of the file, Rational Functional Tester places an incorrect name into the application. Typically, you step through your script (or method), looking for the issue. This is tedious because you know you have to step through the loop a number of times before you get to the point where you can start investigating where the actual problem is. The Eclipse debugger provides a way to alleviate this. You can edit the breakpoint properties and provide a hit count. Hit counts tell functional tester to pause the execution after the *nth* time the breakpoint is hit.

To set a hit count, right-click on your breakpoint (remember the blue dot?) and select **Breakpoint Properties**. This opens a dialog box that displays the breakpoint’s properties. You then need to place a check in the **Hit Count** checkbox and provide an integer (this is the number of times Rational Functional Tester executes the breakpoint before it pauses playback). [Figure 6.7](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch06.html#ch06fig07) shows the execution is suspended after the fourth time the breakpoint is hit. If the breakpoint is set on the while line in your script, the loop is executed four times before the breakpoint is engaged and execution is paused. This is a useful feature for getting to the necessary point in your loop for debugging, without having to uselessly step through the loop multiple times (or modify the source code to simply keep debugging).

**Figure 6.7** Breakpoint properties—Hit Count



Rational Functional Tester’s debugger offers another means to save time during your debug sessions. It enables you to change the values of your variables while debugging your script. This is a great feature to test different values to use with your script’s variables *without* having to stop the debugger, set a new variable value, and restart the debugger. Here’s how to change a variable value while debugging:

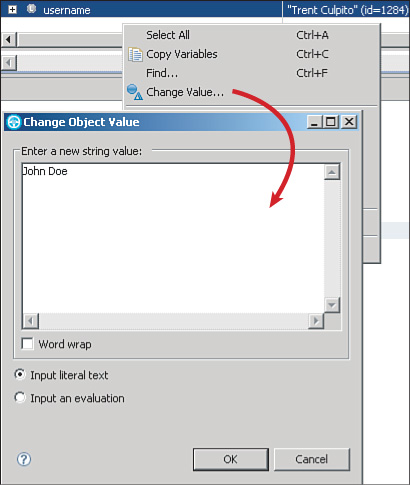
**1.** Right-click on the variable that you want to change (in the Variables view).

**2.** In the **Change Object Value** window, replace the old value with a new one.

**3.** Click the **OK** button.

[Figure 6.8](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch06.html#ch06fig08) shows these steps. The username variable was initially set to Jack Thompson. Using these steps, the value was changed to John Doe. This is while the script is being debugged.

**Figure 6.8** Editing variable values



**Adding Breakpoints While Debugging**

Sometimes, as you step through a script in debug mode looking for problems, you discover a place where you want to add a breakpoint (for the next time you execute the script in debug mode or for the next time you enter a loop). You can add one while you’re debugging (you don’t need to be in edit mode). Right-click in the vertical editor ruler and select **Toggle Breakpoint**.

**Removing, Disabling, and Enabling Breakpoints**

If you decide that you don’t need one of your breakpoints, you can remove or disable it. Disabling it is useful when you think you might want to restore that breakpoint at some point in the future. Disabling a breakpoint preserves a visual marker in IDE. If you need to use the breakpoint again, you can enable it.

To remove a breakpoint, right-click and select **Toggle Breakpoint**.

To disable a breakpoint, right-click and select **Disable Breakpoint**. The breakpoint marker turns from blue to white.

To enable a breakpoint, right-click and select **Enable**; the marker becomes blue again.

You can remove all breakpoints in a single step by selecting **Run > Remove All breakpoints** from the RFT menu.

**Debugging in Visual Studio**

This section discusses debugging Rational Functional Tester Visual Basic .NET test scripts. This section assumes that you already understand basic debugging concepts, such as breakpoints, stepping, and variable watching. These topics are discussed in the “[Common Debugging Concepts](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch06.html#ch06sec1lev1)” section of this chapter. For more information on this topic, refer to the Microsoft Developer Network (MSDN®) Developing with Visual Studio.NET (building, debugging, and testing).

Like most debugging environments, you must execute your program in debug mode to enable the debugging features, such as breakpoints. Furthermore, you must execute Rational Functional Tester scripts from the Debug Functional Test Script playback mode, *not* the normal Visual Studio debug playback. If you run a test script from normal test playback mode or from the Visual Basic .NET debug mode, you cannot debug your test script. You can initiate Debug Functional Test Script playback from the **Script > Debug** menu shown in [Figure 6.9](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch06.html#ch06fig09), *not* the Debug menu. You can also debug test scripts from the toolbar button highlighted in [Figure 6.10](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch06.html#ch06fig10).

**Figure 6.9** Debugging functional test scripts in Visual Studio



**Figure 6.10** Debugging functional test scripts toolbar button



Note

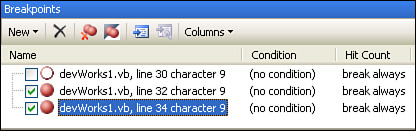
Rational Function Tester Java scripting in Eclipse provides a Test Debug Perspective, which is referenced in online documentation. There is no such perspective in Visual Studio.NET.

**Setting Breakpoints**

Setting a breakpoint enables you to pause execution of a test script when it is run in Debug Functional Test Script playback. You can set and remove breakpoints in several ways: using the Debug menu, pressing **F9**, clicking in the left margin (next to the line number if shown) or clicking a toolbar button (not added by default). In addition to adding and removing breakpoints, Visual Studio enables you to enable and disable breakpoints. When a breakpoint is disabled, it leaves a placeholder at the specific line in the script without actually breaking when run. This can help make debugging easier because it lets you remove breakpoints without forgetting where you initially set them.

Visual Studio also has a Breakpoints window that lists all the breakpoints currently set in your script and helps you monitor and manage them (see [Figure 6.11](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch06.html#ch06fig11)). You can access the Breakpoints window from the **Debug > Windows > Breakpoints** menu, pressing **Ctrl+Alt+B**, or by adding a toolbar button.

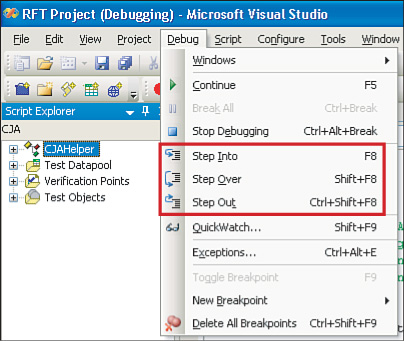
**Figure 6.11** Managing breakpoints in Visual Studio



**Step Into, Step Over, and Step Out**

As previously mentioned, you cannot start debugging a Rational Functional Tester script from the Visual Studio Debug toolbar or the Debug menu. (You must use the Debug Functional Tester Script button.) However, after you begin the debug playback, you can use the Visual Studio menu or toolbar to control debugging, including stepping. Stepping enables line-by-line control of the script playback (see [Figure 6.12](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch06.html#ch06fig12)).

**Figure 6.12** Stepping through a test script playback



The most common debug stepping you will probably use is step over, which moves the line pointer to the next line of the test script. The keyboard shortcut, which can be useful when debugging to step over, is Shift+F8. The reason this is called step *over* is that it skips the underlying code of the test script line and moves to the next line of the test script. For example, if you use step over from the following line, the debug pointer does *not* move into the SubScript1 test script but continues on to the next line in the current script:

CallScript("SubScript1")

If you have scripts that call other scripts, you might use step into. You might also use this if you have a tricky test playback problem, although this is not common. Step into moves the debug pointer into the module of the line being executed. For example, if you use step over from the following line, the debug pointer moves into the first line of the SubScript2 test script:

CallScript("SubScript2")

If you used step into from the following line, the debug pointer moves into the first line of the script helper class, into the function for the test object PlaceAnOrder (you are not likely to do this):

PlaceAnOrder().InputChars("12345")

If you have used step into to move the debug pointer into another script or class, then you might use step out. Step out returns the line pointer back to the calling class script. For example, if you debug a script that was called from another script, step out moves the debug pointer back out to the calling test script.

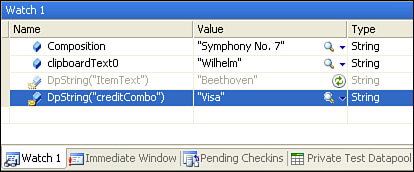
If you use step out from the highest level, or the initial calling test script, or if you run a single test script that is not called from another, step out acts the same as if you selected **Debug > Continue**. Whenever you no longer want to step through a test script, you can press **F5** to continue playback to the next breakpoint, or to the end of the test.

**Changing Variable Values at Runtime**

When you debug a Rational Functional Tester script, you can use the Watch window to monitor the values of any variables you have in your script. These might be simple counters, datapool values, or any other script variable. You might need to see what the value is at a particular point in the script or you might want to change the value during playback. You can manage up to four different Watch windows to help organize the variables you want to monitor.

The easiest way to monitor a variable is to right-click on the variable (anywhere in the variable name) while paused at a breakpoint and select **Add Watch**. This automatically shows the Watch window with the variable along with its current value. Note that you can do this only while the script is playing back in debug mode, although Visual Studio remembers variables that were added from previous debug runs. As you step through the script or pause at breakpoints, the Watch window shows the current variable values at that point in the script (see [Figure 6.13](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch06.html#ch06fig13)).

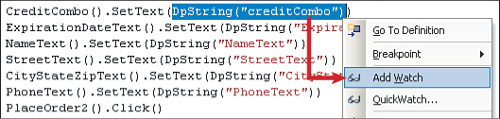
**Figure 6.13** Viewing the value of a script variable



You can change the value of script variables while executing a script in debug mode from the Watch window. Right-click on the variable you want to change and select **Edit Value**. You can change the values of test data, captured from or input to the application under test. You might also do this with a counter value to skip to the end of a looping structure.

You can add a datapool value to a Watch window similarly to the way that you add a script variable. You must select the entire field DpString("name") to get the value of the datapool. If you only set the cursor in this field, or if you select only part of the function name, the Watch window does not display the correct value from the datapool (see [Figure 6.14](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch06.html#ch06fig14)).

**Figure 6.14** Viewing a datapool value during debug



Note that you cannot change a datapool value as you can other script variables. This is because the value is read from a file (not traditional file input/output) directly into the test script. If you need to change values taken from a datapool, either as part of normal test execution or for debugging, you must set the datapool value into an intermediate variable first, and then use this variable in place of the datapool call (e.g., dim creditCard as String = DpString ( "creditCombo" ).

**Adding Breakpoints While Debugging**

Sometimes, as you step through a script in debug mode looking for problems, you discover a place where you want to add a breakpoint (for the next time you execute the script in debug mode or for the next time your code enters a loop or method). You can add one while you’re debugging (you don’t need to be in edit mode). Right-click in the vertical editor ruler and select **Toggle Breakpoint**.

**Summary**

Debugging is a critical skill that you develop with time and experience. Like any skill, you need to exercise it to develop it. Hopefully, this chapter has helped give you the confidence and knowledge to use the debugger in your development environment.

[Prev](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch05.html)

[Chapter 5. Managing Script Data](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch05.html)

[Next](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html)

[Chapter 7. Managing Script Execution](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html)

**Welcome to Safari**. Remember, your free trial will end on March 7, 2015, but you can [subscribe at any time](https://www.safaribooksonline.com/subscribe/)

* [Recommended](https://www.safaribooksonline.com/r/)
* [Queue](https://www.safaribooksonline.com/s/)
* [Recent](https://www.safaribooksonline.com/recent/)
* [Topics](https://www.safaribooksonline.com/t/)
* [Settings](https://www.safaribooksonline.com/u/)
* [Blog](http://blog.safaribooksonline.com/)
* [Support](http://support.safaribooksonline.com/?prod=flow)
* [Support](http://msupport.safaribooksonline.com/?prod=flow)
* [Feedback](mailto:feedback@safaribooksonline.com)
* [Sign Out](https://www.safaribooksonline.com/accounts/logout/)

© 2015 [Safari](http://www.safaribooksonline.com/).   
[Terms of Service](https://www.safaribooksonline.com/terms/) / [Membership Agreement](https://www.safaribooksonline.com/membership-agreement/) / [Privacy Policy](https://www.safaribooksonline.com/privacy/)

[**Make font larger Make font smaller**](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch06.html)