[Skip to content](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.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/ch07.html)
* [Expand Nav](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.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/ch07.html)

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

Top of Form

Bottom of Form

* [Toggle Font Controls](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html)
* [Share this](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.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/ch06.html)

[Chapter 6. Debugging Scripts](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch06.html)

[Next](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch08.html)

[Chapter 8. Handling Unsupported Domain Objects](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch08.html)

**Chapter 7. Managing Script Execution**

**Daniel Chirillo, Daniel Gouveia**

*Managing script execution consists of tasks such as manipulating playback settings, using conditional and looping logic to optimize the flow of your scripts, handling errors your scripts encounter, building regression scripts (for example, scripts that call other scripts), and executing your scripts outside of the Rational Functional Tester environment. This chapter takes you through each of these tasks and teaches you how to enhance your scripts further. You are armed with an arsenal of techniques that can help you transform your recorded scripts into robust functional testing assets.*

**Setting Execution Options Within Your Scripts**

Previous chapters discussed setting some of the playback options via Rational Functional Tester’s global settings. These enable you to adjust logging, tweak the ScriptAssure™ logic, and tune different delays. The problem that teams encounter when working with these global settings is that they are specific to an instance of Rational Functional Tester. This means that if you change your settings to get a script to successfully play back, everybody on your team needs to modify their settings to match yours—if they want to play your script back or include it in their regression test. The way to escape this dependency is by manipulating the outside of Rational Functional Tester’s global playback settings. Rational Functional Tester enables you to set execution options right in your scripts.

This chapter teaches you how to release your dependency on global playback settings. You learn how you can set these from within your scripts. This enables you and your team to work within an environment where you are all not constantly adjusting your playback settings to run other team members’ scripts.

**Manipulating Playback Options Within Scripts**

Rational Functional Tester provides programmatic access for manipulating execution options. These enable you to adjust playback settings right within the body of your script. This provides an excellent means for you to control the behavior of your scripts without depending on Rational Functional Tester’s global playback settings.

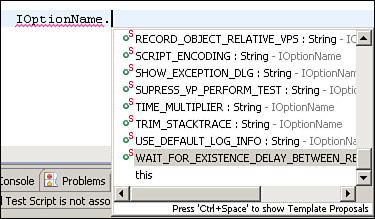
If you find that your script is struggling at certain points—perhaps facing synchronization issues with your application—you first should adjust the global playback settings in Rational Functional Tester. This helps to easily pinpoint the necessary option that would fix the playback of your script. The idea is to adjust the value of one option, re-execute your script, and see if that fixes the problem. After you find the right setting, make a note of it along with the value that you provided. You can then reset it back to its original value (easily accomplished by clicking the **Restore Defaults** button). This information enables you to work with Rational Functional Tester’s Application Programming Interface (API) to modify that particular setting in your script. The first thing you need to understand is the IOptionName interface, which is a simple means to access the different options that enable you to control your script’s behavior. The complete list of fields (options) and their descriptions for this interface can be found in the *Rational Functional Tester API Reference*.

The basic means of working with this interface is to explicitly call it with the desired option name. For example, a call to this, in either version of the tool (VB.NET or Java), looks similar to the following example:

IOptionName.USE\_DEFAULT\_LOG\_INFO

Using the information you garnered from your experiments with the global playback settings, you peruse the IOptionName field settings in the *Rational Functional Tester API Reference* and find which one matches the option you found to fix your problem. You would then reference it and select it from the content assist dialog, shown in [Figure 7.1](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07fig01).

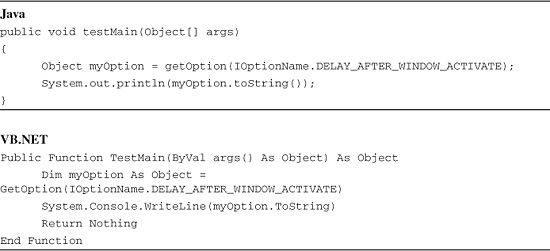
**Figure 7.1** IOptionName—field choices



IOptionName provides you with only the options you need to work with. You manipulate them using one of three methods: getOption, setOption, and resetOption. These enable you to adjust your scripts’ behaviors at any point.

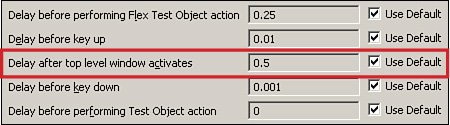
As its name suggests, getOption enables you to retrieve the current value of a given option. This comes in handy if you want to create some logic in your script that first acquires a given playback setting’s value and then adjusts execution accordingly. [Listing 7.1](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex01) provides code examples that show the basic usage pattern of getOption.

**Listing 7.1** Using getOption() in the context of a script



The sample scripts in [Listing 7.1](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex01) acquire the current value of the **Delay after top level window activates** setting. If you run these scripts, the return value of the getOption/GetOption is displayed in the console window. In this case, **0.5** is returned. This is the default value for the **Delay after top level window activates** setting (see [Figure 7.2](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07fig02)).

**Figure 7.2** Global playback settings—other delays and Delay after top level window activates



Changing the value of an IOptionName field is accomplished using setOption. The basic usage pattern for this method is to specify the option you wish to change and the value to which you wish to change it. For example, if your application has touchy mouse-over menus that take less than a second to display one minute and five seconds to display another minute, you can set the MOUSE\_HOVER\_TIME option to address this.

**Java**

setOption(IOptionName.MOUSE\_HOVER\_TIME, 5.0);

**VB.NET**

SetOption(IOptionName.MOUSE\_HOVER\_TIME, 5.0)

This ensures Rational Functional Tester hovers over the necessary region of your application and waits five seconds for the mouse-over menu to display. Using setOption enables you to tweak small pieces of your script without having to use the global playback options.

Finally, Rational Functional Tester’s API also provides a resetOption method. This enables you to turn off your overrides and default back to the values residing in the global playback settings. This is useful for optimizing the behavior of your scripts. For instance, if you increase your mouse delays to handle a tricky drag-and-drop action, you might want to reset them so that you don’t slow down *every* mouse action in your script. Using resetOption is quite easy. You simply make a call to it, specifying the option you wish to reset. An example of this is shown in the following line of code:

resetOption(IOptionName.DELAY\_BEFORE\_MOUSE\_UP);

This result in your script defaulting back to the value set for this option in the global playback settings. The behavior of your script would be such that it would delay only for the necessary drag-and-drop action and return to normal mouse speeds for the duration of playback.

**Flow Control**

Recording your scripts enables you to capture the actions necessary for navigation and verification. There are times when you need to build flow control into your scripts to handle errors and decisions. The latter are typically brought about by the need for things such as role-based testing, negative testing, and testing for Graphical User Interface (GUI) standards. The prior are instances that are neither predictable nor controllable.

Loops and conditional logic enable you to adjust the flow of your scripts based off things such as decisions you need your scripts to make, data you need your scripts to read, role-based scenarios you need to test for, error handling you need to flush out, and so on. With some clever recording and basic programming, you can modify your scripts so that they can handle these situations. This also helps reduce the overall number of scripts you have to maintain.

The basic idea of error handling is to recover from issues that arise during playback, or at least terminate with grace! Rational Functional Tester throws exceptions when it encounters errors. These are typically what drive the creation and continued growth of your error handling. Whether it is continuing script execution after playback or simply terminating it with detailed logging, handling errors is a key capability your scripts should have.

This chapter addresses flow control. It helps to answer the questions surrounding decision and error handling in your scripts. By the end of the next section, you should have a good feel for how to develop your script so it’s adaptable to different situations that might arise in your project.

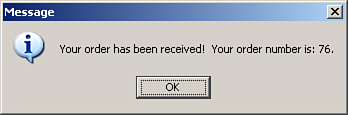
**Handling Positive and Negative Testing in the Same Script**

Developing a script that handles positive and negative testing is a multistep process. It involves multiple recordings and manual editing. It requires a dataset to work with that can include a database, external file, or native datapool. The general idea is that you record the basic flow through your application. You then amend your script with the necessary steps to test for and handle the alternate flows (for example, when you enter “bad” data).

The first thing you need to do is figure out where and when the error handlers are invoked in the application you are testing. This enables you to figure out how to design your script. After you have that done, record the base script. Essentially, this is a recording of the “happy day” scenario, using positive data. Finally, you need to set your application’s state to the point where the error handler is invoked. This is where you insert a recording into your script that captures the steps needed to deal with the handler. Depending on the number of error handlers you try to capture for your script, you might need to iterate this process the appropriate number of times.

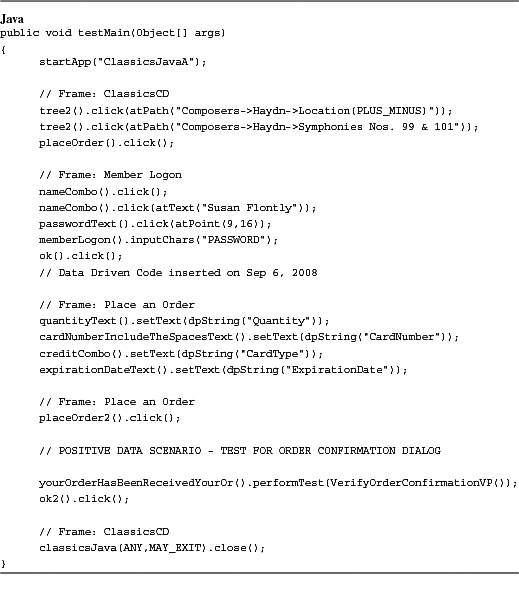
To exemplify this, you can use the ClassicsJavaA application that comes pre-configured with Rational Functional Tester. When looking at testing the process of ordering a single CD, note that two error handlers can be invoked. One is engaged when the quantity field is not populated. The other is engaged when the credit card number or expiration date is left empty. When you record the “happy day” scenario, you look to test to make sure that the order confirmation message dialog box displays after placing your order (see [Figure 7.3](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07fig03)). You then perform a secondary verification to make sure the order confirmation message is correct.

**Figure 7.3** Order Confirmation dialog box



[Listing 7.2](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex02) shows what the recorded “happy day” scenario would look like in Java and VB.NET when connected to a datapool for quantity and credit card information.

**Listing 7.2** Happy day scenario for ClassicsJavaA





Referring to the basics steps outlined in the second paragraph, the next thing you do is set ClassicsJavaA to the state where the quantity error handler is invoked. Please refer to the error dialog box in [Figure 7.4](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07fig04).

**Figure 7.4** Error handler #1—missing quantity value

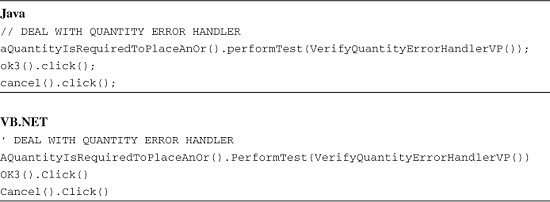


After performing that step, you need to locate the place in your script where this error dialog box is launched. In this case, it is the following line:

placeOrder2().click();

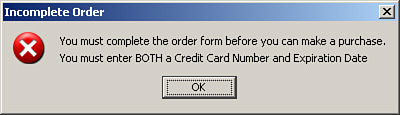
Upon playback, your script executes this line and places the order into the database. If the quantity field is empty, ClassicsJavaA pops open the error handler (shown in [Figure 7.4](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07fig04)). You need to insert a recording after this line in your script. This is where you capture the steps for handling this dialog box. In this case, you record a verification point (VP) to ensure the proper message is displayed in the error dialog. You also record the action to dismiss the handler (for example, clicking the **OK** button). [Listing 7.3](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex03) displays the result.

**Listing 7.3** Dealing with the quantity field error handler



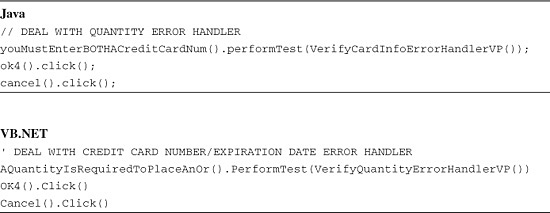
You need to iterate through this process again and deal with the second error handler. This is the one that is engaged when the credit card number or expiration date is left blank. You can see an example of this error dialog in [Figure 7.5](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07fig05).

**Figure 7.5** Error Handler #2—missing credit card number or expiration date



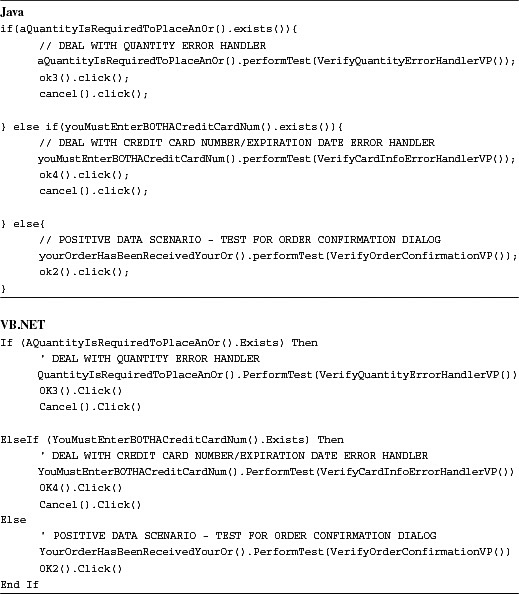
The steps for this are identical. You need to insert a recording that verifies the correct error message displays and handles getting rid of the error dialog. The code shown in [Listing 7.4](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex04) looks similar to the lines that dealt with the first error handler in your script.

**Listing 7.4** Dealing with the Credit Card Number/Expiration Data Field error handler



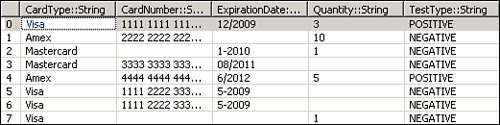
After the necessary pieces are recorded for the three scenarios, you need to add the flow control. This requires conditional branching. The basic logic is something like: “If the quantity error handler was launched, handle it. If the card information error handler was launched, handle that. Otherwise, just deal with the normal scenario, testing for the order confirmation message.” Converting that to code, you get something similar to the two scripts in [Listing 7.5](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex05).

**Listing 7.5** Flow control



The last piece to complete this script is tying things together with the datapool. The datapool not only contains the data to enter into your application’s fields, but it also contains a column letting the script know if a row of data is a positive or negative test. Please refer to [Figure 7.6](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07fig06) for an example of a datapool that handles both positive and negative testing.

**Figure 7.6** Datapool for positive and negative testing



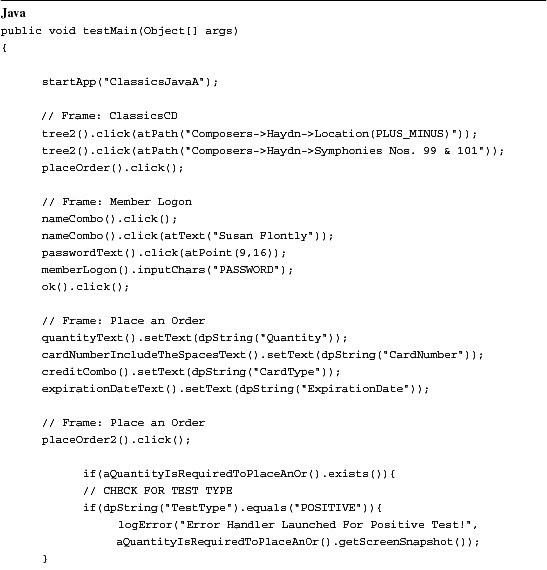
This column enables you to avoid false positives and negatives. For example, if an error handler is launched while testing a normal “happy day” scenario (for example, valid data is provided), you need to log the error and, at some point, submit a defect. This demands that you amend your script to check to see if it currently runs through a positive or negative test, providing for error logging, if necessary. [Listing 7.6](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex06) provides sample code snippets that show how to edit the quantity error handler to check for the test type (such as whether it is a positive or negative test).

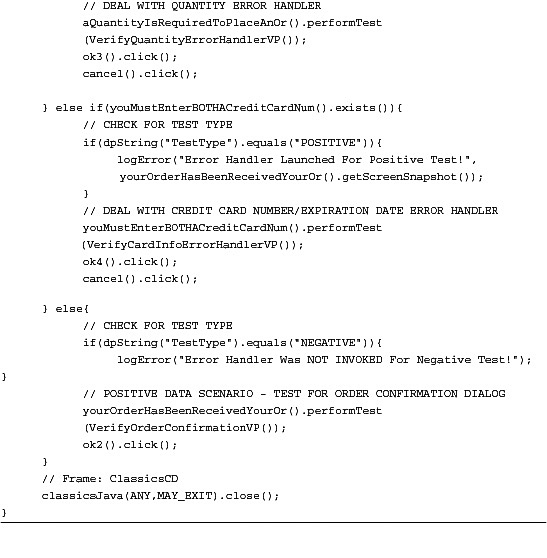
**Listing 7.6** Checking for a positive or negative test type

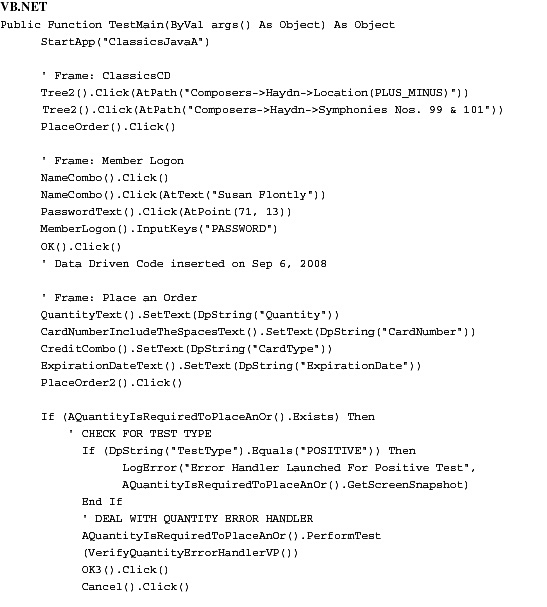


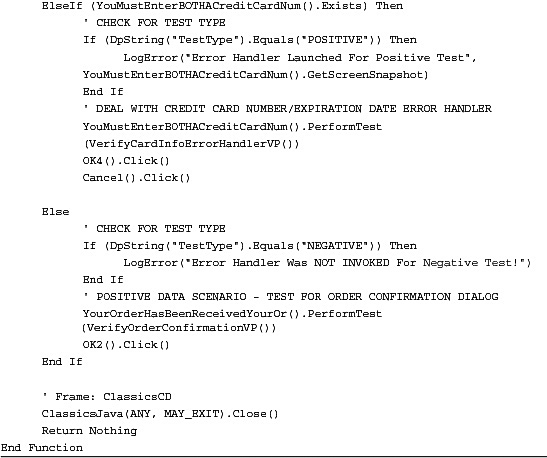
The basic idea here is that the script checks for the test type and, if necessary, logs the error message, complete with an image of the error handler that was incorrectly launched. You need to modify the other two pieces of your conditional branching (that is, the normal scenario and the credit card information error handler) in a similar manner. Again, the point is to avoid false positives and negatives. The result of your script editing looks like [Listing 7.7](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex07).

**Listing 7.7** Complete positive and negative testing script for ClassicsJavaA









**Localizing the Looping of a Datapool to a Certain Section of a Script**

Rational Functional Tester’s datapool capability is useful for testing sections of your application that require data entry. For instance, you can quickly iterate through all the possible positive and negative data combinations that are required to test user registration, user logon, order entry, and so on. It is easy to create datapools, especially when you import existing data. It is simple to connect a datapool to your scripts. However, as powerful and easy to use as datapools are, they do have one slight issue that can affect you. The lowest level of granularity they work at is the script level.

When you execute a script associated to a datapool, it plays back the specified number of iterations. It loops through the datapool and pulls out a row of data to use each time. It then uses that row’s individual values to plug into the application that you test. Each iteration through the datapool, in fact, starts the script from the beginning—every single time! In most cases, this isn’t an issue. When it does become an issue for people, there are simple ways around it. One of the best ways to deal with this is to use the datapool methods in a loop.

The easiest way to do this is to set up a while loop that iterates through the entire datapool. Calling dpDone() enables your loop to know when it is finished. The following line of code shows what this looks like in Java:

while(!dpDone())

If you use VB.NET, it mimics the following line:

While (Not (DpDone()))

To move to the next row, you simply call the dpNext() method. Lastly, to obtain the individual values of the current datapool row, you call the dpXXX() for the value’s data type. For instance, you call dpString() for string values, dpInt() for integer values, dpFloat() for floating point integers, and so on. You need to pass the datapool column index or column name to these methods. For instance, calling the dpString() method, using a column name, looks something like following:

dpString("ColumnName").

The VB.NET version would resemble the following:

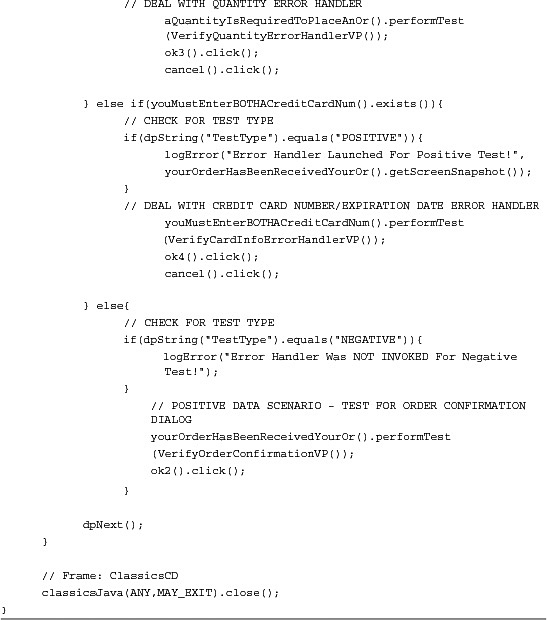
DpString("ColumnName").

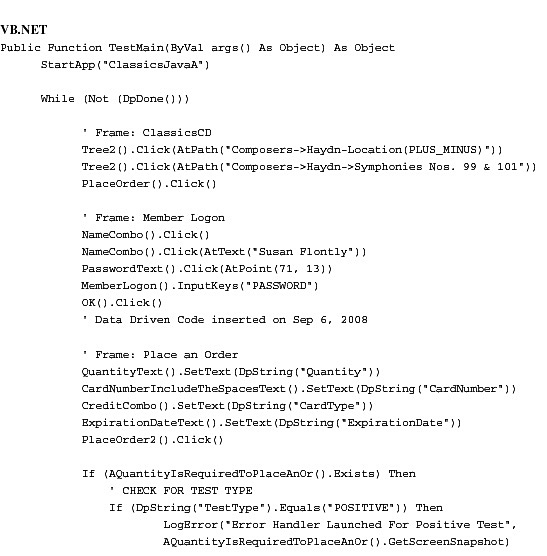
Using column indices is as simple as passing the column number to the method—dpString(1) / DpString(1) for Java and VB.NET, respectively.

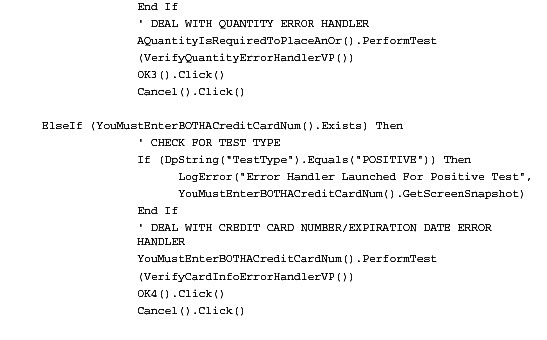
The previous section covered controlling the flow of your script with conditional logic. It showed how to use a single script to accomplish both positive and negative testing for ordering a CD. It was connected to a datapool containing both good and bad data. Playing it back results in your script iterating through the datapool, opening and closing the ClassicsJavaA application each time. If you strategically place a while loop into this script and make the necessary datapool method calls, you can avoid opening and closing the application for each iteration. The end result looks similar to the Java and VB.NET scripts found in [Listing 7.8](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex08).

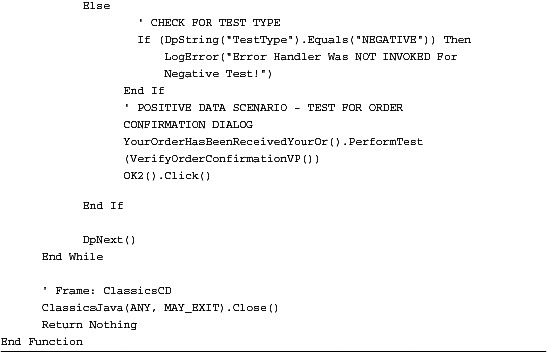
**Listing 7.8** Enhancing the “Error-Handling” script by looping through a datapool











The scripts in [Listing 7.8](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex08) adjust the flow control of the default datapool loop (for example, the internal loop that occurs by simply using a datapool). In fact, they actually override the looping mechanism of Rational Functional Tester’s datapool functionality. When you execute these scripts, they work with the while loops that you embedded into them. This is an important thing to note when you begin developing scripts that call other scripts (sometimes referred to as *shell* scripts). This is covered in the “[Developing Regression Scripts](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07sec1lev5)” section.

**Preventing Script Execution from Stopping If an Exception Is Thrown**

To quote a colleague, “Hope for the best; plan for the worst.” Unexpected events, despite how hard your work, almost certainly occur. In this section, you examine techniques that you can use to handle problems in three ways:

• Find out that a problem has occurred.

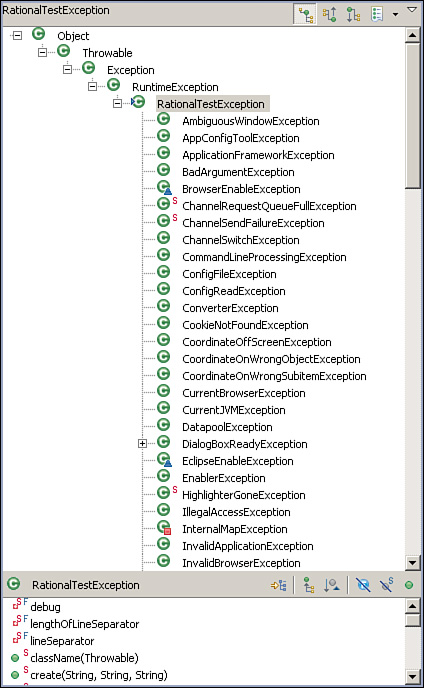
• Prevent, if you choose, script execution from suddenly coming to an end; meaning, enabling yourself the opportunity to clean up and end gracefully.

• Handle the problem and go back and try and repeat the thing that caused the problem.

As you probably know, without adding anything to a script, any exception thrown during script execution time causes the execution of your script (or suite of scripts) to come to an abrupt end. In the log, there is mention of an “unhandled exception.”

Various Rational Functional Tester-specific exceptions can occur at runtime. The root class of many of these problems is RationalTestException, which is inherited by many subclasses (see [Figure 7.7](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07fig07)).

**Figure 7.7** Test exception hierarchy



The (arguably) most commonly thrown exceptions are ObjectNotFoundException and AmbiguousRecognitionException (which, incidentally, inherits from ObjectNotFound Exception).

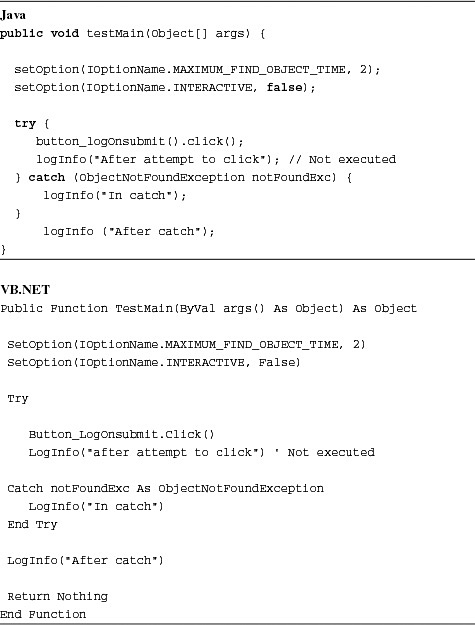
You can use two techniques (either one of the following or a combination of the two) to handle exceptions:

• Add a try/catch.

• Override RationalTestScript events that are automatically called by the playback engine when (or right before) an exception is thrown.

For readers who might be new to try/catch, we start off with an example that simply illustrates how a try/catch is used and the flow of control. In this contrived example, you have a single statement: a click on a button. The button is not present, though. To prevent Rational Functional Tester from waiting 20 seconds for the object (you know it’s not there) and from displaying the Exception dialog box, you add two calls to setOption(). Please refer to the examples in [Listing 7.9](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex09).

**Listing 7.9** Simple try/catch example



If you execute this script, you should notice a couple of things:

• Two of the three messages were written to the log. The statement immediately following the click() was not executed.

• There’s no failure in the log. The log makes it appear that the script executed successfully. In a sense, it did. Script execution halts when an *unhandled* exception is thrown. Catching the exception is considered handling it (the fact you didn’t handle it in a meaningful way is another story).

Next task: What would you do if you wanted your script to handle multiple exceptions, for example, a PropertyNotFoundException (which is thrown if you try to get a property that doesn’t exist from a test object) and an ObjectNotFoundException? Easy—use multiple catch blocks.

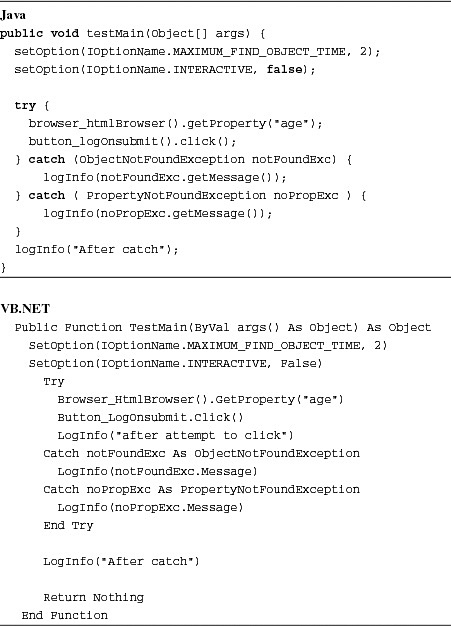
In the following example, you have two statements that could potentially cause an exception to be thrown:

• You attempt to get the age property from the browser (of course there is no such property).

• You try to click on a button that is not present.

[Listing 7.10](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex10) shows examples of using catch blocks to tell your script what to do when one of the two previously listed statements causes an exception to occur.

**Listing 7.10** Using catch blocks to handle exceptions



As in the previous example, the script completes successfully. In this example, you write a more interesting message to the log within the catch, you retrieve and write the exception message to the log, which in this case includes “Property age not found.” If the age property did exist, the second exception would have been caught for the absent Logon button, writing the appropriate message to the log.

Many types of exceptions can be thrown. If you wanted to catch *any* type of Rational Functional Tester runtime exception, you can be generic and just catch RationalTestScriptException, shown in [Listing 7.11](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex11).

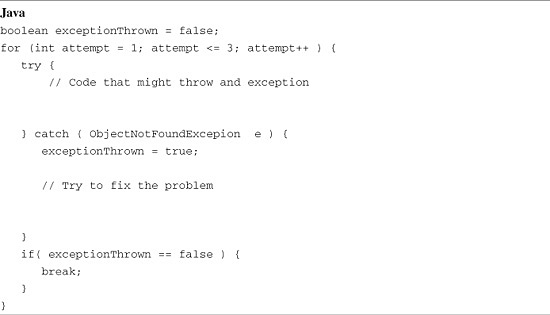
**Listing 7.11** Using the generic RationalTestScriptException to catch any exception



Wouldn’t it be helpful if you could not only catch the exception but also *fix* the problem? Of course, it would. You can fix the problem in the catch, but the challenge is then going back to the statement that caused the exception to be thrown. There are a couple of ways to do this:

• Put a loop around the try/catch block. The loop enables you to attempt to execute a statement (or series of statements) several times. You end up with something like the pseudo-code in [Listing 7.12](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex12).

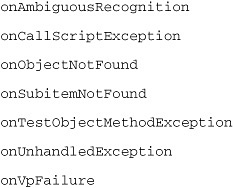
**Listing 7.12** Placing a loop around a try/catch block



One issue with this approach is that you might end up with many blocks of code like this.

• Don’t use a try/catch at all; instead, override a RationalTestScript event.

You can override seven exception-related events. Your scripts never call these events. These are:



These events are defined in the RationalTestScript class. The implementation defined in RationalTestScript is to write a message to the log and to allow execution to come to a halt. You have the option of overriding this behavior. You can override any RationalTestScript even in a test script or in a HelperSuper class.

Next, you look at onObjectNotFound. The first thing you should know about these events is that, unlike try/catch blocks, unless you *do something* in your event-handling code, the exception is thrown again after your own code is executed and execution is abruptly stopped.

To illustrate this point, override the onObjectNotFound event. If you’re using Eclipse, right-click anywhere in your script editor and select **Source** > **Override/Implement Methods**. Then, in the dialog that displays, under RationalTestScript, select **onObjectNotFound** and click **OK**. Eclipse inserts an event stub into your script. In Visual Studio, you need to type the code.

As in the first example, you have a script with a single statement—a click on an object that you purposefully make sure isn’t present when you execute. When Rational Functional Tester fails to find the test object, execution automatically leaves testMain() and goes to the onObjectNotFound event, in this case to the overridden onObjectNotFound event. The first thing you do is put a single statement in the event—one that writes to the log—just to prove that the code is being executed and just having the event code doesn’t prevent the exception from being thrown. Refer to the code samples in [Listing 7.13](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex13) for the respective Java and .NET implementations of this proposed solution.

**Listing 7.13** Example of overriding the onObjectNotFound event



If you execute either of the scripts in [Listing 7.13](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex13), you see your log message and a failure in the log. You do not see the “after attempt to click” log message. Playback comes to a halt.

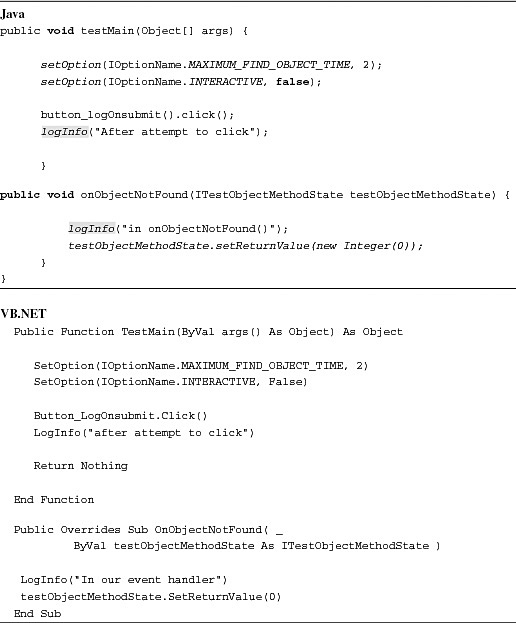
How do you prevent playback from bombing? The Rational Functional Tester API documentation for onObjectNotFound has the following statement: “If this event is not handled and the testObjectMethodState is not modified, an ObjectNotFoundException is thrown.” So *two* things must occur to prevent the exception from being thrown:

**1.** Override the event (which you’re doing).

**2.** Modify the testObjectMethodState.

So how do you modify the testObjectMethodState? If you look up ITestObjectMethodState in the API, you see (among others) three methods: findObjectAgain(), setFoundTestObject(), and setReturnValue(). Calling any of these methods prevents the exception from being thrown. Use setReturnValue(). Setting any return value other than a null (nothing in VB.NET) is sufficient to prevent the exception from being thrown. [Listing 7.14](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex14) shows how to implement this in Java and VB.NET.

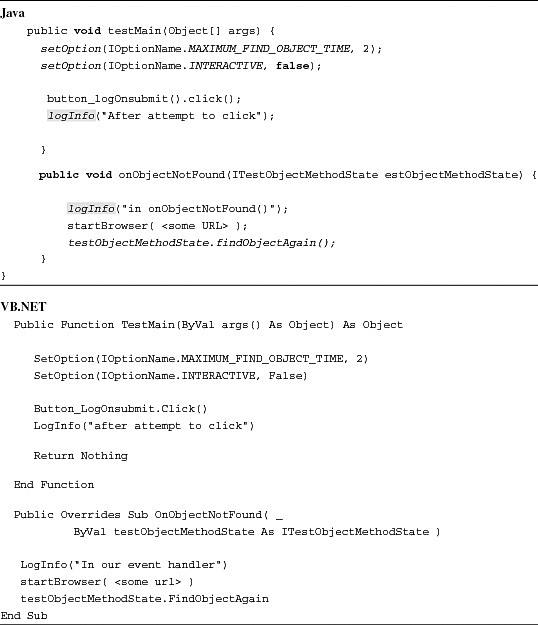
**Listing 7.14** Preventing exceptions from being thrown, terminating script execution



If you execute, you can see that not only did script playback not come to an end, but after your event was fired, execution flow went back to testMain() and resumed with the next statement. In many cases, continuing execution after an exception has occurred might not be reasonable. What’s neat is that Rational Functional Tester’s execution apparatus makes that possible for you if you need it.

Now for the big finish: Try to fix the problem and instruct Rational Functional Tester to execute the method call that caused this situation. Although powerful, this can be challenging to implement in real life. To fix the problem, the event has to somehow know (or figure out) what should be done to right the wrong that’s taken place. A simple example follows. Assume, for the purposes of this example, that if the button is not present, this indicates that something caused the application process to die, for example, you’ve probably seen Internet Explorer suddenly disappear. In this case, to right the wrong, all you need to do is launch the browser and navigate to a particular URL. That’s a single statement, startBrowser(). Directing Rational Functional Tester to execute the code that caused the event handler to fire is also a single method call: You call findObjectAgain() on the ITestObjectMethodState argument to the event. This is illustrated in [Listing 7.15](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex15).

**Listing 7.15** Fixing the problem in the onObjectNotFound event handler



**Adding a Screenshot to the Log If a Verification Point Fails**

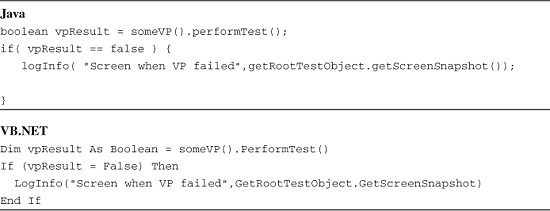
Sometimes it can be a challenge to convince others that an application is exhibiting defective behavior. Pictures can sometimes help you make a stronger case.

Rational Functional Tester makes adding screenshots to the log trivial. You can capture a screenshot by invoking getScreenSnapshot() on the RootTestObject. The image is returned as a BufferedImage (System.Drawing.Bitmap in Visual Basic.NET)

BufferedImage image = getRootTestObject().getScreenSnapshot();

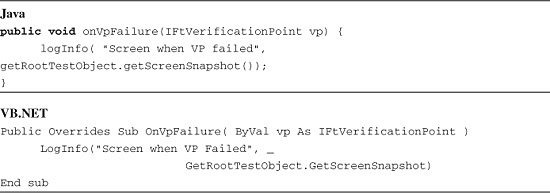
In addition, each of the logging methods—logInfo(), logWarning(), and logError()—is overloaded. The two-argument version takes a String and a BufferedImage. What if you want a screenshot added to the log if a verification point (or all verification points) fails? Well, the long way to achieve this is to look at the return value of verification point (performTest() returns a Boolean). This is shown in [Listing 7.16](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex16).

**Listing 7.16** Checking for a verification point failure before logging a screenshot



To prevent the need to check the return value of each verification point, override onVpFailure (shown in [Listing 7.17](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex17)).

**Listing 7.17** Overriding onVPFailure to log a screenshot



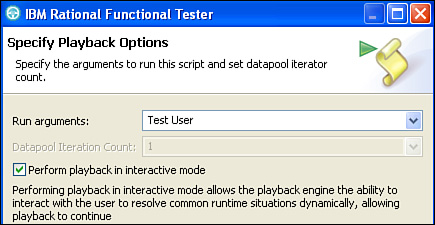
For this event to be applied to every script, put it in a Helper Superclass (see [Chapter 3](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch03.html#ch03), “[General Script Enhancements](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch03.html#ch03)”).

**Passing Arguments to a Script from the Rational Functional Tester User Interface**

Rational Functional Tester enables you to set playback options prior to executing a script. One of these options is the ability to pass arguments to your script upon playback. To do this, you need to follow two steps: specifying the arguments and handling those arguments in the scripts.

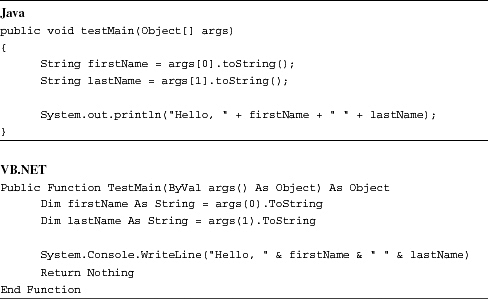
Passing arguments to your scripts is similar to passing arguments to UNIX/Linux shell scripts or Windows batch files. You simply provide the desired arguments, separating them by spaces. [Figure 7.8](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07fig08) illustrates an example of this.

**Figure 7.8** Specify playback options—Run Arguments



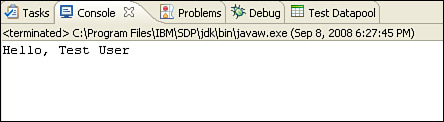
[Figure 7.8](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07fig08) shows Test and User as the two arguments being passed to the script. Upon playback, these are passed into your script in an object array. You need to edit your script so it can handle these arguments. [Listing 7.18](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex18) shos a simple example of handling these arguments.

**Listing 7.18** Handling arguments passed to the script from the Rational Functional Tester playback options



You need to parse the object array. As shown in [Listing 7.18](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex18), the first cell of the array contains the first name—either arg[0] when using Java or arg(0) when using VB.NET. The second cell of the array holds the last name, arg[1] or arg(1). The net result of these scripts is a Hello message displayed in the console window ([Figure 7.9](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07fig09) for Eclipse and [Figure 7.10](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07fig10) for .NET).

**Figure 7.9** Script output in console window—Java



**Figure 7.10** Script output in output window—VB.NET



This is an oversimplified example. However, the concept remains the same for much more complex scripting. For example, you might wish to pass in the specific web server or database server to test against on a given day. You simply specify the server information in the playback options and ensure that you have the code to handle them in your script, capturing the data from the object array that contains your argument(s). In other words, you do the same thing exemplified in the scripts found in [Listing 7.18](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex18), substituting a first and last name for a web server and database server.

**Passing Arguments to Scripts Invoked from Other Scripts**

The previous topic discussed how you pass arguments into your scripts from Rational Functional Tester’s playback options. This is a useful technique for dealing with such dynamics as testing against different web servers, database servers, and so on. What about the scenario of dealing with dynamics between scripts? How do you develop your scripts to manage consistency throughout your application? Rational Functional Tester provides for this by passing arguments to other scripts, using the callScript(RationalTestScript script, Object[] args) method. This enables you to invoke another script from within the currently executing script, passing information to it in an Object array. The VB.NET version of this is CallScript(scriptFullName As String, args() As Object).

Imagine you want to test for the following scenario, using the ClassicsJavaA application:

**1.** Launch the ClassicsJavaA application.

**2.** Order a CD.

**3.** Verify the order is successfully placed.

**4.** Verify the order exists in the order history portion of the application.

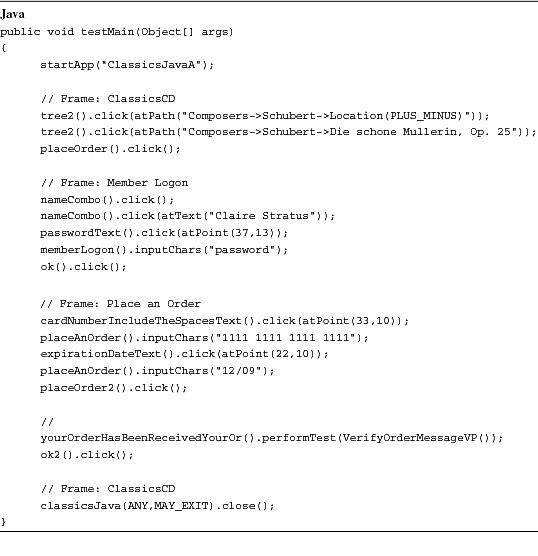
**5.** Cancel the order.

**6.** Close the ClassicsJava application.

This can be accomplished in a single script. In fact, if this were a real automation project, you would probably already have a script that handles the testing of placing an order in this application. What happens if another member on your team already created a script to handle the testing of reviewing that an order exists in the ClassicsJavaA order history section? Would you rerecord that functionality into your existing script? Ideally, you would not. You would reuse the piece that already exists, making the necessary modifications to make the two scripts work together.

The two scripts in [Listing 7.19](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex19) show the recording of launching ClassicsJavaA, ordering a CD, verifying the order was placed (using the order message), and closing the application.

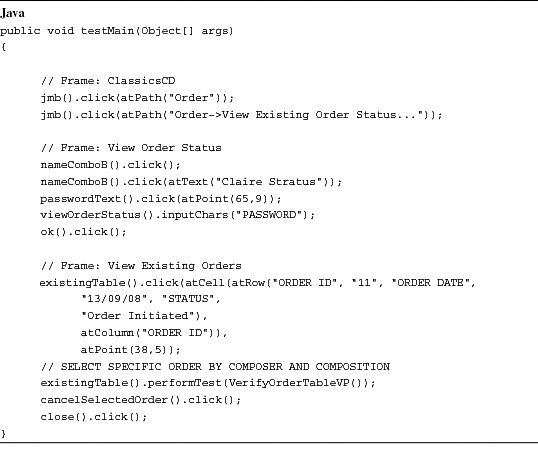
**Listing 7.19** PlaceCDOrder

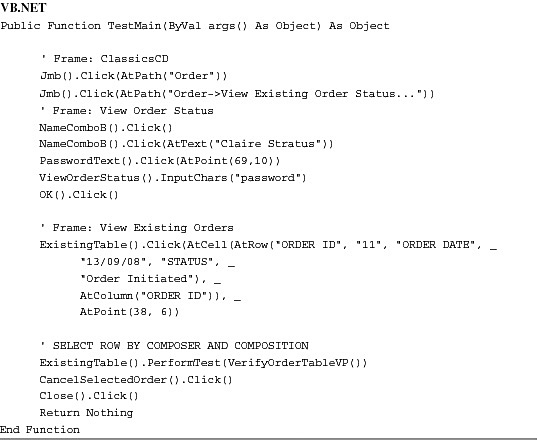




The next set of scripts (located in [Listing 7.20](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex20)) shows what a script might look like for reviewing the order history in ClassicsJavaA. The scripts simply access the order history, select the desired order row in the history table, verify it, cancel the order, and close out the order history window (see [Listing 7.20](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex20)).

**Listing 7.20** ReviewOrderHistory





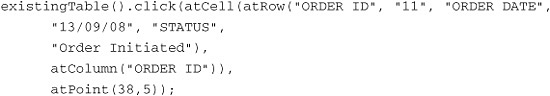
Mentioned previously, the ReviewOrderHistory script might be something that a team member already recorded. Working together, you and your teammate can make simple modifications to this script and enable it to communicate with and pass data to your PlaceCDOrder script. Only three things are needed:

**1.** Modify the "PlaceCDOrder" script to capture the necessary data for properly selecting the correct row in the Order History window.

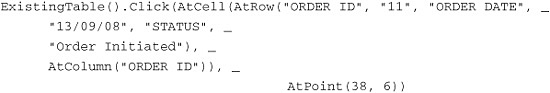
**2.** Modify the "PlaceCDOrder" script to invoke the "ReviewOrderHistory" scripting, passing this information gathered in step #1.

**3.** Modify the "ReviewOrderHistory" script to use the data that was captured in the "PlaceCDOrder" script (see step #1).

To accomplish step #1, you need to investigate the "ReviewOrderHistory" script. This lets you know what you need to capture in the "PlaceCDOrder" script. You are looking for the line where the desired order row is selected. It looks like the following code snippet in Java:

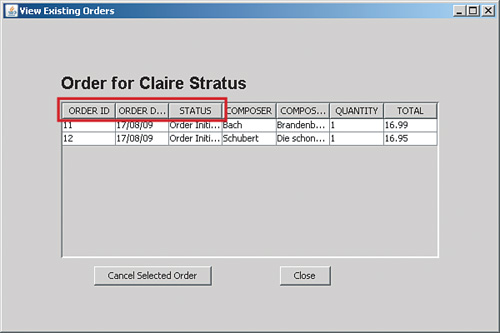


VB.NET looks like the following:



Rational Functional Tester clicks on the row specified by a particular ORDER ID, ORDER DATE, and STATUS (11, 13/09/08, and Order Initiated, respectively). Looking at [Figure 7.11](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07fig11), you can see that these are the table columns.

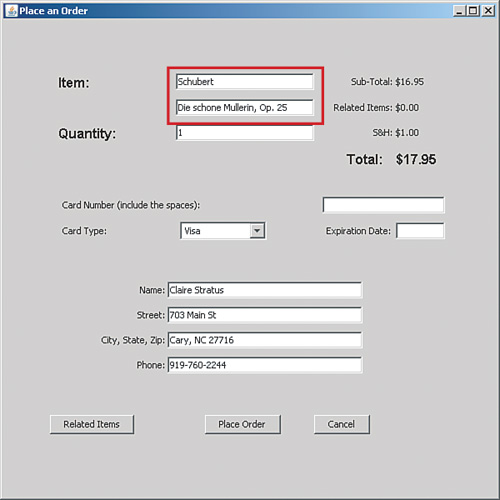
**Figure 7.11** View existing orders window



Basically, Rational Functional Tester identifies a row by certain columns. In step #1, you need to figure out which of these columns is the best to use for identifying your order. For the purpose of this example, you use the COMPOSER and COMPOSITION columns. You should take note that you need to modify the previous line of code for the third step of this process.

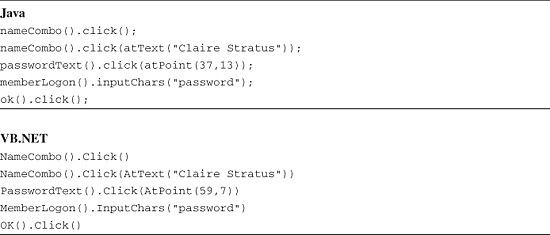
With the information you garner from the "ReviewOrderHistory" script, you can easily accomplish the first step by inserting a recording into your "PlaceCDOrder" using Rational Functional Tester’s Verification Point and Action Wizard. [Chapter 1](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch01.html#ch01), “[Overview of Rational Functional Tester](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch01.html#ch01),” discussed how you can use this wizard’s **Get a Specific Property Value** option to capture a property value for a particular object on your application. Before inserting the recording for this, you need to first open the ClassicsJavaA application to the point where the *Place an Order* window is open, displaying the composer and composition data (see [Figure 7.12](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07fig12)).

**Figure 7.12** Place an Order window



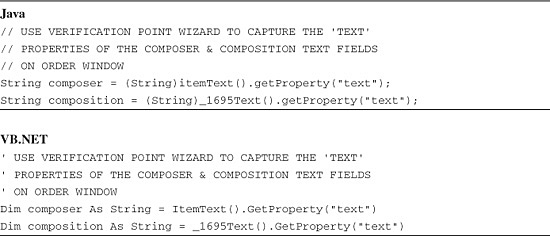
You also need to find the proper place to put the newly recorded lines of code in your script. You can do this at any point in your script that deals with the Place an Order window. For this example, you can insert the recording after the lines in your script that log into ClassicsJavaA as Claire Stratus. Please see [Listing 7.21](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex21).

**Listing 7.21** Login steps found in script snippet for ClassicsJavaA



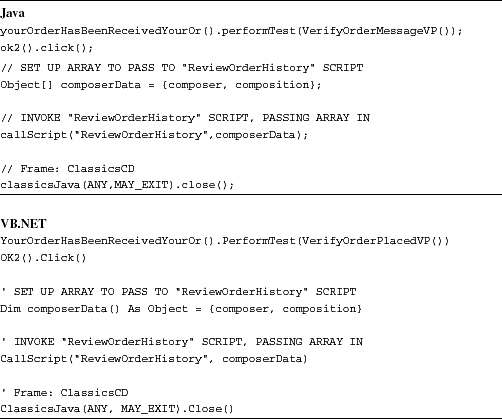
After you place your cursor after the last line in login steps (for example, ok().click()), you can insert a recording into your script. You want to engage the Verification Point and Action Wizard, pointing the Object Finder Tool at the composer text field, residing on the Place an Order window. You need to select the **Get a Specific Property Value** option, specifying the *text* property. This grabs the value that is currently in the composer text field. You need to repeat this process against the composition text field. After you finish recording these two steps, ideally supplying descriptive comments for each one, you see something similar to the code in [Listing 7.22](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex22).

**Listing 7.22** Code to acquire composer and composition text field data



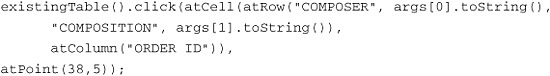
The second step you need to do is provide the code that invokes the ReviewOrderHistory script, passing in the composer and composition data. The opening paragraph in this section discussed using the callScript(RationalTestScript script, Object[] args) method and CallScript(scriptFullName As String, args() As Object) function. They are simple to use. You need to create a couple of lines of code to create the object array for storing the composer and composition data in it. You then call this method, passing in the name of the script you want, ReviewOrderHistory in this case and the array you created. Much like inserting a recording into your script, you need to properly locate this code. If you look at the flow of your PlaceCDOrder script, you note that the last two steps verify and close the order confirmation message and then close the ClassicsJavaA application. You want to place your code in between these sections. The final result looks like the code in [Listing 7.23](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex23).

**Listing 7.23** callScript(RationalTestScript script, Object[] args)

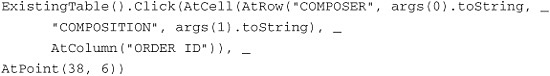


The examples in [Listing 7.23](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex23) show that your PlaceCDOrder script now invokes the ReviewOrderHistory script, passing it the composer name and composition as an array. Remember, you captured the composer and composition values by following the first step in this process.

The third and final step of this process is to modify the ReviewOrderHistory script so that it can use the composer information in a meaningful way. Remember the mental note that you were supposed to make in Step 1? You were supposed to remember that you need to modify the line in your script that makes the order row selection, substituting the different column names and values. Doing so results in the following line in your Java script:



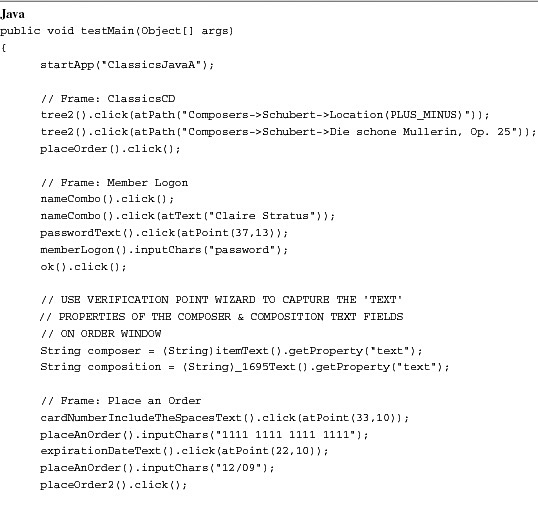
... and in VB.NET:

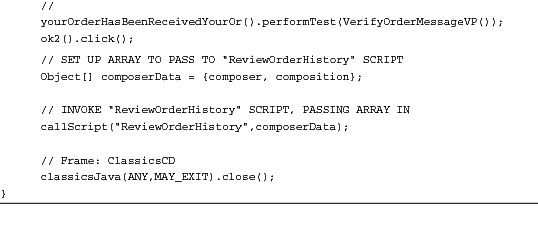


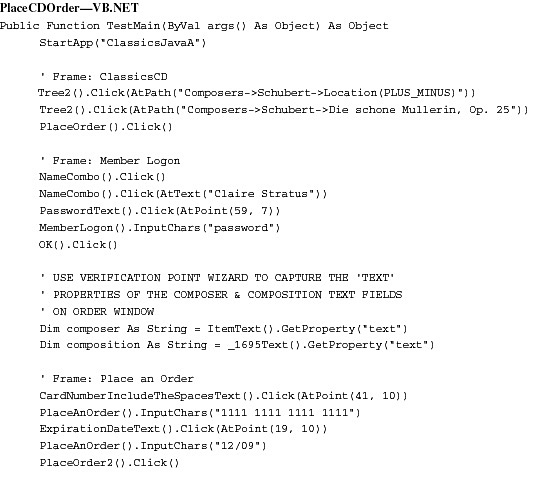
The previous lines of script—Java and VB.NET respectively—reference the 0 and 1 cells of the args array. The 0 cell represents the composer text, whereas the 1 cell represents the composition text. This is because you set the array up in the PlaceCDOrder script in that order. It is important to keep track of how data gets stored in one script so you know the order in which to access it in another.

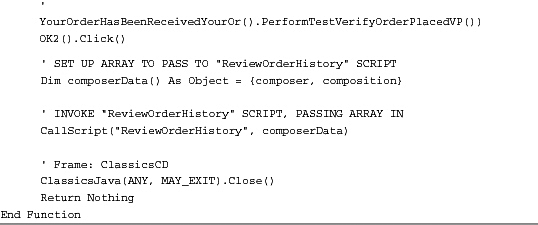
The steps covered up to this point take you through passing data between scripts. [Listing 7.24](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex24) shows the final product of PlaceCDOrder, incorporating the necessary changes for invoking the ReviewOrderHistory script.

**Listing 7.24** PlaceCDOrder—the final product



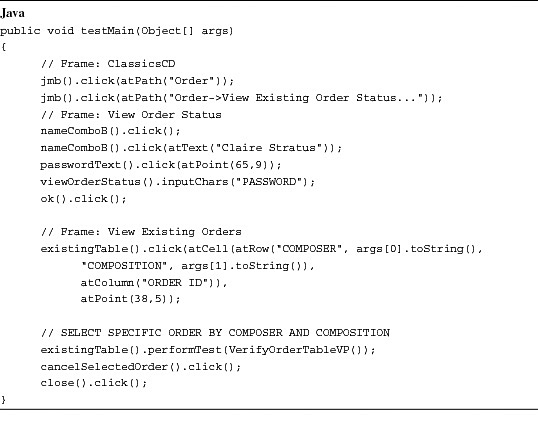


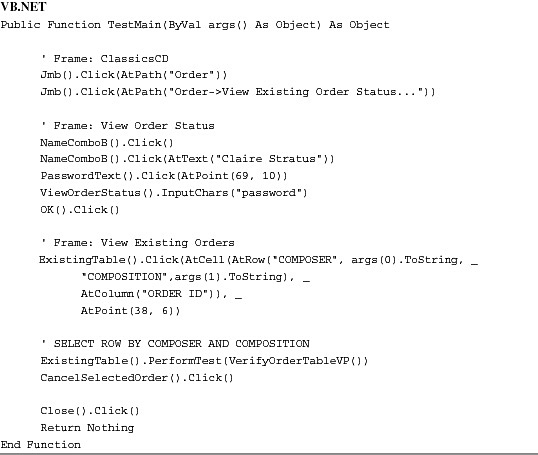




[Listing 7.25](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex25) displays the updated contents of the ReviewOrderHistory script, including the modification that was made to parse and use the data passed to it via its args array.

**Listing 7.25** ReviewOrderHistory





The next section, “[Returning Values from My Scripts](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07sec1lev4),” covers returning data from scripts. This is the last section you need to understand before learning about regression scripts. The section, “[Developing Regression Scripts](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07sec1lev5),” ties this topic with the next one into a nice neat package. You see how to create one script that simply invokes other scripts and catches return values from some and passes those as arguments to others.

**Returning Values from My Scripts**

Rational Functional Tester provides you with the valuable ability to return values from your scripts. Returning values from your scripts enables you to share data across scripts, without the need for external files or global variables. You need to do a few basic things to return data from your scripts.

The first step is to specify the data type you want your script to return. The return type that you choose depends on what you want to share with another script. You need to remember that scripts catch return values as objects. Therefore, you want to specify an object or a data type based off of an object (for example, integer, string, and so on). Do you simply want to pass a 1 or 0 back to signify that your script did (or didn’t do) what you wanted? You want to pass it back as an integer. Do you need to pass along an Order ID? You might try passing it back as a string. Ultimately, you need to orchestrate how you want your scripts to run together. This enables you to figure out the return values you need for sharing data among your scripts. In any case, you need to modify the basic script’s testMain() method signature to specify the return value.

Whenever you create a script (either an empty one or a recorded one), it has a testMain() method. This is the method that drives the execution of Rational Functional Tester scripts. The following lines of code show what the Java and VB.NET signatures look like for this testMain().

**Script Signature—Java**

public void testMain(Object[] args)

**Script Signature—VB.NET**

Public Function TestMain(ByVal args() As Object) As Object

By default, the Java version of this method returns nothing (for example, void), whereas the VB.NET version returns an object. To change the return value of testMain(), you simply need to supplant the default value with the Object type that you need. Examples follow:

**Script Signature—Java**

public String testMain(Object[] args)

**Script Signature—VB.NET**

Public Function TestMain(ByVal args() As Object) As String

Looking at the signatures in the previous lines of code, you can see that the testMain() method now returns a string. What is the string that it needs to return? This is up to you, which brings us to the second step.

After you specify the return type in your script’s testMain() signature, you need to modify the script to return the corresponding data value. This is a two-part process. The first part requires you to capture the data in the body of your script. The second part is where you construct the return statement. The return statement is as simple as typing “return” and specifying the data to return.

**Return Statement—Java**

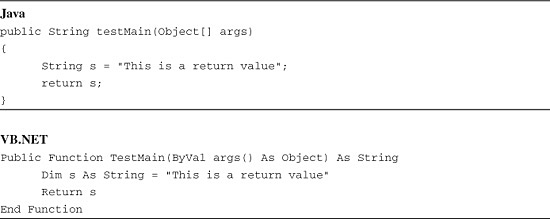
return "This is a return value";

**Return Statement—VB.NET**

Return "This is a return value"

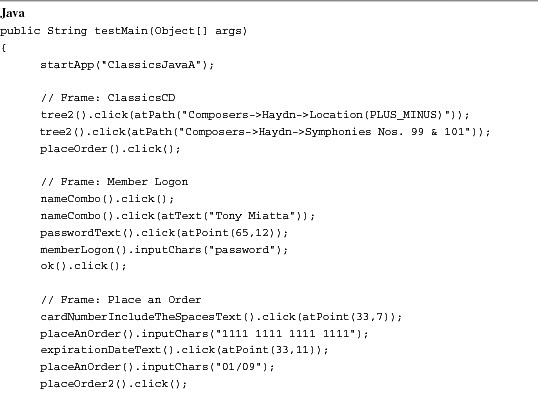
The prior statements return a string value. You can also pass a variable to the return statement. Looking at the coding examples in [Listing 7.26](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex26), you can see how all these steps come together, specifying the new return value in the testMain() signature, capturing the desired data and constructing the proper return statement.

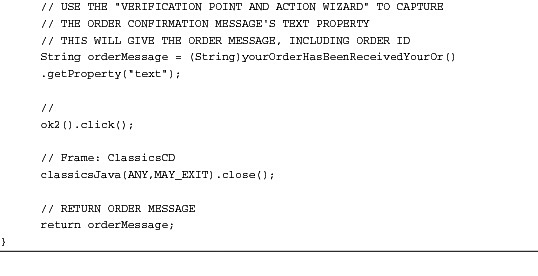
**Listing 7.26** Simple return example

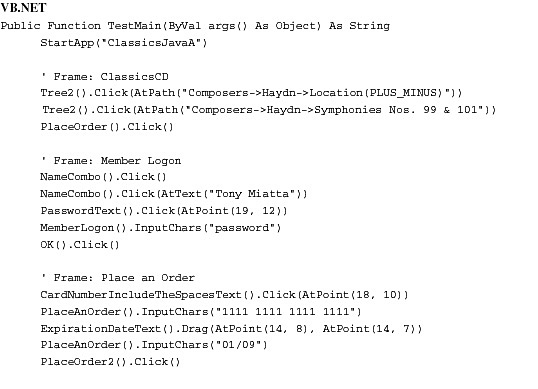


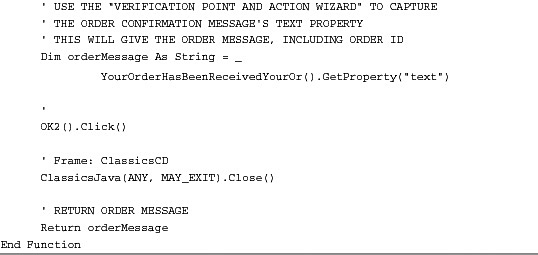
These are oversimplified examples. They basically create a string and return it. The next set of examples provides you with context. They modify a recorded script to return an order message. This script places an order for a CD, using ClassicsJavaA. Please refer to [Listing 7.27](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex27).

**Listing 7.27** Returning data from a recorded script









The next few sections discuss different scenarios for shell scripts. These are scripts that primarily just call other scripts. You examine how you combine the past two topics—passing arguments and returning data—and create robust shell scripts for regression testing.

**Developing Regression Scripts**

Rational Functional Tester enables you to create scripts whose job is simply to invoke other scripts. These are sometimes referred to as *shell scripts*. This section refers to them simply as *regression scripts*. The ultimate goal of the scripts is to cover the testing of an application or each of its subsystems.

The next couple of sections show basic to advanced usage of this capability. You gain an understanding of the different techniques for passing data between scripts, dynamically adding calls to newly created scripts, and managing the flow of your regression scripts.

**Creating a Simple Regression Script**

The previous section mentioned that a regression script is one that simply invokes other scripts. This is done using the callScript() methods. You were exposed to a flavor of this method in the “[Passing Arguments to Scripts Invoked from Other Scripts](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07sec2lev6)” section in this chapter. In reality, many ways to create regression scripts exist. They range from simplistic implementations that simply call other scripts to extremely robust implementations that manage the handling errors within scripts and passing data between them.

The most simplistic regression script is one that uses the callScript() method without any arguments. Its purpose is to invoke the necessary scripts and let them do what they were created to do. It neither passes arguments to the scripts it invokes nor catches return values from them. If you look at the scenario of placing an order in the ClassicsJavaA application, it might look something like this:

**1.** Launch ClassicsJavaA.

**2.** Place an order for a CD.

**3.** Review that the order exists in the database.

**4.** Close ClassicsJavaA.

You can almost envision having a one-to-one mapping of scripts to steps. That is, you have a script that launches the application (StartClassicsJavaA), one that orders the CD (PlaceCDOrder), one that reviews the order in the database (ReviewOrderHistory), and one that closes the application (CloseClassicsJavaA). To create a regression script that invokes these other scripts, you need to:

**1.** Create an empty script (**File** > **New** > **Empty Functional Test Script**). Note that for Rational Functional Tester VB.NET, you select **File > New > Add Empty Script**.

**2.** Place the cursor in the body of your newly created script.

**3.** Right-click on the script you want to invoke (the “StartClassicsJavaA” script would be good one to start with!) and select **Insert as “callScript.”**

**4.** Repeat Steps 3 and 4 as many times as necessary to include your scripts (in this case, it is three more times, calling the remaining three scripts).

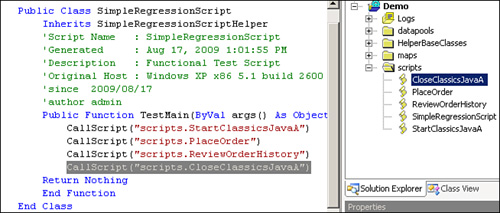
[Figure 7.13](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07fig13) provides you with a visual representation of this process using the Eclipse version of Rational Functional Tester.

**Figure 7.13** Creating a simple regression test—Java



[Figure 7.14](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07fig14) shows what these steps look like in the VB.NET flavor of Rational Functional Tester.

**Figure 7.14** Creating a simple regression test—VB.NET



One important factor that you need to consider is the ordering of the scripts. In the prior example, you had four scripts that accomplished the scenario of ordering a CD in the ClassicsJavaA application. Each script carries out a small task, relying on other scripts to successfully get the application to the necessary state for it to work with. This is considered modular scripting. This type of scripting helps to keep your scripts small and focused and eases the cost of script maintenance. It does require you to ensure that your regression scripts call these modular scripts in the proper sequence. If you do not, you will soon find that your regression scripts error out. Refer to [Figures 7.13](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07fig13) and [7.14](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07fig14) and note that the sequence has been ordered correctly. If you don’t want to worry about setting up the proper sequence in a regression script, you can create your other scripts so that they are self-reliant. In other words, you create your scripts so that they get the application into the state it needs to be in and then carry out its designated task. They don’t have to rely on another script to do this. Some people refer to this as creating *round-trip scripts*. If you wish to pursue this path, there is a useful strategy for creating regression scripts to support this. This is covered in the “[Creating Regression Scripts That Dynamically Include Scripts As They Are Created](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07sec2lev9)” section.

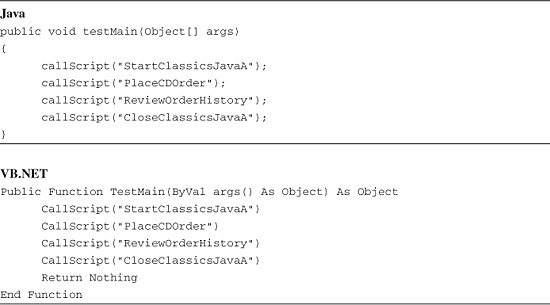
You should now know how to create simple regression scripts. The process is simple. In fact, many people skip the GUI-supported means of calling scripts. They simply create the empty scripts, type the first **callScript(“SomeScript”)** command, and then copy and paste it as many times as they need, replacing the old script name with the new one. The next section covers how to take advantage of catching return values and passing arguments in your regression scripts. These were discussed earlier in the “[Returning Values from My Scripts](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07sec1lev4)” section. They are discussed in more detail in the context of managing data between your scripts.

**Managing Data in Regression Scripts**

The previous section showed you how to create basic regression scripts. These can be useful. Chances are, however, that you need to share data among your scripts. This section expands on your knowledge of passing arguments to scripts, sending them back and catching return values from scripts, and developing regression scripts.

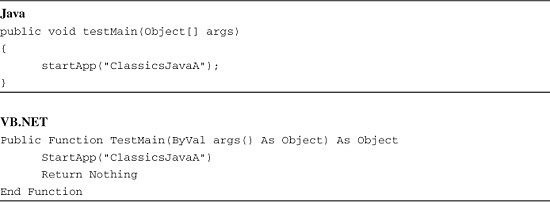
The example in the previous section covered the simple scenario of placing an order for a CD using the ClassicsJavaA application. The resulting regression script that was built looks like the scripts in [Listing 7.28](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex28).

**Listing 7.28** Simple regression script



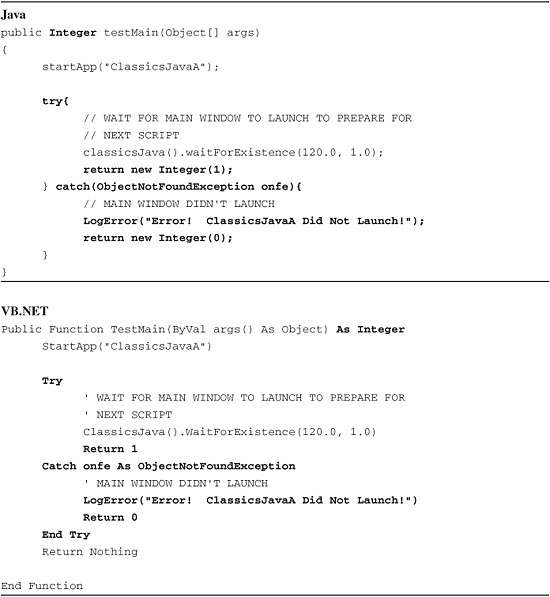
What if StartClassicsJavaA encounters an error that didn’t enable it to complete execution? You might want your regression script to terminate playback, which prevents other scripts from trying to launch. You want StartClassicsJavaA to return a success/failure flag and your regression script to catch it. This is a key example of how to control the flow and manage data in your regression scripts. StartClassicsJavaA should be a straightforward script. Its only purpose is to launch the application. Unless you add a verification point to check that the application launched, it should ultimately be one line. [Listing 7.29](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex29) shows the basic version of the StartClassicsJavaA script.

**Listing 7.29** StartClassicsJavaA



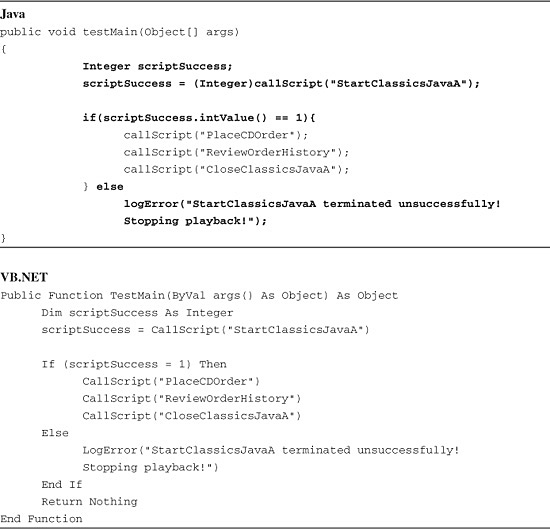
With some clever scripting, you can enhance your scripts so that they can deal with errors that occur during playback. For instance, you can add a small amount of code to the StartClassicsJavaA script that checks to see if the window exists or the process was started, returning an integer that specifies whether it executed successfully or not. You can also add log messages to your script, citing any errors. This is shown in [Listing 7.30](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex30).

**Listing 7.30** Simple Regression Script Enhancement #1, Return Success Value



The net result is that you now have a script that attempts to launch the ClassicsJavaA application. It waits for the main window to open. If it opens in two minutes (120 seconds), then it completes successfully, returning the integer “1.” If the main window doesn’t open within two minutes, the script assumes the application encountered an error trying to launch and writes an error message to the log file. It also returns the Integer “0.” You need to modify your shell script so that it catches the integer returned from the StartClassicsJavaA script and allows it to control the remaining flow based off of it. Please refer to [Listing 7.31](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex31) for an example of this scripting.

**Listing 7.31** Simple regression script enhancement #2, catch return value

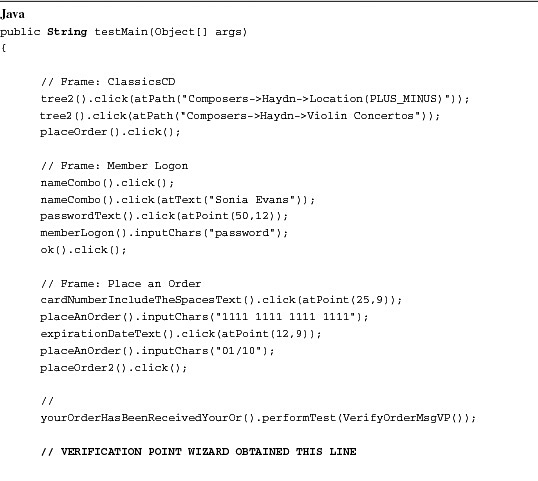


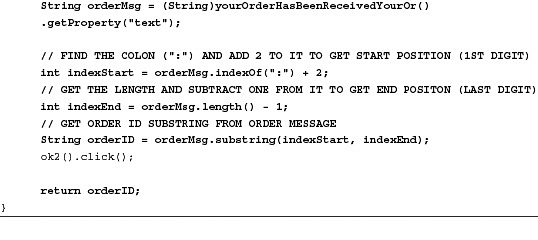
The regression script is now able to handle an issue with the StartClassicsJavaA script trying to start the ClassicsJavaA application. The variable scriptSuccess is used to catch the Integer object returned by the StartClassicsJavaA script. If scriptSuccess equals “1,” then playback can continue. If it equals “0,” an error has occurred and playback terminates, writing the appropriate error message to the log. This can be enhanced to try to launch the application again.

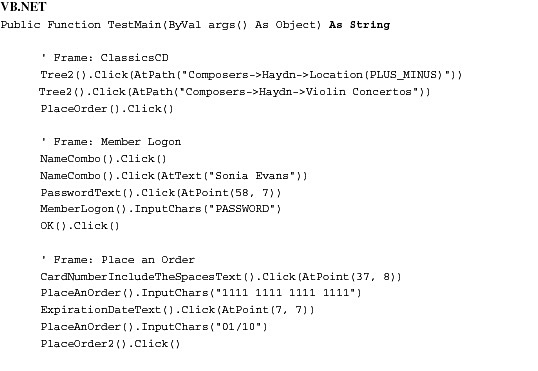
What if you need to pass information from the PlaceCDOrder script to the ReviewOrderHistory script (for example, an Order ID)?You need the PlaceCDOrder script to return the Order ID and the regression script to catch it, passing it as an argument to the ReviewOrderHistory script. This is great illustration of how to manage data consistencies among scripts.

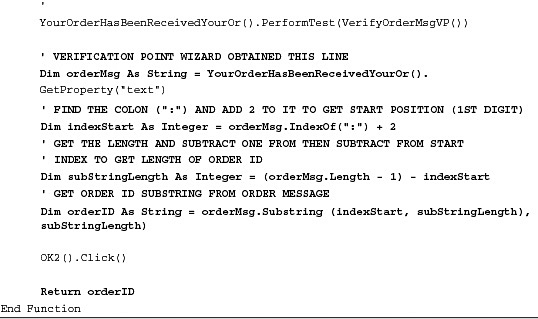
You need to modify PlaceCDOrder, so that you can pull out the Order ID. You need to obtain the message text on the Order Confirmation dialog box. This can be done by inserting a recording into your script, using the Verification Point Wizard to capture the text on the Order Confirmation dialog box. This declares a variable and stores the text value of the label into it. You then need to parse out the actual Order ID and modify your script, so that it returns this value as a string. This is handled in the scripts contained in [Listing 7.32](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex32).

**Listing 7.32** PlaceCDOrder script, return Order ID





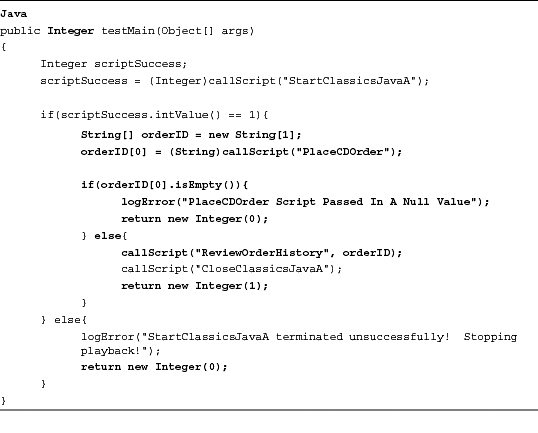


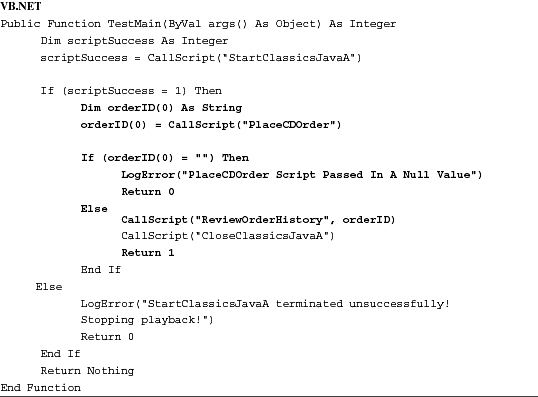


The second thing you need to do is catch the return value in your regression script. Because you plan on passing this into the ReviewOrderHistory script, you should create a single-cell array. This is because you need to pass an Object array into the callScript() method. After you set up your array and catch the return value into the zero cell (for example, orderID[0]), you should probably check if it is a null value (for example, an empty string). This enables you to prevent your script from passing in an empty string and cause issues in the ReviewOrderHistory script. If you do receive a null value, you might want to stop execution. You can handle this in a couple of ways. You can use Rational Functional Tester’s stop() method. This is similar to clicking the Stop button on the script playback window. It terminates execution and writes a *User StoppedScriptError* message in the log. You can also change your regression script so that it returns an Integer object. This enables you to use the return statements to discontinue playback. It also provides the benefit of returning a success (1) or failure (0) flag. This is useful if you are creating regression scripts for each of your application’s major areas of functionality and then calling those regression scripts in a large, system-level regression script. Each of the smaller regression scripts can tell the system-level one if it completed successfully or not.

You will also need to modify your regression script to pass the Order ID into the ReviewOrderHistory script (provided it isn’t null). This is done by passing the array you created to the callScript() method. Refer to [Listing 7.33](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex33) for the appropriate examples.

**Listing 7.33** Simple regression script enhancement #3, managing the Order ID





Of course, you need to modify the ReviewOrderHistory script so that it handles the Order ID. The idea is that it is now an argument being passed in by the regression script. The Order ID itself was captured and returned by the PlaceCDOrder script.

Keeping in line with the other scripts, you might wish to have the ReviewOrderHistory and CloseClassicsJavaA scripts return a value, indicating its success or failure. The easiest way is to mimic the StartClassicsJavaA script, returning an integer value. This enables you to continue managing the flow of your regression script.

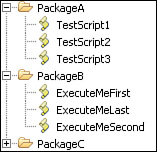
You should now have an understanding of the different techniques that can be employed for managing data and handling script failures within the context of a regression script. Please note that there are always two parts to each modification. You need to edit the regression script *and* scripts that it invokes.

**Creating Regression Scripts that Dynamically Include Scripts as They Are Created**

You can create dynamic regression scripts, using the callScript() method. This enables you to continue adding new scripts to your project, automatically including them in your regression test. This involves managing your test scripts in packages. In some cases, it also involves having your scripts be self-reliant and avoid the modular approach.

The first thing that you will want to think about is organizing your scripts. This is best done using packages (such as creating new folders in your Rational Functional Tester project). One school of thought is to look at the major areas of functionality and decompose them into separate folders. Another way you could do this is via organizing your packages by the use cases of your application. The end result is that you will have multiple packages that contain scripts to test the different pieces of your application. [Figure 7.15](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07fig15) illustrates this.

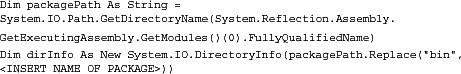
**Figure 7.15** Organizing test scripts by package



The first thing you need to do is obtain the path to your project. This is accomplished with the following line of Java:

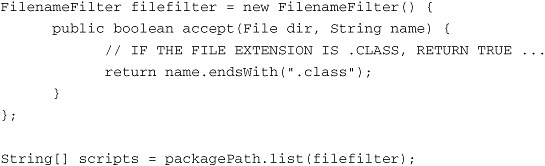
File packagePath = new File(getCurrentProject().getLocation() + "\\" +  
<INSERT NAME OF PACKAGE>);

If you use VB.NET, you need these two lines:



Both examples perform the same task. They obtain the location of the Rational Functional Tester project and append the name of the package (that is the folder) to it. This constructs the necessary path to the scripts that you want to execute.

After you tell the Rational Functional Tester where the package containing your scripts is, you need to obtain the list of scripts in it. The actual directory on the file system that corresponds to the package in the Rational Functional Tester GUI contains both compiled and uncompiled files. You simply need to get the list, using only one type of file. To do this with Java, you need to first create a FileNameFilter and then use this to obtain the array of file names that match the filter. Please see the following snippet of Java code for an example.

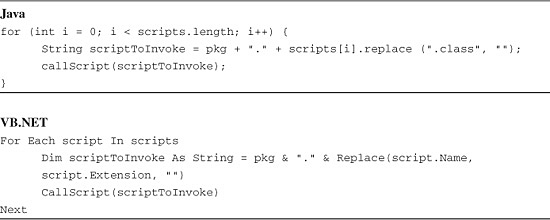


VB.NET makes this much easier. You need only to call the directory object’s GetFiles() method, specifying the extension to filter on.

Dim scripts() As System.IO.FileInfo = dirInfo.GetFiles("\*.testsuite")

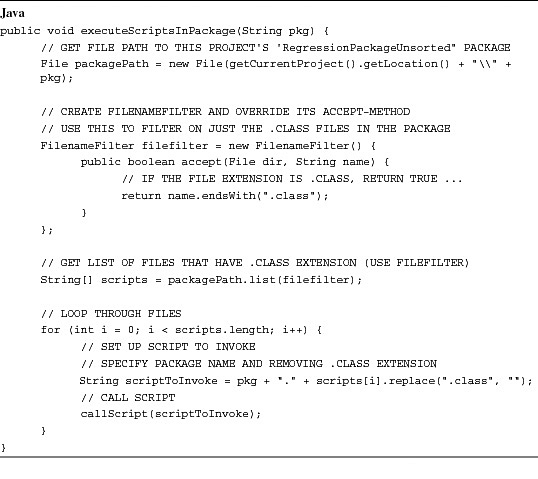
In both cases, you end up with an array of files. You can set up a for loop to access each script. The callScript() method doesn’t require file extensions, so you need to remove them. Also, because you use packages, you need to preface the script name with the name of the package itself, using a period to separate them. This is shown in code snippets found in [Listing 7.34](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex34).

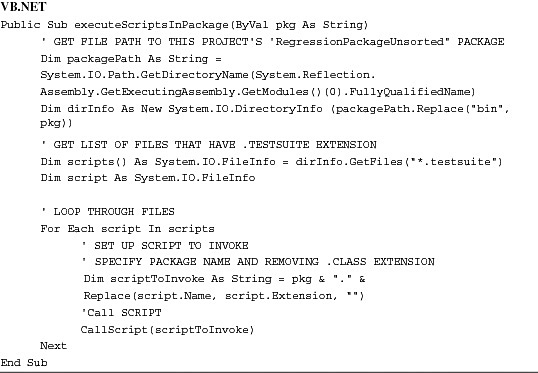
**Listing 7.34** For loop to obtain script names



Because you use this code multiple times and pass multiple packages to it, it makes sense to turn it into a method (or Sub) and store it in a Helper Super Class (or Helper Base Class). This is discussed in [Chapter 3](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch03.html#ch03). The final product, including comments, looks like the examples found in [Listing 7.35](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex35).

**Listing 7.35** Custom method for dynamic regression testing





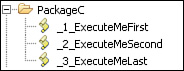
Calling this method from a script is quite easy. You need only call executeScriptsInPackage() and pass in the name of your package as a string. For instance

image

You can now continuously add scripts to each of these packages. The executeScriptsInPackage() method (or subroutine if you’re using VB.NET) simply calls the newly added scripts as it loops through the packages.

One drawback, however, is that your scripts are called alphabetically (as they live in the directory). Therefore, you need to ensure that your scripts are self-reliant. They need to get the application into the state in which they need it to be. You can also add numeric prefixes to your scripts. This ensures a certain order is followed. Please refer to [Figure 7.16](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07fig16) for what this might look like.

**Figure 7.16** Numerically ordered script



You can’t begin the name of your script with an integer; therefore, one solution is to use underscores. This is how the issue was addressed with the package of scripts in [Figure 7.16](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07fig16). When it is passed to the executeScriptsInPackage() method, \_1\_*ExecuteMeFirst* will be executed before \_2\_*ExecuteMeSecond*, which will be executed before \_3\_*ExecuteMeLast*. Without the numeric prefix, *ExecuteMeLast* would be invoked before *ExecuteMeSecond.*

**Executing a Functional Tester Script from the Command Line**

Functional Tester has a rich command-line interface that enables you perform many Rational Functional Tester tasks, including script execution, from the command line. This discussion is limited to using the command line to recompile and execute scripts. Consult the Rational Functional Tester API documentation (the rational\_ft class) to explore other capabilities.

**Introduction to the Rational Functional Tester Command Line Interface**

To use the command-line interface, your command can do two things:

<Launch RFT> <pass arguments to RFT indicating what you want done>

The Eclipse and .NET versions of Rational Functional Tester differ significantly in how Rational Functional Tester is launched. For .NET, launch Rational Functional Tester by executing either rational\_ftw.exe (for Visual Studio version 2005) or rational\_ftw11.exe (for Visual Studio2003). Both executables reside in the Rational Functional Tester installation directory, which can be referenced by the system variable IBM\_RATIONAL\_RFT\_INSTALL\_DIR.

For Eclipse, launch Rational Functional Tester by executing java.exe and passing arguments. The next two sections are devoted to showing you how to launch the Eclipse-based version of Rational Functional Tester.

**Introduction to Executing Java Programs from the Command Line**

We start with a primer on Java command-line options. If you’re comfortable working with Java from the command line, you can skip this section.

The first thing you need to be sure of if you’re going to work with Java from the command line is that java.exe is in your system path. To verify this, open a command prompt window and type **Java**. If Java is not a recognized program displays, you need to run fdsik. Just kidding. You need to add a JRE to your system path variable.

As you are probably aware, the result of compiling Java source files (.java files) are .class files. Many .class files can be packaged together into .jar files.

The entry point to any Java application is a class with a main() method. To launch a Java application (that requires no arguments), you use the following command:

java.exe <Name Of Class With Main Method>

For example:

java MyNiftyJavaProgram.

When you launch a Java program, you are in essence launching two programs: java.exe (the JVM) and the program of interest. Note that you don’t add .class; it’s assumed by the JVM (and you get an error if you include it). This is the simplest case; one in which it is assumed that the directory that contains the MyNiftyJavaProgram.class file is in the execution environment’s (in this case, the shell’s) classpath. If it is not, you need to indicate to the JVM where it should search for .class files. To tell the JVM which directories to search, you either change the system classpath variable or pass a classpath argument to java.exe. You examine the latter.

Assume that the MyNiftyJavaProgram.class file is in the C:\My Class Files directory. To pass this directory as a classpath argument to the JVM when launching the MyNiftyJavaProgram, you would type:

java -classpath "C:\My Class Files" MyNiftyJavaProgram

Note the double quotes. They’re required if the directory you’re passing to the classpath argument contains spaces.

To add multiple directories to the classpath argument, separate them with a semi-colon:

java -classpath "C:\My Class Files;D:\Your ClassFiles" MyNiftyJavaProgram

When you deal with real Java programs (even ClassicsJavaA), you almost always deal with .jar files and not individual .class files scattered about. If the MyNiftyJavaProgram.class file is in a .jar file named Nifty.jar in the same C:\My Class Files directory, you need to add the Niftty.jar file to the classpath to make it possible for the JVM to find the MyNiftyJavaProgram.class file contained within it:

java -classpath "C:\My Class Files\Nifty.jar" MyNiftyJavaProgram

Some .jar files are created in such a way that, internally, they indicate which class is the entry point (has the main method). These .jar files can be double-clicked to launch execution. To launch execution from the command line, you pass the -jar argument to java.exe. You don’t need to indicate which class contains the main; that information is tucked away in the .jar file.

If MyNiftyJavaProgram were such a Java application, you could launch it like so:

java -jar "C:\My Class Files\Nifty.jar"

Now something you can try at home: ClassicsJavaA (and the infinitely more robust JavaClassicsB) are Java applications built in this way. They reside in ClassicsJavaA.jar and ClassicsJavaB.jar in the FTSamples directory of the Rational Functional Tester installation directory. To launch ClassicsJavaB, the command is:

java -jar      "C:\Program  
Files\ibm\SDP\FunctionalTester\FTSamples\ClassicsJavaA.jar"

**Launching Rational Functional Tester (Eclipse) from the Command Line**

You launch Rational Functional Tester by passing to java.exe the name of the main class *or* the .jar file that contains the entry point for Rational Functional Tester. The entry point class is com.rational.test.ft.rational\_ft, which resides in rational\_ft.jar. You therefore launch Rational Functional Tester by executing either of the following:

java -classpath <complete path to rational\_ft.jar>  
com.rational.test.ft.rational\_ft

or

java -jar <complete path to rational\_ft.jar>

If you use the first variation, the JVM needs to know where the com.rational.test.ft.rational\_ft class is located, which you can communicate by passing a classpath argument. The com.rational.test.ft.rational\_ft class is in the rational\_ft.jar file, which is located in the bin directory of your Rational Functional Tester installation directory. The complete path to the bin directory is stored in the name IBM\_RATIONAL\_RFT\_INSTALL\_DIR system variable. To reference the value of this system variable at the command line, place its name in a pair of %: %IBM\_RATIONAL\_RFT\_INSTALL\_DIR%.

If the path to the Rational Functional Tester installation directory contains any spaces, which it will if you accept the default installation directory of C:\Program Files...., you must put the entire classpath in double quotes.

To kick off Rational Functional Tester (without supplying any arguments, which cause it to complain), use the command:

java -jar "%IBM\_RATIONAL\_RFT\_INSTALL\_DIR%\rational\_ft.jar"

Try it out. You should see the following complaint:

Usage: rational\_ft -datastore <directory> [options] ....

**Executing Functional Tester Java Scripts from the Command Line**

To execute a script in the most straightforward manner (such as with the least number of arguments), after using the appropriate command to launch Rational Functional Tester, you need to tell it:

• Where the project is

• That you want to execute a script

• Which script to execute

Let’s assume that you have a project that resides on your local area network in \\Freddy\RFT\_Projects\Nifty and that you want to execute a script that resides in the root of the project called SanityTest. To execute this script, you issue the following command:

**Java**

java -jar "%IBM\_RATIONAL\_RFT\_INSTALL\_DIR%\rational\_ft.jar" -datastore  
\\Freddy\RFT\_Projects\Nifty      -playback SanityTest

**VB.NET**

"%IBM\_RATIONAL\_RFT\_INSTALL\_DIR%\rational\_ftw" -datastore  
\\Freddy\RFT\_Projects\Nifty      -playback SanityTest

If your SanityTest script resides not at the root of the project but in the scripts subdirectory, your command is:

**Java**

java      -jar "%IBM\_RATIONAL\_RFT\_INSTALL\_DIR%\rational\_ft.jar" -datastore  
\\Freddy\RFT\_Projects\Nifty      -playback **scripts.SanityTest**

**VB.NET**

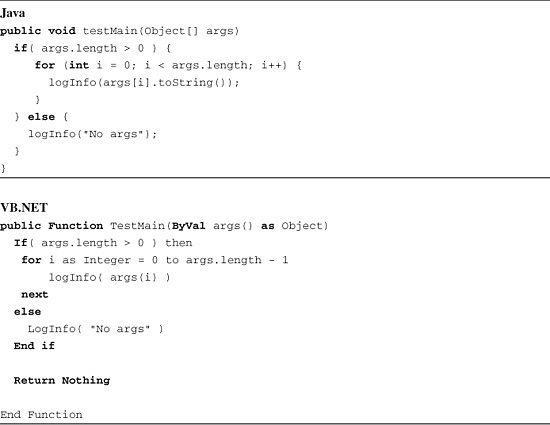
"%IBM\_RATIONAL\_RFT\_INSTALL\_DIR%\rational\_ftw" -datastore  
\\Freddy\RFT\_Projects\Nifty      -playback **scripts.SanityTest**

Note that subdirectories are delimited by a dot, not a backslash. Also note that script names are case-sensitive. If you get the case wrong, an exception is thrown. The Eclipse version of Rational Functional Tester outputs a message to the command line. The Visual Studio version of Rational Functional Tester silently fails.

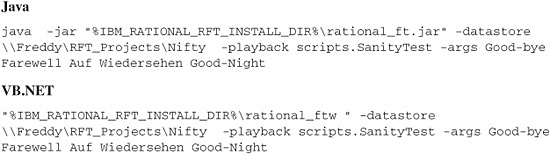
**Passing Arguments to a Script from the Command Line**

[Listing 7.36](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex36) illustrates arguments passing with the following simple scripts.

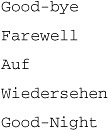
**Listing 7.36** Passing arguments to a script from the command line



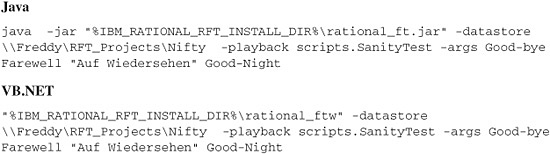
You can execute the scripts in [Listing 7.36](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch07.html#ch07ex36), using the following commands (Java and VB.NET respectively):



These will result in the following words being written to the log file:



To pass Auf Wiedersehen as a single argument, place it in double quotes:



**Compiling Functional Tester Scripts from the Command Line**

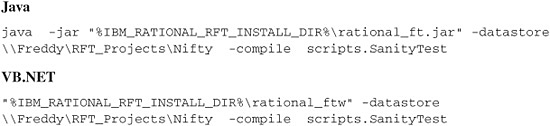
In addition to execution, examine how to compile a script from the command line, because without this, you might run into trouble executing scripts. In particular, if someone makes a change to a script in Rational Functional Tester but doesn’t execute it, depending on settings in Eclipse, you run the risk of not executing the latest version.

To compile a script from the command line, you must be sure that the standard Java compiler—javac.exe—is in your system PATH (Eclipse, incidentally, does not use javac.exe; it uses its own *incremental compiler*). When you install Functional Tester, a full JDK is installed in the main SDP directory, for example, C:\Program Files\IBM\SDP\jdk.

To compile a script, pass the following arguments to Rational Functional Tester:

-datastore <> -compile <script>

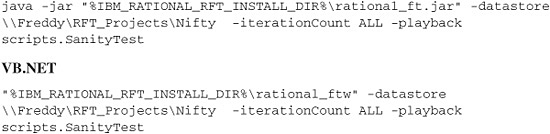
To compile your SanityTest script, use the following:



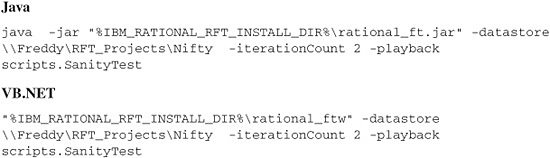
No news is good news. The only output you see from compilation are compilation errors.

**Passing Datapool Arguments**

If your script is associated with a datapool and you want to iterate through all records, pass the iterationCount ALL argument:



If you just want to use a specific number of rows, pass that number:

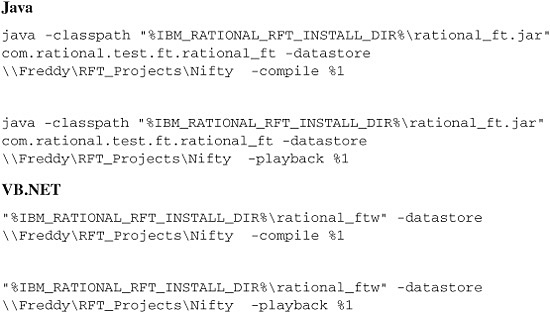


**Executing Functional Tester Scripts from a Batch File**

As comfortable and confident as you might feel at the command line, you might have better things to do with your time than type long commands (especially when typos are taken into consideration). To make it easier to execute scripts from the command line, create a .bat file.

Begin by creating a modest batch script—one that is easy to use (involves little typing) but is the least flexible. It compiles and executes any single script that you want as long as it’s in the Nifty Rational Functional Tester Project.

To create the batch file, enter the following commands in a plain text editor (be sure each is on a single line without a carriage return). Substitute the path here with the path to one of your Rational Functional Tester projects:



The %1 token refers to the first argument passed to the batch file. Save the file in any directory that’s in your system PATH under the name **runRFTScript.bat**. To execute a script, issue this command:

runRFTScript <script name>

What’s inflexible about this batch file is the following:

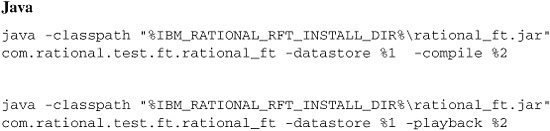
• The path to the datastore is hard-coded.

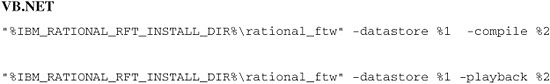
• You can’t pass arguments to the script.

If you wanted to execute a script in a different Rational Functional Tester project, you would have to either:

• Change the path in the batch file.

• Make the path a variable and allow your users to pass whatever path value in (shown in the following code).





Now you can execute any script in any project, but you have to pass two arguments: The path to the Rational Functional Tester project and the script to execute is as follows:

runRFTScript \\Freddy\RFT\_Projects\Nifty      scripts.SanityTest

**Scheduling Script Execution**

To schedule script execution, your best bet is to create a batch file that executes the desired scripts. The simplest implementation of this is to create a main Rational Functional Tester script that knows all the scripts that need to be executed (and executes them by invoking callScript()). Your batch file needs only to execute the main Rational Functional Tester script.

After your batch file is created, you can schedule it to run as a Windows task. To schedule a .bat file to run as a Windows task, do the following:

**1.** Open **Scheduled Tasks** in the Control Panel.

**2.** Select **Add Scheduled Task**.

**3.** In the list of applications, click **Browse** and select your batch file.

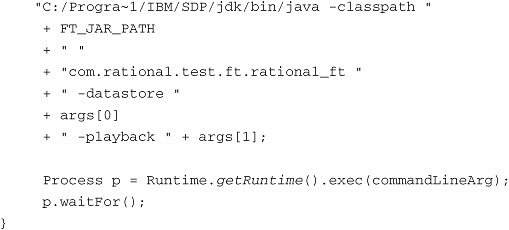
**4.** Enter a name for your task and choose how often you want it to execute. Initially, the choices seem a bit limited. If you don’t see the frequency with which you want to execute (such as every day at 6:00 AM and 6:00 PM), don’t fret. You can fine-tune the frequency by going into Advanced Options after the task is created.

**Executing a Functional Tester Script from a Plain Java Class**

You can execute Rational Functional Tester scripts from a class that’s not a Rational Functional Tester script, such as from a Java main(), but Rational Functional Tester *must* be installed on the machine from which you’re executing.

To execute from a Java main(), use Java’s capability to execute system commands to execute a script via the command-line interface:





**Summary**

You should now have a good sense of how to enhance your scripts for different scenarios. Whether it is dealing with errors, adjusting playback options for small snippets of your scripts, or using the command line to execute your scripts, you now know the techniques to deal with them.

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

[Chapter 6. Debugging Scripts](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch06.html)

[Next](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch08.html)

[Chapter 8. Handling Unsupported Domain Objects](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch08.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/ch07.html)