JavaStar Tutorial

THE NETWORK IS THE COMPUTER™

SunTest, Inc.
A Sun Microsystems, Inc. Business
901 San Antonio Road
Palo Alto, CA 94303 CA USA

Contents

Introduction to JavaStar 1

This chapter introduces you to JavaStar and this tutorial, describing:

- The Continuum of GUI Test Tools
- The JavaStar Model
- Looking at the Example Application
- What this Tutorial Covers
- Using the Tutorial Directories and Files (including how to skip around between chapters)

Even if you plan to skip around in the tutorial, be sure to start here to get basic information that applies to all chapters.

The Continuum of GUI Test Tools

To place JavaStar in the continuum of developing test tools, it makes sense to look at how GUI test tools have developed over the years and what features they offer.

The first GUI test tools were coordinate-based, using bit-mapped images. These tools recorded keystrokes and mouse movements based on the screen coordinates affected, and then played them back to simulate user interaction. One of the problems with this approach was that any change to the GUI meant that tests had to be re-recorded; it was nearly impossible to edit them.

Later test tools improved GUI testing by simulating an object-oriented approach. The tool defined “widgets” in the program under test, and would send events to these widgets to simulate user interaction. Here, a change to the GUI meant editing all tests, but at least it was easier to edit them.

When the concept of window declarations of GUI maps was developed, test developers had a way to abstract component information. By abstracting the component information, developers could deal with GUI changes that would otherwise break existing tests. Tweaking the map for an application would update all tests by reference. This worked well except for internationalization, where most products required a map for every locale.

JavaStar continues the evolution by providing an object-oriented tool that offers an easy way to generate declarations for the program under test. JavaStar uses Java to ensure a true object-orientation, not a simulation. JavaStar accesses the Java AWT components used to create the GUI of the test program and works directly with these components in the tests it creates. Because these tests are created in Java, they are object-oriented.

The JavaStar Model

JavaStar uses two types of files—scripts and JavaStar Test (JST) files. A script is a Java program (constructed by JavaStar or by hand) that tests a select portion of your test program's user interface. A JST defines a test consisting of multiple scripts and other JST files.

To understand this better, it might help to think of a script as one of those interlocking building blocks you might have played with as a child. On its own, an individual block might not have been too exciting, but you could combine it with other blocks to create a complex structure. Each block was re-usable, too. A block you used to create an airplane could also help you build a house.

A script is similar in that, on its own, it might test only one feature of your application. But, combined with other scripts, you can create a JST that exercises your entire application. Better yet, you can create multiple JSTs that each use these scripts in different ways, providing you with a strong test suite.

For creating scripts and JSTs, JavaStar provides you with:

- The capture tool for creating effective scripts—one that record keystrokes, mouse movements, and comparisons
- An API you can use to extend these scripts even further, or use to write scripts entirely by hand
- The composer you use to graphically assemble scripts into JavaStar Tests

Using the Test Composer

The Test Composer is a lot like a work area. Here, you define which scripts you want to use, the order in which you want them to execute, and the conditions under which you want them to run. For example, at some point in your test, you might want one script to run if the previous script ended successfully, and another script—perhaps a recovery script—to run if the previous script ends with an exception. In addition, you might want the recovery script to restart the application before it executes your steps, just to make sure you return to a clean state. You can define all of this in a matter of seconds using the Test Composer's point-and-click interface.

You are not limited to using scripts within JavaStar Tests (JSTs)—you can also reference other JST files. For example, if you wanted to define an acceptance JST for your application, you might want to include a number of already-

defined JSTs (maybe one that exercises each application window) as part of the test. You just reference a JST file as you would a script, and JavaStar includes it in the test.

JSTs and their nodes (the individual scripts and JSTs that make up the composed test) also accept parameters. This means that you can create highly flexible scripts and tests that you can use for a number of purposes, just by changing the parameters.

Running JSTs and Viewing Results

When you run a JST, JavaStar opens a window to show you a graphical display of the test-nodes in the JST and flashes each node as it executes. You can easily track where the test is in execution, and, should the test suspend playback, you'll know where to look for the problem.

When you view results of a test run, JavaStar displays these results as an outline tree that follows the structure of the JST itself. You can look at the results on a node-by-node basis, or read summary information for the entire test. For large tests, this makes the results easier to view and understand.

Analyzing Applications for Test

Before you start developing tests in JavaStar, you may already have in mind some tests you want to create. To take advantage of the test tool's power, you can start by breaking the tests down by function, creating discrete building blocks that you might later combine for a test. These building blocks determine individual scripts.

You can also begin your test definition by examining all the components of your GUI and determining how to test each component. If you look at ways to make each test flexible and re-usable, you can quickly determine which scripts you need to write.

You'll probably end up using both of these methods to define your test suite—this tutorial uses a little of each as an example.

Looking at the Example Application

This tutorial uses the Name Database, one of the example Java applications included with JavaStar. If you look in the `javastar\examples` directory, you'll see there are three versions of this application. Most of the tutorial uses namedb1, though later lessons contrast this version with others.

The Name Database is a straightforward, no-frills, name-and-address database application. It provides functions you can use to:

- Enter name and address records into a database
- Open, save, and close name databases
- Change and delete records
- View all the names in the database as a list and select a record to view
- Search any or all fields in the database for a text string, then view the records that match

While the Name Database provides the simplicity this tutorial needs for an example, it isn't the kind of full-featured application that you might make available to customers. While your applications are likely to be larger and more complex than namedb, you'll use the same approaches to testing them as you'll learn here for namedb.

Looking at the GUI for the Name Database, you can see that the application has three windows (Figure 1-1 shows these):

- Name Database main window
- Names dialog
- Search dialog

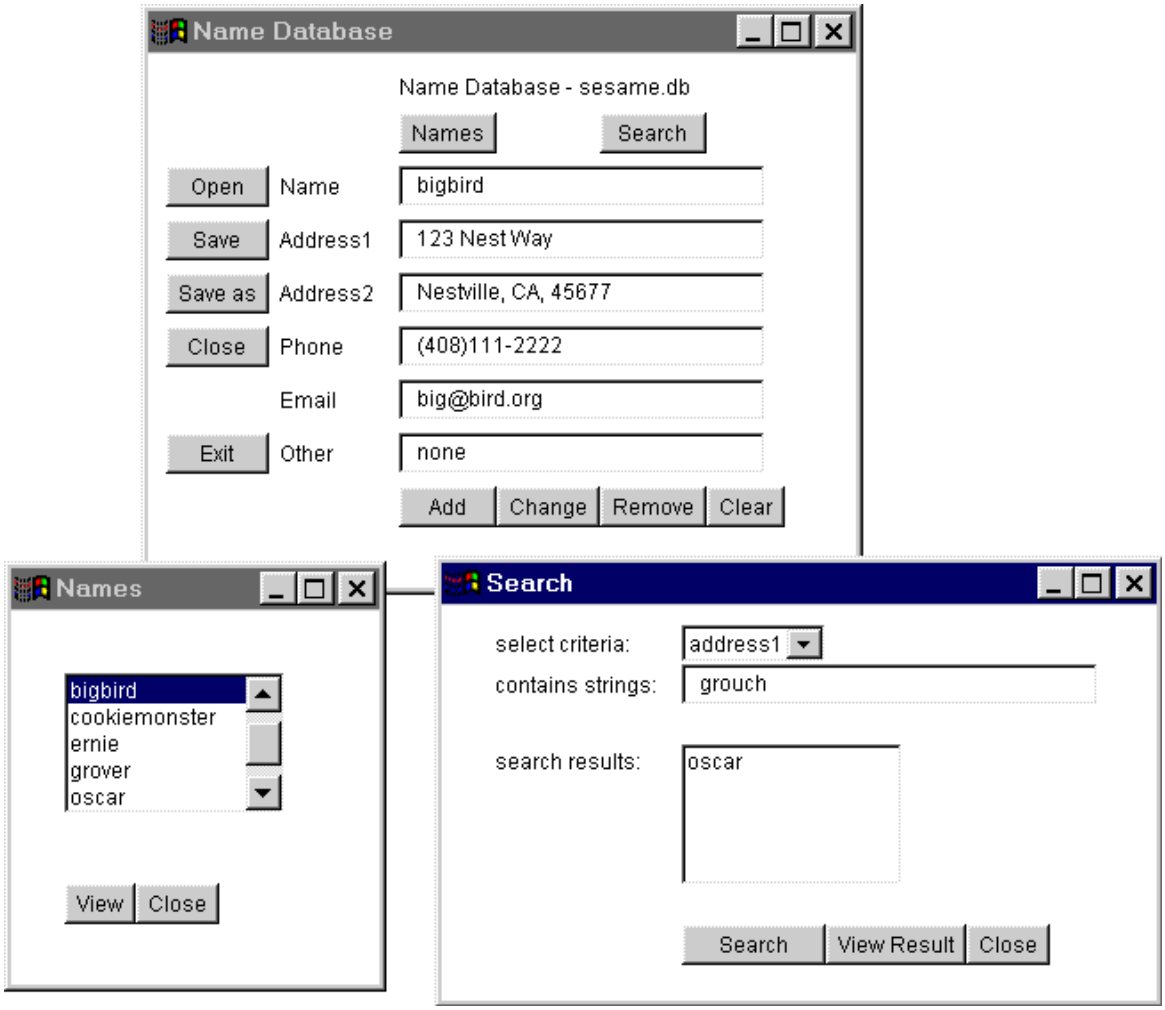


Figure 1-1 Name Database windows

The tutorial uses these three windows to demonstrate the principles and techniques of JavaStar testing. The goal is to give you ideas on how you can plan testing for a Java GUI, and to show you how much power JavaStar gives you by providing you ways to quickly build complex tests.

What this Tutorial Covers

The JavaStar tutorial is divided into two sections: Part 1, Java Star Basics; and Part 2, Advanced JavaStar.

Part 1: JavaStar Basics

The first part of the tutorial is for all JavaStar users. Even if you are familiar with other GUI test tools, stepping through the basic tutorial will help you get comfortable with the JavaStar controls, as well as the JavaStar test model.

"Getting Started with JavaStar**"** guides you through recording your first JavaStar script, playing the test back, and viewing the results. From this, you'll learn what you need to do to run JavaStar, and you'll get comfortable with the recording controls, including verification operations.

"Moving to the JavaStar Model" introduces the JavaStar test model and describes how it improves tests to make them much more versatile and powerful.

"Generating Declarations**"** shows you how to generate declarations from your application under test, and then reference these declarations in your scripts.By doing this, you can work around many problems that arise when developers change the program interface during your test phase.

"Using a Modular Approach**"** examines the test created in the previous chapter and analyzes why a modular approach would be effective. This chapter shows you how to compose a JavaStar Test (JST) file from multiple scripts, how to analyze the results from a modular test, and shows you how to navigate through the multi-level results.

"Adding Parameters for Flexibility**"** takes the scripts you created in the Modular Approach chapter and shows you how to turn the inputs into parameters. You'll learn how to edit the JST file to pass parameters to scripts, increasing re-use potential for your code.

"JavaStar from the Command Line**"** shows you how to run JavaStar tests from the command line, making it easier for you to build test suites that you launch using a test harness.

Part 2: Advanced JavaStar

This section of the tutorial is designed for those JavaStar users who know how to program in Java. Part 2 shows you how to use advanced JavaStar features and develop highly customized test solutions with the JavaStar API.

"Using the JavaStar API**"** addresses how to further customize JavaStar scripts with your own code and how to access the power of the JavaStar API. This chapter illustrates the anatomy of JavaStar script so you can determine what to edit and what you need to leave intact.

"Using Non-Component Locators**"** shows you how to test an application or applet created using a non Java-AWT toolkit. It covers how to use existing locators.

"Testing JFC Components" specifically addresses the use of non-component locators for testing Swing, or JFC, components.

"Writing Non-Component Locators" is a detailed guide on developing non-component locators for use with JavaStar.

Using the Tutorial Directories and Files

While this tutorial follows a specific example from beginning to end, with each chapter building on the previous one, you can do the exercises out of order. The `tutorial` directory contains a subdirectory for each chapter. These contain any files you need to begin with and the solutions to exercises. You'll get instructions on when to copy files to your work directory to continue the lessons.

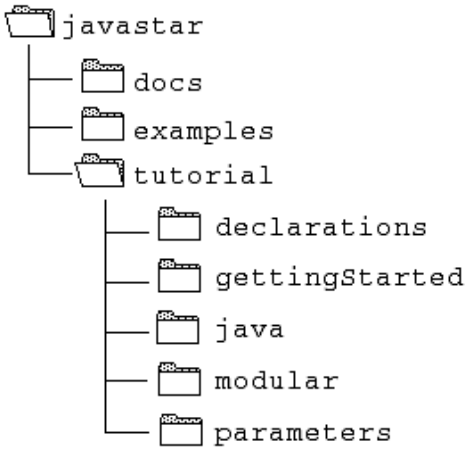


Figure 1-2 JavaStar tutorial directory tree

The instructions in each chapter assume that you using the `tutorial` subdirectory as your work directory. This is where you'll store the scripts you create, as well as the test results. If you decide to use a different directory, keep the same relative file locations. Otherwise, not all the scripts will run.

Part 1 — JavaStar Basics

Getting Started with JavaStar 2

This chapter gives you a quick way to get up and running with JavaStar and familiarize yourself with many of the controls.

You'll create a test for the Name Database that tests:

- Loading a database
- Adding a record
- Searching for the record and verifying the results

This is not an exhaustive test of the Name Database capabilities. However, it introduces you to the basics of recording and playing back scripts with JavaStar. In later chapters, you'll look at how to improve the approach to take advantage of more advanced JavaStar features. The lessons that follow take this first test and transform it, step-by-step, into a test optimized to use the JavaStar model.

After completing this lesson, you should be able to:

- Set up your environment to run JavaStar
- Launch JavaStar
- Record a script that captures key and mouse strokes, and that compares results
- Play back a script using Run Test
- Examine results in the Results Viewer, and use the Viewer options to filter the results for the information most important to you

Topics:

Setting up JavaStar

Preparation for this lesson involves:

- Making a Copy of the Test Database
- Launching JavaStar
- Creating a Project File

> ***Note –*** This tutorial does *not* provide installation instructions for JavaStar. For step-by-step installation details, see the chapter "Preparing to Use JavaStar" in the *JavaStar User's Guide*.

Making a Copy of the Test Database

In this tutorial, you'll be using a database (`.db`) file that stores names and addresses. Each test run will change the contents of the database. Because each test will require that the database be in its original state, you'll need to preserve a "clean" (unedited) copy at all times. Before each test run, you can copy the original database over the edited version.

1. **Copy the `sesame.db` file from the `JavaStar\examples\namedb1` directory to the `JavaStar\tutorial` directory.**
2. **In the `JavaStar\tutorial` directory, copy `sesame.db` to `test.db`.** Now you have two databases in the tutorial directory—the `sesame.db` master database, which you'll keep clean and unedited, and the `test.db` test that you'll replace each time you run a test.

If you prefer to always copy the `sesame.db` file from the `JavaStar\examples\namedb1` directory, you just need to be sure that no other users have access to that directory. If they do, they might inadvertently load and use this database, corrupting the integrity of your tests.

Launching JavaStar

Before you launch JavaStar, make sure you are in your working directory.

> ***Note –*** Your JavaStar work directory defaults to the directory where you launch JavaStar. You can reset this default within JavaStar, *however*, this doesn't effect the default directory that appears when you open a file dialog window. This becomes an issue when you are opening file dialogs from within your application. But because JavaStar stores the file you open as a path relative to your working directory, this is only an issue if you change your working directory in a later test run.

Launching JavaStar from a UNIX Environment

1. **If you are not already in the JavaStar `tutorial` directory, change to that directory.**

2. **From the command line, type:**

   JavaPath `javastar`

   For example, if you already have `java` defined in your system path, you type:

   ```
   java javastar
   ```

   Note – If you get an error message stating that namedb cannot be accessed, click OK to dismiss the window. You'll define the proper path to namedb in the upcoming section Creating a Project File.

Launching JavaStar from Windows 95 or Windows NT

- **Double click on the JavaStar icon on your desktop, or choose JavaStar from the Windows Program Manager.**

Main Menu

If your launch is successful, you should see the JavaStar opening screen, with the main menu displayed to the left:

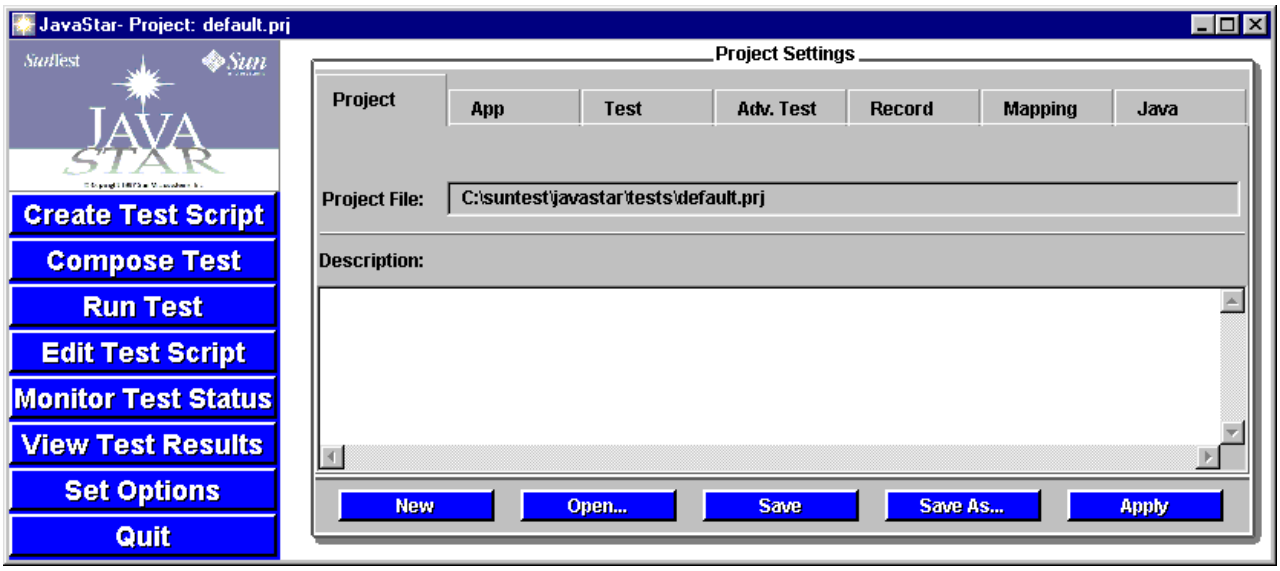


Figure 2-1 JavaStar main screen

If you get the message `Can't find class javastar`, check your `CLASSPATH` setting to make sure the path to JavaStar is correct.

Creating a Project File

For the next several lessons, you'll be working with the same working directory and the same test application. To set your test environment defaults, you'll create a project file that stores information about the application under test, your Java settings, and other information.

You can find the Project Settings panel on the main screen, to the right of the main menu.

To create a project file for the tutorial:

1. **Set the application settings for the project.**

   a. **In the main screen, click on the Project Settings** App **tab.**
   The panel for application settings moves to the forefront. By default, the application option should already be selected.

   b. **Set the** Class **field using the** Browse... **button.**
   Click the Browse button to bring up the file dialog. Navigate to the `\javastar\examples\namedb1` directory and double-click on the file `namedb.class`.

   The `CLASSPATH` field is automatically filled in with the path to the application you selected.

2. **Set the project test settings.**

   a. **In the main screen, click on the Project Settings** Test **tab.**
   The panel for test settings moves to the forefront.

   The default settings for the work directory and the results directory (where JavaStar writes .log and other result files) appear in italics beneath their respective text fields.

   b. **If your default work directory, results directory, and JST path settings do not point to the tutorial subdirectory, change them accordingly.**
   You can use the Browse... button to navigate to the directory you want, or type the path in directly. For example, if you are running JavaStar in a Windows environment and used the default installation, type:

   ```
   C:\SunTest\JavaStar\tutorial
   ```

   for each path.

3. **From the buttons along the bottom of the screen, choose** Save As...**.**
   The window closes and your changes are written to the JavaStar properties file.

4. **Enter `namedb.prj` as the file name and save.**

Recording a Script

Now you're going to record a script that tests several functions of the Name Database. Testing multiple functions in a single recorded test script isn't necessarily the recommended way to build tests—Chapter 2, "Using a Modular Approach" discusses this in more detail—but for now it provides a tour of the JavaStar features. It also gives you a deeper appreciation of why you'll later want to compose tests of many small scripts instead.

Because this process contains a lot of steps—click here, type this, click that—this part of the tutorial is broken down into sections. These sections are all part of one script, though, so you can't selectively carry out sections and have the script run properly. If you're the type of person who learns best by reading these instructions through before, instead of carrying them out, you'll find this organization a bit more readable.

Start Creating the Script

1. **Start the application you want to test:**

   a. **In the JavaStar main menu, click** Create Test Script**.**
   This opens the Create Test Script dialog. This dialog includes the same fields in the Project Settings App panel; this is so you can override settings for a session, or, if you have not defined a project file, define settings to use while the Record / Playback window is open.

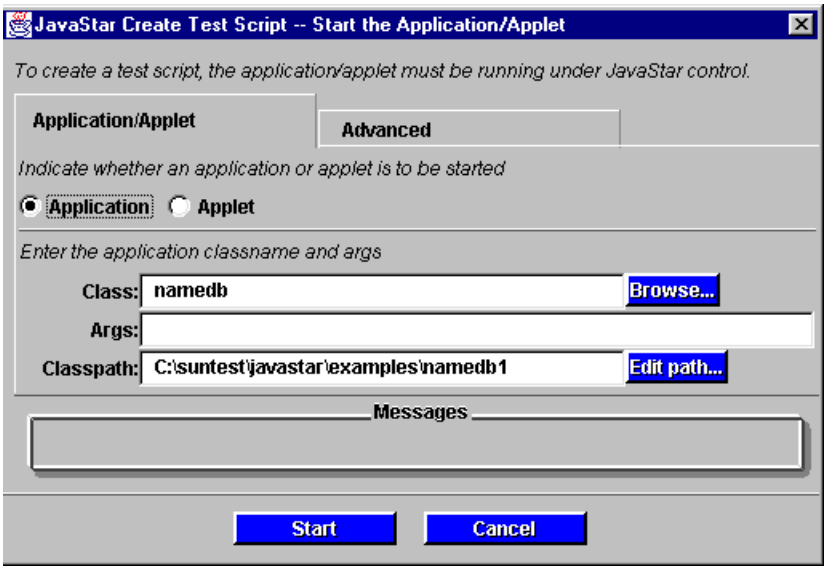


Figure 2-2 Create Test Script dialog

If you created a Project File as described in "Creating a Project File," the Class and Classpath fields should already be filled in.

b. Click the Advanced **tab**

This brings the advanced options to the forefront. See Figure 2-3 for an example. Here you can override more project file settings, including the directory settings you defined in the Project Settings Test panel. For now, you're just going to override the logfile name.

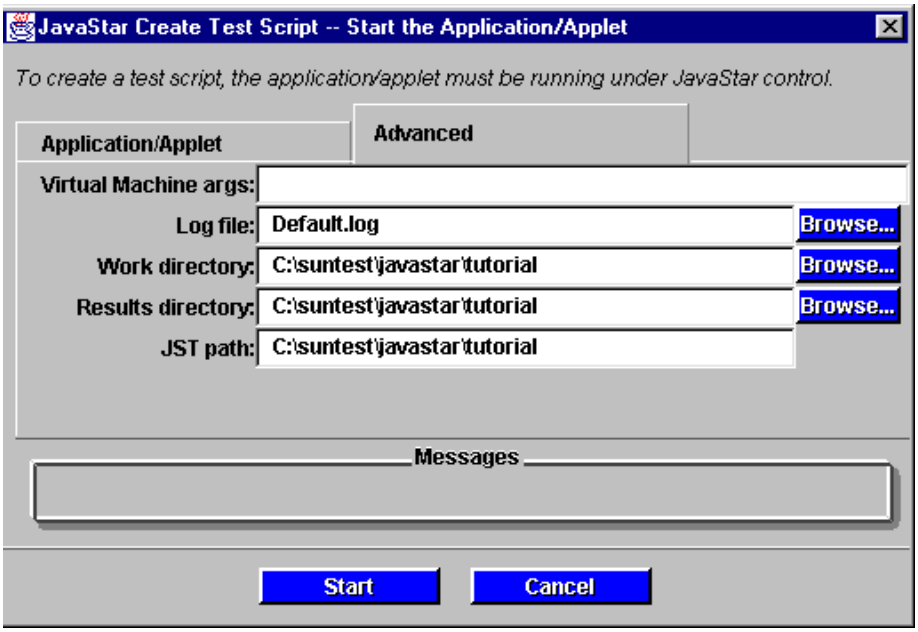


Figure 2-3 Advanced tab for Create Test Script

c. **Change the** Log file **field value to** `tutorial.log`**.**

d. **Click** Start **to launch the application under test.**
The Record / Playback window and the Name Database application open.

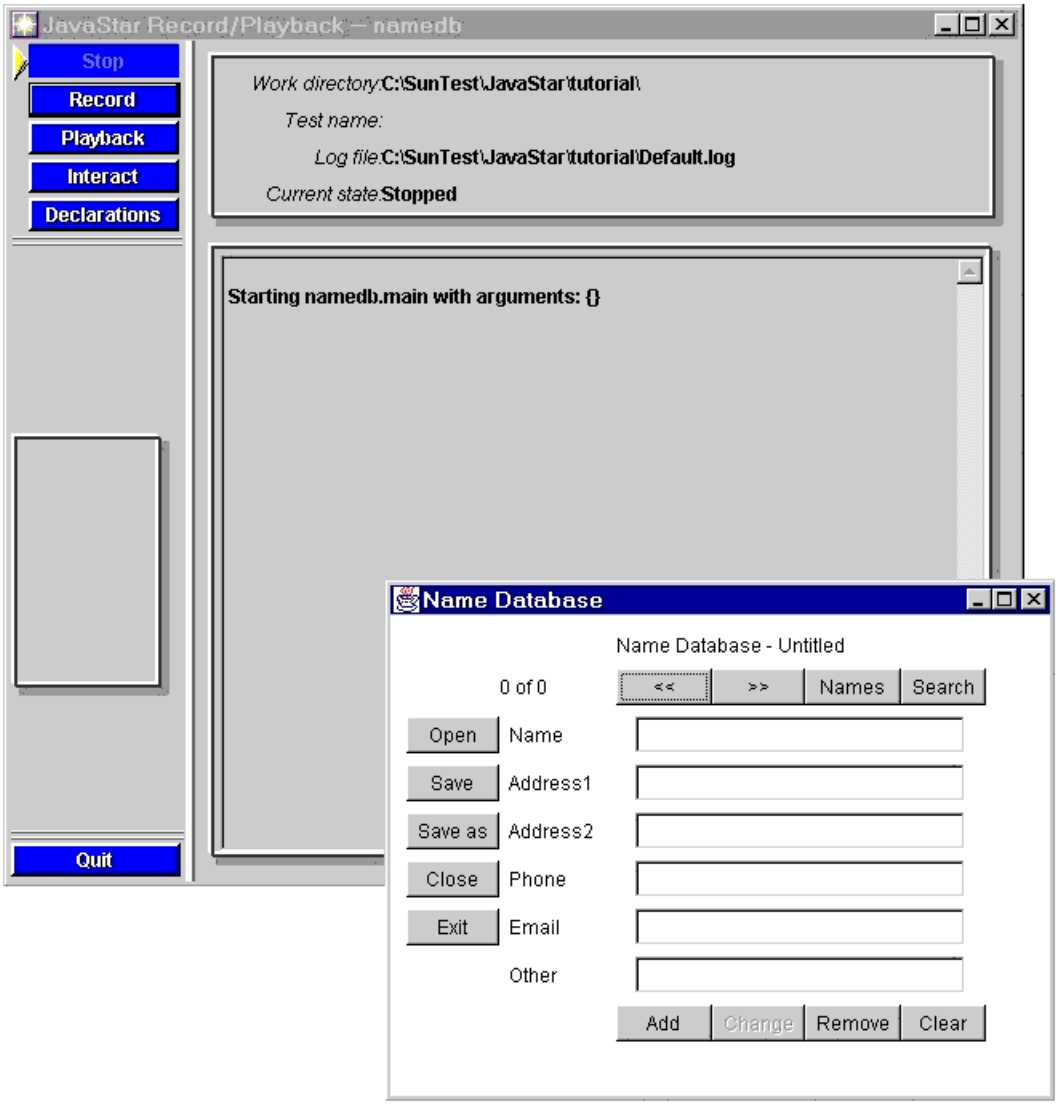


Figure 2-4 Record / Playback window

If namedb does not start

If you get the message:

```
There is some problem accessing the class classname. Either:

1. It is not in the CLASSPATH.
2. It is being accessed in the wrong way.
(e.g. String is invalid, java.lang.String is correct.)
```

Here are some things to try:

- Make sure your application path is reflected accurately in the Classpath field (check your Project File, too). If you typed the path in directly, you might try using the Browse button to navigate to the directory, so you can be sure you have the latest path.
- Check that you spelled the application name correctly in the Class field of the Application/Applet tab. Do not include the `.class` extension. Make sure this class is the "main" class of your application. Use Browse to navigate to the directory and make sure the `.class` file is still there.

Verify that you are using a fully-qualified class name. If your class is within a package, be sure to type packageName.className.

2. **Move the Name Database window to one side.**
   Position it as best you can (given your monitor size) so you can see both the Record/Playback Window and the Name Database window. This makes it easier to record your interaction with the application under test while keeping the JavaStar controls accessible.

3. **Begin record mode**:

   a. **Click** Record**.**
   This brings up the Record Test Script dialog.

   b. **In the** Create Script **field, type** `TestNameDB`**.**
   This lesson does not use non-component locators or map files, so leave the fields relating to these topics blank. The advanced JavaStar tutorial for "Generating and Using Declarations" addresses Record with map files, and 'Locators for Non-Components" describes Non component locators.

   This window also provides you with an option to toggle the Record with delays checkbox. If you turn this feature on, JavaStar will record the time of any delays between events and add this to the script. You can later scale the delays using settings in Playback Options. For this script, delays aren't important, so you can leave this box unchecked.

   Note – If you were recording a script that interacted with a Canvas component, recording with delays would be a good idea. With a Canvas, the speed of user interaction has an affect on the result, especially when combined with the speed of the system on which you run the test. Taking advantage of the delay feature gives you more control—you can later scale that delay to compensate for system speed.

   c. **Click** OK**.**
   You're now in record mode. Anything you do in a Name Database window will be recorded to your script.

Opening the Test Database

The first step of the test will be to load the database into the Name Database application. Whether or not this first step succeeds is critical to the integrity of the test—if it does not succeed, your later results are compromised. To ensure this doesn't happen, you'll insert a synchronization comparison.

JavaStar provides two types of comparisons: verifications and synchronizations. The difference between the two is not in what you can compare (that's identical) but in how JavaStar responds to the results during playback.

With a verification, JavaStar performs the comparison, evaluating the results as you specified. If the verification fails, JavaStar checks the component again, repeating until a specified timeout interval has elapsed. If the verification hasn't succeeded by the time the timeout expires, JavaStar logs this as a verification failure and continues on with the test.

A synchronization works the same way with one exception—if the comparison fails at the timeout, JavaStar throws an exception and the script ends abnormally. Terminating the current script after a failed synchronization and indicating that the termination was abnormal is important, because a synchronization error requires recovery. (In the next chapter, you'll learn how to compose a test that automatically handles this kind of recovery.)

While you follow the steps in this section, the Record / Playback window should be open and visible on your desktop. This window contains the controls for recording (to the left) and dynamically displays the log file for this session in the text panel to the right.

1. **Make the Name Database window active.**
   Click on the border of the Name Database window.

2. **In the Name Database window, open the test database.**

   a. **In the Name Database window, click** Open**.**
      This brings up the file dialog window.

   b. **Open `test.db`.**
      If the file dialog doesn't automatically open to the `tutorial` directory, navigate to that directory. Select `test.db` from the list of files. Click Open. The file dialog closes.

      If you are following this tutorial on a UNIX system, you may be prompted with OK instead of Open within the file dialog window.

   Note – JavaStar doesn't record specific mouse-clicks or movements because, if it did, the tests wouldn't be platform independent—what worked for a UNIX file browser might not work for a Windows 95 file browser. By passing only a relative file call to the dialog, JavaStar keeps your test platform-independent, even if the file dialog itself is not.

3. **Now, to setup a comparison to check that the right file is open, click** Synchronize **in the Record/Playback window.**
   When you click Synchronize, JavaStar automatically pauses record mode. The right portion of the window also changes to show prompts for the operation you selected.

4. **Set up the synchronize operation as follows:**

   c. **Click on the component you want to verify in the application under test—in this case, click on the** Name Database - test.db **label at the top of the Name Database window.**
      Note that the text panel of the Synchronization window displays the selection code. In addition, JavaStar flashes the component so you know which one you chose.

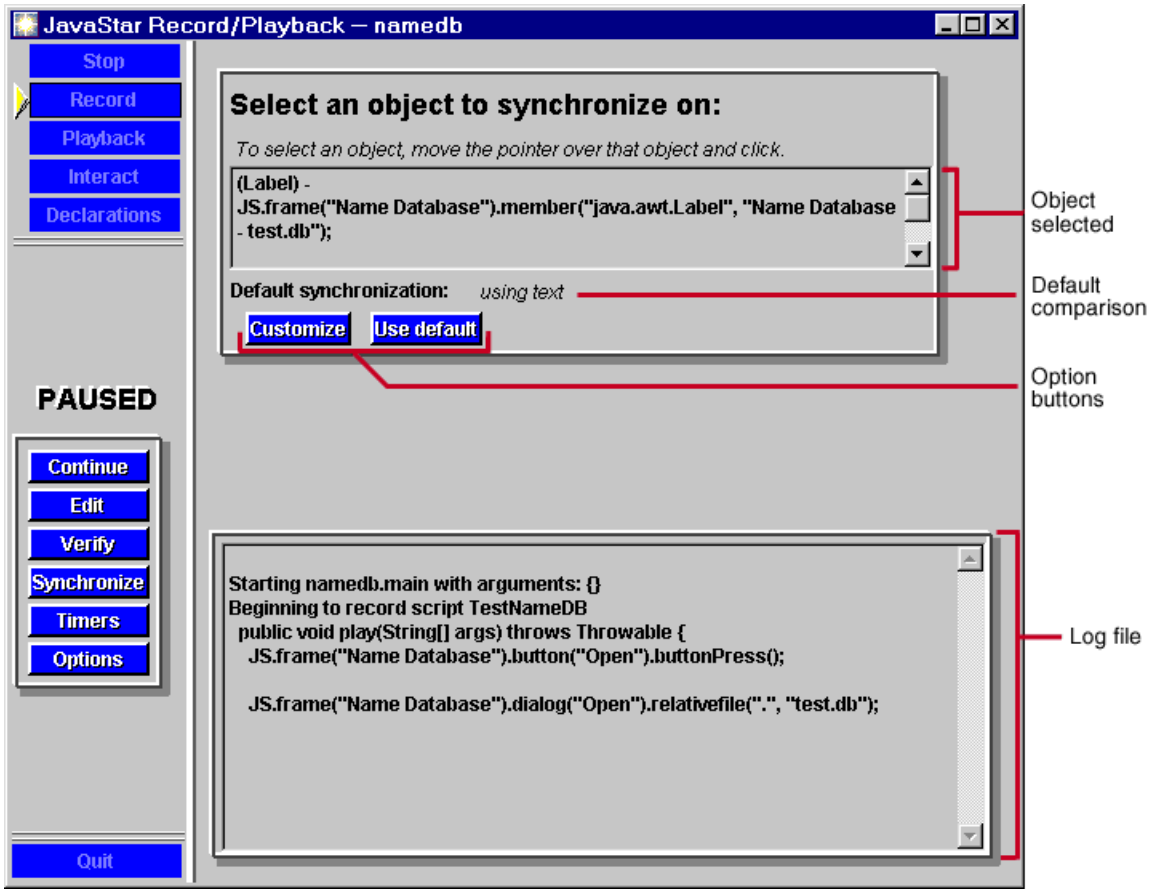


Figure 2-5 Select for Synchronization

Selection code for the object you selected appears in the synchronization select panel. Immediately below, JavaStar lists the default method of synchronization, and gives you the option to choose the default or customize the method of comparison.

The bottom panel displays the log file for this session.

d. Click Use default**.**
The panel changes to prompt you for a purpose for the comparison. The object to compare and the method of comparison are displayed here, as well.

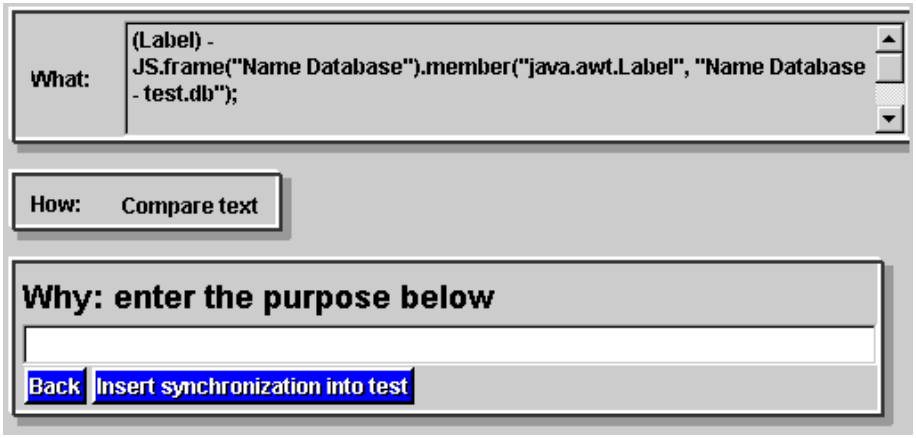


Figure 2-6 Synchronization prompt for purpose

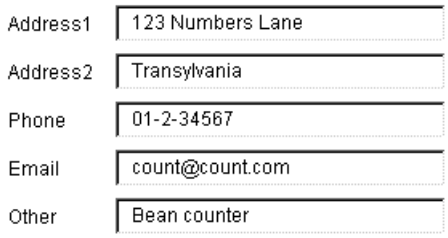
e. **At the prompt to** enter purpose below**, type `Continue only if correct file loaded`.**
While you don't have to type a purpose for the comparison, it can be helpful to do so. JavaStar includes the purpose in the test results, making it easier for you to evaluate results. Because this string also appears in the script code, it can help you understand the script if you decide to edit it in the future.

f. **Click the** Insert synchronization into test**.**
JavaStar returns you to the first synchronization panel.

g. **In the left button bar, click** Continue**.**
The right portion of the window changes to display recording data, and record mode resumes.

Recording Text Input

1. **In the Name Database main window, clear the display.**
   Click Clear. This only clears the display—it doesn't remove the record that displayed when you opened the database.
2. **Enter record for `Count von Count`.**
   a. **Click in the** Name **text field and type `Count von Count`**
   b. **Using the TAB key to advance through the fields, enter the following information into the remaining fields:**



3. **Click** Add **to update the database with this information.**
4. **Save the database.**
   Click the Save button.
5. **Click** Clear **to clear the display.**
   This returns the application to a "neutral state" before proceeding to the next step, where you'll select this record from the names list and verify that it displays accurately.

Checking the Search Operation

Now you're ready to verify whether the search function will locate the record you just added. As mentioned earlier, the verify option works similarly to synchronize—the dialog is almost identical. The difference is how JavaStar processes the two types of comparisons. Synchronizations that fail throw an exception, but do not affect the pass/fail count for comparisons. In contrast, verifications that fail do not throw exceptions (meaning that your test is not interrupted) but JavaStar notes them as comparison failures in the test log. Each type of comparison is useful in different situations.

For comparing the search results, a verification makes sense. If the search fails, it doesn't necessarily affect the integrity of the tests that follow. Noting the failure in the log is sufficient.

This part of the test uses two types of verification:

- A verification of the search results list
- A verification of each field of the record

When you verify the search results list, you'll be looking for the number of items returned by the search. For this exercise, you'll obtain that value using the VerifyAny feature to compare the return values for any of the component's methods and variables. You'll compare the return value of the list component's `getItemCount()`.

To verify each field of the record that the search operation returns, you'll use the "using text" option. This is the same option you used when synchronizing to the database filename.

1. **To bring up the Search window, click** Search **in the Name Database main window.**

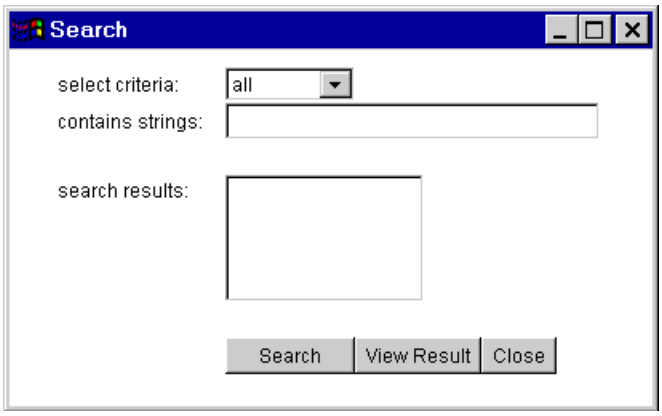


Figure 2-7 Search window

2. **From the** select criteria **choice list, choose** address2**.**
   This operation will now only search the address2 field.
3. **In the** contains strings **field, type** `Transylvania`**.**
4. **Click** Search**.**

5. **Verify the number of items returned by the search**:

   a. **In the Record/Playback window, click the** Verify **button.**
   The right portion of the window changes to show the first verification panel: select an object to verify. This panel is very similar to the Synchronization panel you worked with earlier.

   b. **In the Name Database search window, click on the search results list as your object to verify.**
   The selection code for the list appears in the verification panel. The default method for this component (a List) is "enabled."

   Note – At this point, instead of selecting a object—or to override an object you already selected—you could toggle on the All visible windows option. This option would instruct JavaStar to verify all objects in all visible windows for your application under test. For this example, though, you need specific information on a single component, so leave this option unchecked.

   c. **In the Verification panel, click the** Customize **button.**
   The panel changes to show the object to verify and to prompt for a verification method.

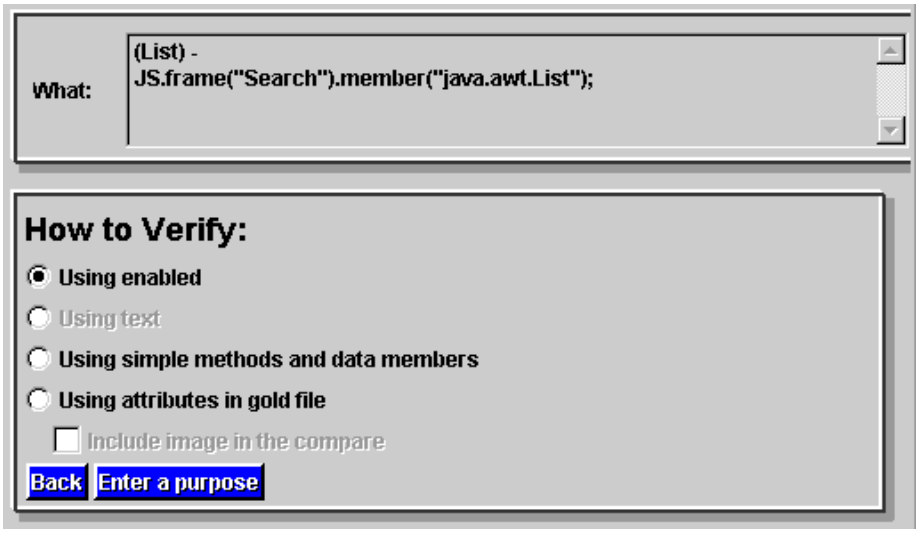


Figure 2-8 How to Verify panel

   d. **Select** Using Simple Methods and Data Members**.**

   e. **Click the button** Select Simple Methods and Data Members**.**
   The panel changes to show you a list of all available simple data members and methods, sorted by name.

f. **Scroll through the method list to locate `int getItemCount()`, and select it.**
To select the method, click anywhere on the line other than on the returns button. A black bar highlights the line to let you know it is selected. You can select multiple lines when you verify using simple methods and data members, and JavaStar will compare them all. For this exercise, though, you'll only compare `getItemCount()`.

If you want to preview the return value to make sure this is the correct method, click the returns button next to the name. This shows you the current return value based on your interaction with the application—in this example, the return value should be `new Integer(1)`.

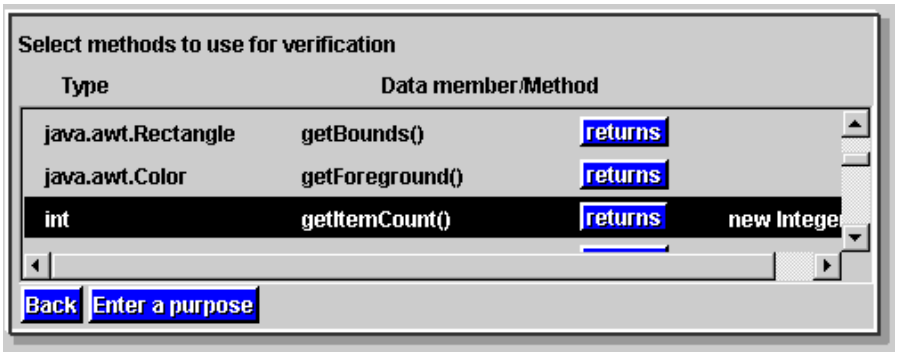


Figure 2-9 Select methods to use for verification

g. **Click the** Enter a purpose **button.**
The panel changes to show you the object you selected for comparison and the comparison method. It also prompts you for a string to identify the purpose for the comparison.

h. **Enter `Verify number of items found` in the purpose field.**

i. **Click** Insert verification into test**.**

6. **In the Record/Playback window, click** Continue **to resume recording.**
Note that the log shows this line of code added to the script:

```
JS.frame("Search").member("java.awt.List").verifyAnyMethod
(this,false,true,"getItemCount",new Integer(1),
"Verify number of items found");
```

This code uses the `verifyAnyMethod()` method from the JavaStar API library.

7. **Select the search result and click** View result**.**
The record is displayed in the Name Database main window.

8. **Set up a series of comparisons to ensure that the contents of each field for this record is correct.**
   Here you'll do verify operations instead of a synchronize, because even if one field fails the test during playback, you want the test to continue on and test the others.

   a. **Go back into verify mode.**
      Click Verify.

   b. **Click inside the** Name **text field to select it.**
      The field flashes to confirm your selection, and the code displays in the verification window. The verification panel shows "using text" as the default method of comparison, meaning that if you use the default, JavaStar will compare the text inside the field (not the field label).

   c. **Click** Use default**.**
      The panel changes to prompt you for a purpose.

   d. **In the purpose field, type** `Verify text entry`.

   e. **Click** Insert verification into test**.**
      This adds the verification code to your script:

      ```
      JS.frame("Name
      Database").member("namedb").member("java.awt.TextField"
      , 0).verify(this,"Count von Count", "Verify text
      entry");
      ```

      JavaStar returns you to the first Verification panel so you can specify more verify operations.

   f. **Set up verifications for the remaining text entry fields by clicking inside each field, clicking** Use default**, then clicking** Insert verification into test**.**
      The type of comparison and the Why string remain the same as long as you keep the Verification window, so you don't have to reset these for each field.

   g. **Click** Continue **to resume recording.**

9. **Close the Search window of the Name Database.**

What if you need to create a test before the application performs the correct computations?

You might need to create a test that exercises a feature of the product that, at the time you create the test, returns an incorrect value. If you insert a comparison (whether a verification or synchronization) you will be setting the test to compare to an incorrect value—meaning that as long as the tested

feature returns the wrong value, the test will pass. What can you do to create a test that checks for values that are not yet correct at the time you record the script?

In this case, you can manually edit the script and replace the string JavaStar uses for comparison with the correct text. Because this is a text comparison, this is relatively easy to do. You'll learn how to edit scripts in the chapter "Adding Parameters for Flexibility."

Ending Record Mode

1. **In the Name Database main window, click** Clear**.**
   This returns the application to a neutral state. It's a good idea to end your scripts in a state you can anticipate, especially if you plan on running one script after another, as in the case of a composed test.

2. **In the Record/Playback window, click** Stop**.**
   JavaStar saves the script as a `.java` file and compiles the code into a `.class` file. It writes these files to your work directory.

3. **Quit the Record/Playback window.**
   JavaStar will prompt you for confirmation that you want to quit. Once you respond affirmatively, JavaStar closes both the Record/Playback window and the application under test.

Running the Test

You can playback tests using the Run Test option of the main menu or using the Playback button in the Record/Playback window. However, if you had played your script back immediately after recording it, the results would be added to the same log file that contains your recording log. Mixing these results means less readability and a larger log, so, in general, playing back immediately after recording is reserved for when you want to debug your script.

When you use Run Test, JavaStar creates a new log file and launches the test application again, in a fresh state.

1. **Copy the original `sesame.db` to `test.db`, replacing the existing file.**
   This ensures that you start with a clean database. You can do this without exiting JavaStar. For example, on a UNIX system, you can copy the file in a shell. In Windows, you can copy the file in Windows Explorer.

   Note – You'll need to copy `sesame.db` over `test.db` repeatedly during the course of this tutorial, as you run and re-run tests. You might want to create a batch file or shell script that refreshes the database for you, to save some typing in subsequent exercises.

2. **From the JavaStar main menu, click** Run Test**.**
   The Run Test window opens.

3. **In the** Test name **field, type** `TestNameDB`**.**
   In this example, the script does not take any arguments so you can leave the Test args field blank.

4. **Click the** View **tab to see view options.**
   These three check boxes are where you determine whether you want to see the JavaStar windows along with the application window (necessary if you want to use JavaStar controls), just the application window, or neither one.

   Keep Show application and playback window checked so you can watch the progress through the script. If you run a script later on and don't care to use any controls, you can choose the Don't show playback window option. The last option, Don't show application and playback window is more useful when you run tests from the command line. When you're running the JavaStar GUI, you'll need to use the Status Monitor to find out when your script finishes running.

5. **Click the** Advanced **tab to see more options.**
   The advanced options include any additional classpath you need to use for this test run, Java arguments you want to insert, the log file name, the working and results directory, and the JST path. You can leave these with their default values for this test run.

6. **Click** Start**.**
   Because you did not specify a log file name, JavaStar automatically opens a log file using the format *testname*`.log` (which, in this case, translates to `TestNameDB.log`).

   JavaStar opens the Record / Playback window and launches the test application. In contrast with how the Record / Playback window looked while you were recording, it now shows two text panels—one for the JavaStar script that is executing, and the other for the log file. As the script runs, the code scrolls through the upper panel. The current line number shows in the Location information, and this line is also highlighted in the text panel. See Figure 2-10.

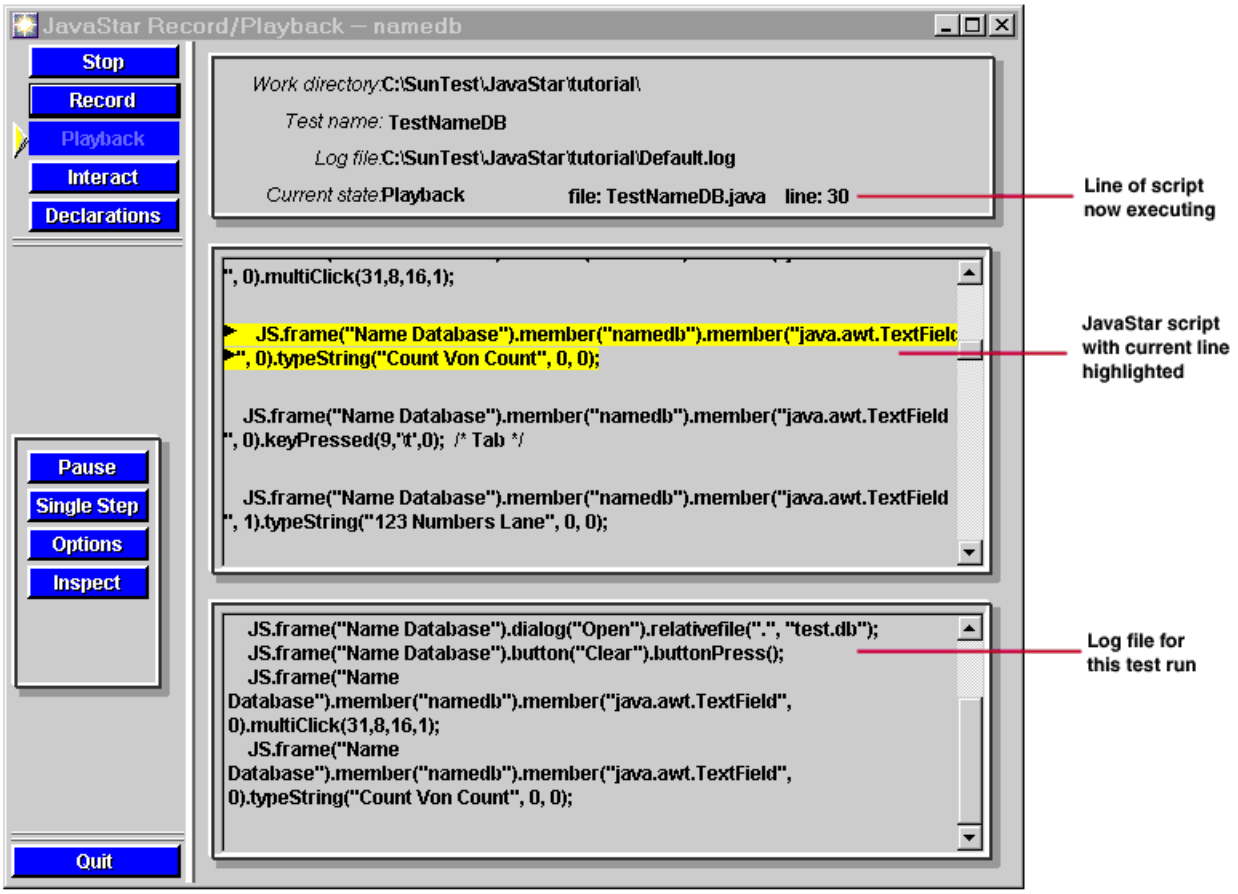


Figure 2-10 Record / Playback window during playback

At the end of the test, the Record / Playback window stays open, and you can see summary information in the log file panel. See Figure 2-11.

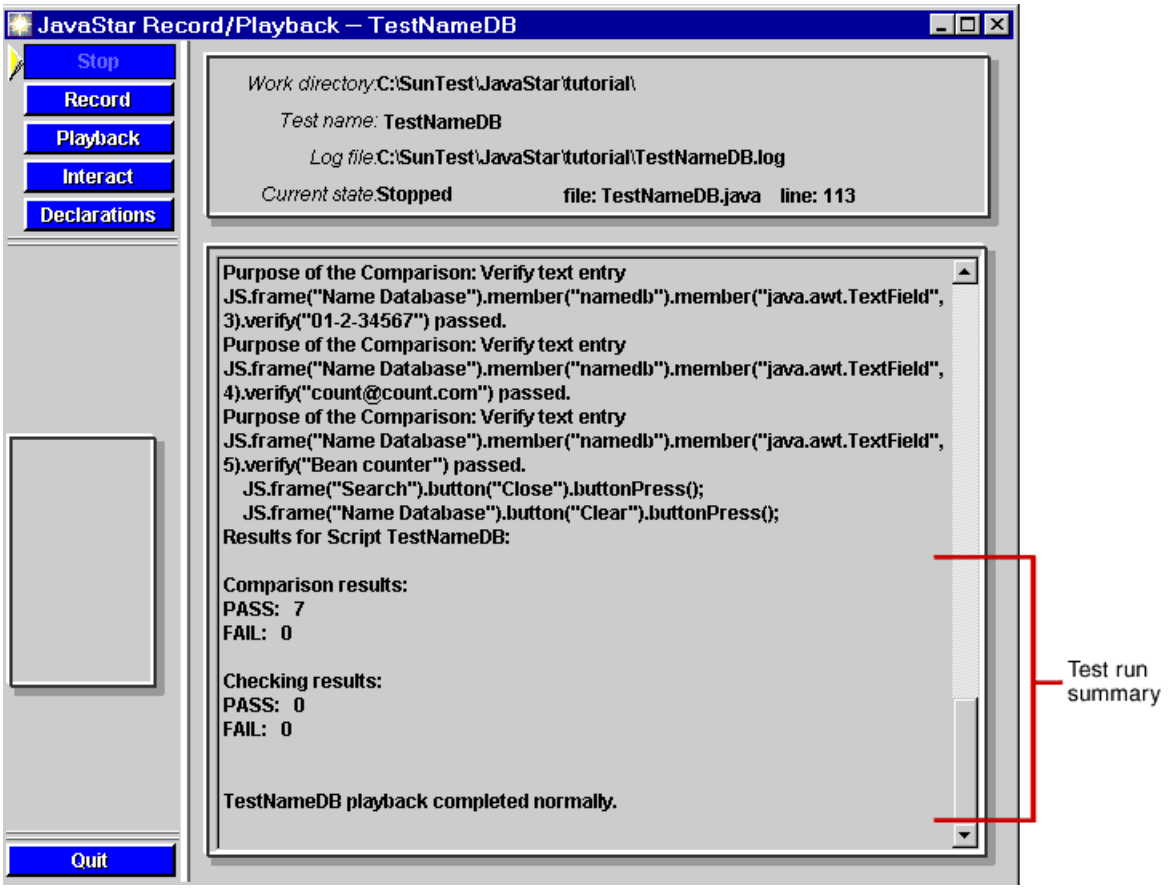


Figure 2-11 Record / Playback window with test run summary

7. **When the script finishes, quit the Record/Playback window.**

Viewing the Results

The last thing to do is to examine the results of the test run. This section shows you how to view and filter the results of a single script.

1. **From the JavaStar main menu, click** View Test Results**.**
   The Results Viewer opens. If you just ran the `TestNameDB` script, the Results Viewer should have this script already loaded.

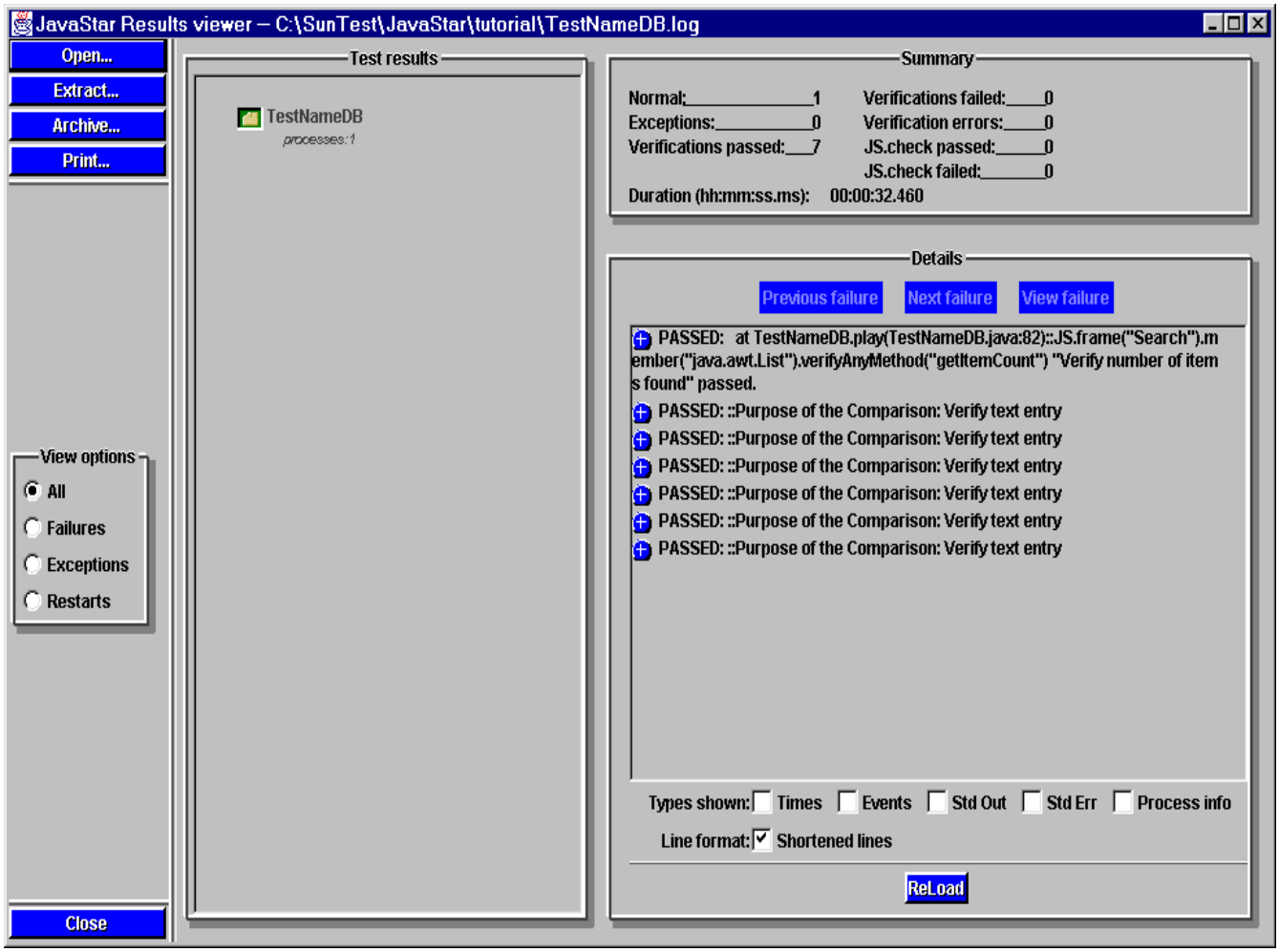


Figure 2-12 Results Viewer

The Summary for `TestNameDB` should show 1 normal condition and 7 verifications that passed. The normal condition tells you that the script itself ended normally—the synchronized comparison did not throw an exception. If this was a test composed of multiple scripts, you might see several normal conditions and some exception conditions in the summary, because JavaStar tracks the end state of each script.

A verification error can occur in a number of circumstances—such as when JavaStar can't find an auxilary file it needs for the test—and you should check the log file for a specific message. JS.check pass / fail conditions reflect the results of custom code that you can insert (using the JavaStar API check method of the JS class) to track your own conditions. You did not add any custom code to the test you just ran, so all the values are 0.

Note – The chapter "Using the JavaStar API" discusses the JavaStar API and shows how you can incorporate API methods into your scripts.

2. **Examine the log file entries in the Details panel.**
   Here you can see the comparisons this test executed. The Results Viewer displays these in brief form, because the Shortened lines option is selected, by default. The + sign next to each event means that you can see more information by clicking on the line to expand it. Experiment with expanding and contracting this information, or turning off Shortened lines and reloading the file to see complete information for each event.

3. **To see process information, toggle the** Process Information **checkbox on, then click** ReLoad**.**
   You can find the Process Information checkbox at the bottom of the Details panel.

   By default, the Results Viewer filters out:

   - Event details
   - Timestamps
   - Process information

   Event information is the logging of each event in the script—this information takes up a lot of space and isn't always useful. Timestamp information shows you when each event executed. Process information shows when the test started, what the name of the test is, what playback options were in effect for this test run, what directories you used, and the environment under which you ran the test.

   Whenever you change checkbox settings, you need to reload the log file for the new filter settings to take effect.

4. **When you're done examining the results, click** Close**.**

What These Results Didn't Illustrate

This example of the Results Viewer doesn't show the full value of the tool, mainly because it does not demonstrate results for nested JST (JavaStar Test, or composed test) files, and there were no failures in this test.

In later chapters, you'll see how JavaStar shows results for JSTs within an expandable/collapsible tree, and the summary information will be more helpful. Comparison failures are harder to demonstrate, because you'd have to edit the script to force it to fail in order to move through failures using the Next Failure button. Refer to the chapter "Viewing and Analyzing Results"in the *JavaStar User's Guide* for details on viewing failures and updating comparisons.

Summary

This chapter covered the basic features of JavaStar. After doing the exercise that follows (creating a script to test the Search window) you should feel fairly comfortable recording and playing back scripts, as well as viewing their results.

The next chapter describes the JavaStar test model, and why it makes sense to design tests using the principles of this model.

Exercise: Testing the Names Window

Now that you've created a script that tests the Search window, you're ready to do the same with the Names window.

Instructions

Write, run, and view the results for a script that:

1. Loads the test database
2. Displays the names of records in the Names list
3. Selects and displays the `Count von Count` record in the main window
4. Verifies that the text for each field is correct.

Solution

To see a solution for this exercise:

1. **Copy `TestNames.java` and `TestNames.class` from the `\JavaStar\tutorial\getStarted` directory into your work directory.**
2. **Use** Run Test **to playback the test and** View Results **to examine the log.**

Moving to the JavaStar Model 3

This chapter introduces you to the JavaStar test model and describes how this model effects the way you create your scripts.

Topics:

About the Model

Now that you're familiar with the basic features of JavaStar, you can focus on learning the test model that helps you get more power out of JavaStar. The "Getting Started with JavaStar" chapter didn't focus on design, just mechanics. After following the lesson, you ended up with a long test (`TestNameDB`) which, while exercising certain features of the Name Database, isn't particularly useful for building additional tests.

To introduce you to the JavaStar test model, this tutorial will step you through the re-design process for `TestNameDB`. This way, you can see the impact the design has on a test that was originally not very flexible. The chapters that follow contain lessons that demonstrate:

- Using declaration files to make GUI maintenance easier
- Making scripts modular so they can be used in multiple tests
- Passing data as arguments instead of hard-coding it

While three chapters of lessons may seem like a lot, once you're familiar with the model, you'll find you can develop the initial tests quickly. Tests that build on top of your initial tests will be much faster to create, as well, because you'll re-use existing modules.

Deficiencies of the Previous Approach

After recording `TestNameDB`, you're probably all too familiar with how tedious it can be to record a long script from start to finish. This may be especially true if you encountered a problem and had to do the script over. It's a painstaking process that yields an awkward test.

Some of the deficiencies of `TestNameDB` are:

- There is no abstraction from the interface of the application under test. If one or more components of the application under test change, you have to update every reference to the modified components.
- The script combines several tests into a single script (loading a database, adding a new record, and verifying the record using the search window). This:
  - Makes the script difficult to record without making an error.
  - Limits re-use potential, because you can only re-use this script if you want to test these three functions in this same order. Otherwise, the script isn't useful beyond one test run per application release.
  - Outputs results that are long and hard to navigate. JavaStar provides a feature that jumps ahead to each failure, but if you're looking for something other than failure, wading through these results is tough work.
- The script uses hard-coded data, not allowing you to test the same features with different data—unless you want to maintain copies of the script that use different data.

Fortunately, each of these deficiencies can be overcome with a change of approach.

Making Tests More Robust

You can make your tests easier to create and maintain by:

- Writing Tests Using Declaration Files
- Making Tests Modular
- Passing Data as Arguments

Writing Tests Using Declaration Files

Using JavaStar, you can generate declaration files for your application's GUI in a matter of minutes. These files contain the declarations for each GUI component. Once you've created the files, you can record scripts that reference the declaration files.

What this does is provide you with a level of abstraction. You can change the names of the declarations (possibly make them more meaningful for test purposes) and incorporate the new names into the tests you record. If the user interface changes—for example, if a component changes type, name, or position—you can update the declarations file to reflect the change, and your tests will automatically reference the new information.

When you use declaration files, your tests require less direct maintenance, resulting in a significant time savings.

Making Tests Modular

Writing tests as small modules that exercise a specific feature—perhaps a single component—is a lot like creating building blocks. Alone, the scripts don't do much, but when combined with other scripts, you can create many combinations.

In the case of `TestNameDB`, there are many options for breaking the test down into smaller scripts. As it stands, the test performs the tasks of loading a database, adding a record, and verifying the record through a search mechanism. Each of these tasks is made up of smaller actions, such as entering data or clearing the display. These tasks and actions, when separated into different scripts, create a strong base of re-usable test material.

To take advantage of smaller scripts, JavaStar provides a Test Composer. Using this feature, you can link scripts together, defining the order and test conditions for execution. You can also define certain scripts as restart nodes for error recovery. The files you create in the Test Composer, called JavaStar Test (JST) files can be nested to build more sophisticated test suites.

Passing Data as Arguments

Because you can perform the same test with different data and get different results, it makes sense to keep your tests as independent of the data as possible, and make it easy to change the values you feed them. With JavaStar, you can do this easily by replacing data in the scripts with references to arguments, then passing the data to a script using the parameter feature of the Test Composer. The source of the parameter can be a parameter passed into the JST, a constant defined in the JST, or a value read from a property file.

Using arguments is an important part of making re-usable test modules.

Summary

This chapter introduced you to the basics of the JavaStar model. The lessons that follow demonstrate how to put this model into action with the test you've already created. While the tutorial provides you with files for each chapter so that you can skip around, it is probably easiest to understand the model by following the evolution through all three chapters.

Generating Declarations 4

This chapter shows you how to generate declarations from your test application and use these in your test to create a more robust test suite.

Topics:

About this Lesson

For automated tests to really save you time, they need to be easy to maintain as your program under test changes. If you have to tinker with each script—or re-record it—because one component changes, you're likely to wonder if you're really saving time. This chapter shows you how to generate declarations—a feature JavaStar provides to make it possible to update a whole collection of tests by simply editing one file they all reference.

Generating declaration files is best done as the first step in creating your scripts. JavaStar includes a feature for automatically incorporating declaration files while recording, but for it to work, you must generate the declaration files before recording the scripts.

As an introduction, this lesson shows you the impact simple GUI changes can have on your application under test, then guides you through the process of modifying your test to be more dynamic. Upon completing this chapter, you should know how to:

- Identify test-run problems caused by changes to the application under test.
- Generate declarations for an application under test.
- Modify declarations to create abstracted component names.
- Record a script that uses declaration files.

Setting Up for this Lesson

For this lesson, you're going to start with a different version of the Name Database application. If you look in the `examples` subdirectory of the javastar directory, you'll see that there are three name database directories: `namedb1`, `namedb2`, and `namedb3`.

Up to this point, you've worked with `namedb1`. This version corresponds to an initial development version. The `namedb2` version represents a development update to `namedb1`, with GUI modifications. The tutorial doesn't address `namedb3`.

1. **Copy the original `sesame.db` over `test.db`.**
2. **If you did not do the exercises in the chapter "**Getting Started with JavaStar**", or if you no longer have these tests in the your directory, copy the contents of `\tutorial\gettingStarted` to `\tutorial`.**
3. **Launch JavaStar.**
   The JavaStar main screen opens, loading the last project file you used. If this is not `namedb.prj, the file you created in the chapter` "Getting Started with JavaStar," open `namedb.prj` before continuing.
4. **Create a project file for the namedb2 application.**
   You can do this by modifying the `namedb.prj` file:
   a. **Click the** App **tab to bring the panel forward.**
   b. **Delete the contents of the** Classpath **field.**
      This prevents JavaStar from appending a classpath to the existing one when you select a new application.
   c. **Click the** Browse **button for the** Class **field.**
   d. **In the file dialog, navigate to the `\javastar\examples\namedb2` directory and double-click the `namedb.class` file.**
   e. **From the buttons that appear along the bottom of the screen, choose** Save As...**.**
   f. **Save the file as namedb2.prj.**

Debugging Test Run Errors Caused by GUI Changes

Declarations address problems that arise when the components in the test program's GUI change during development.

1. **From the JavaStar main menu, click** Run test**.**
2. **For** Test name**, type `TestNameDB`.**

3. **Start the test.**
   Watch the Name Database main window as the test executes. It fills in each field with the test data until it gets to the Phone field—then it skips to the Email field and enters the phone information there (see Figure 4-1 on page 43).

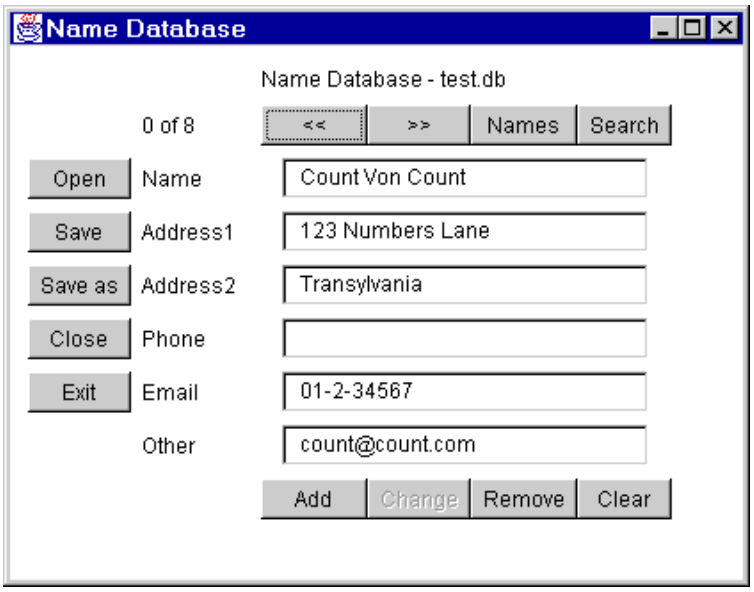


Figure 4-1 Name Database main window during playback

Once the test fills in the Other field, it attempts to Tab to the next field. Here is where it pauses, then (according to whatever timeout value you have selected) the script ends. The Record / Playback window shows the exception thrown by the script.

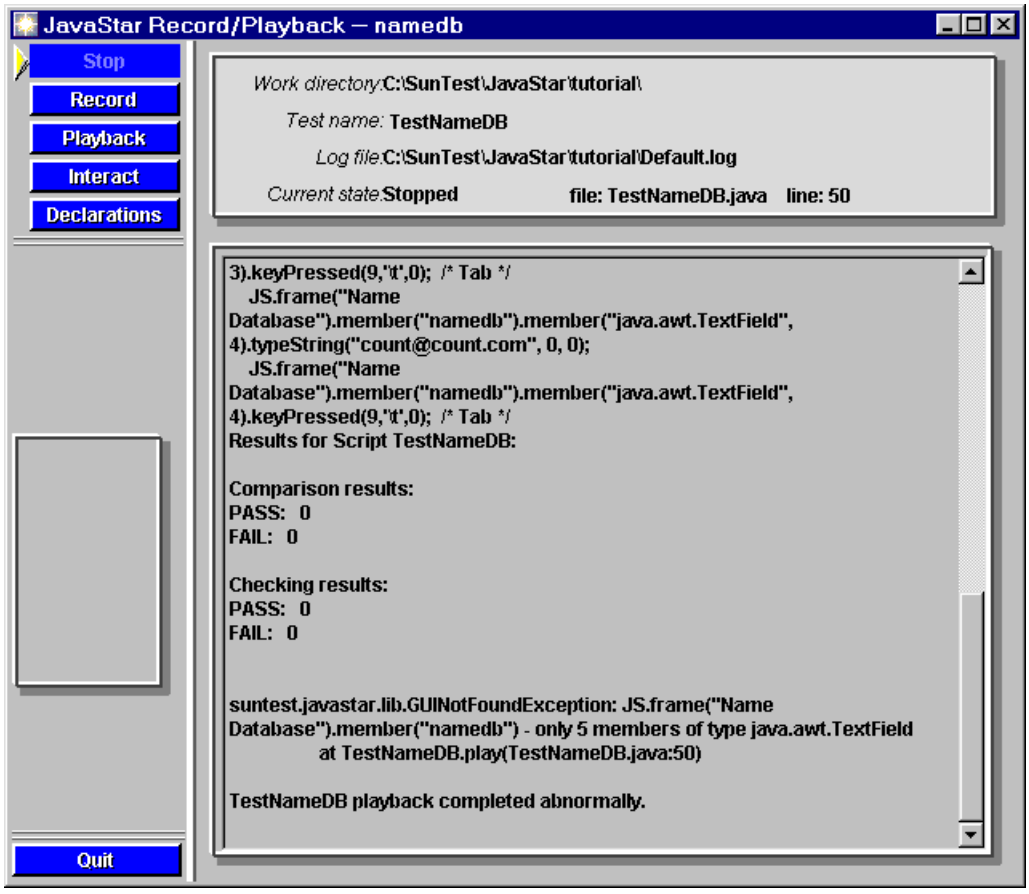


Figure 4-2 Record / Playback window with exception information

4. Investigate the problem.

The Record / Playback window provides the following information on the failure:

```
suntest.javastar.lib.GUINotFoundException: JS.frame("Name
Database").member("namedb") - only 5 members of type
java.awt.TextField
   at TestNameDB.play(TestNameDB.java:50)

TestNameDB playback completed abnormally.
```

From here you can see that JavaStar found only five TextField components, not the six TextFields that the script expected. This tells you where the script failed, but not why it passed up the Phone field. Your next step is to find out the type of the Phone field.

5. **Inspect the GUI.**
   You can examine any component of your GUI without leaving Record / Playback mode and without having to examine the script. At this point, while your test has stopped, the Record / Playback window is still open. To examine the GUI:

   a. **Click** Interact**.**
      This allows you to work directly with the application without recording or playing back a script.

   b. **Click the** Inspect **button.**
      This button appears in the lower left button panel. It is enabled only after you select Interact. Clicking Inspect opens the "Select an object to inspect" panel in the right side of the window.

   c. **Click in the** Phone **field of the Name Database.**
      You may need to drag the test program to one side to see both windows at once. When you click in the text field, the field should flash to confirm your selection. The code for the selection also appears in the Select an object to inspect panel.

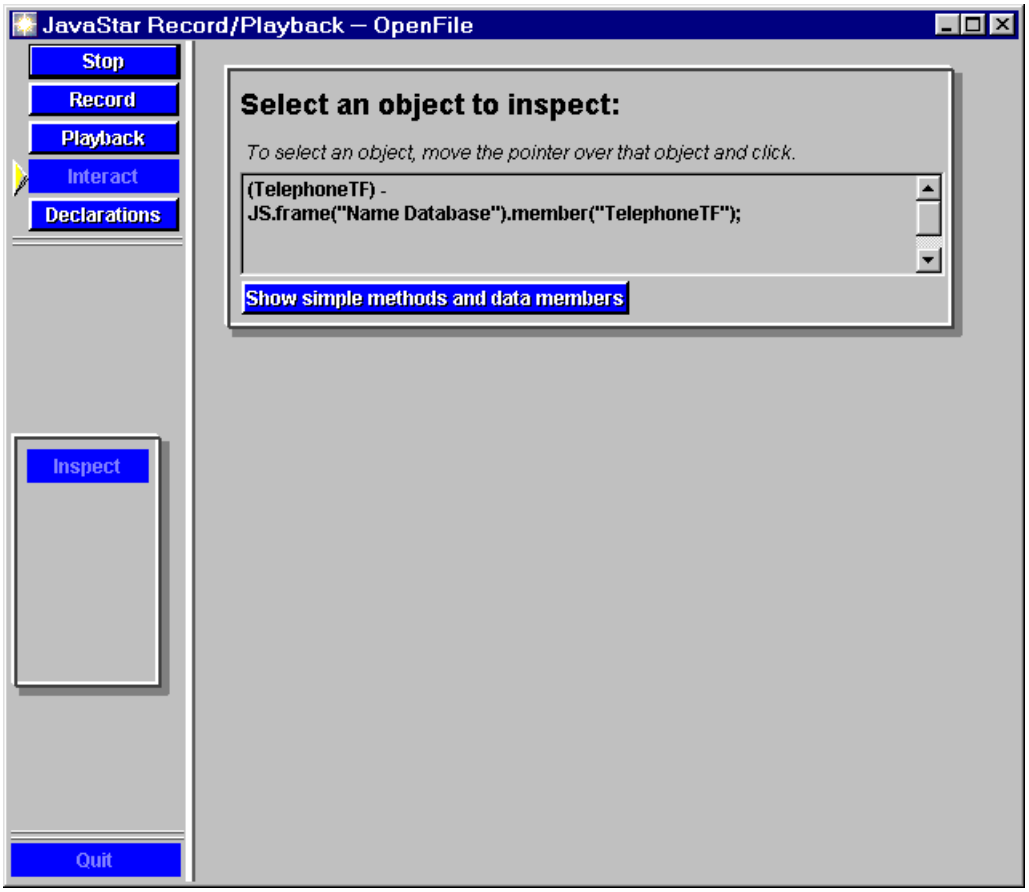


Figure 4-3 Select for Inspection window with Name field selected

This code shows:

```
(TelephoneTF) -
JS.frame("Name Database").member("TelephoneTF");
```

This code reveals the problem: The `TestNameDB` script is trying to enter the phone data into a text field, but the application has been modified to change the Phone TextField to a component of type TelephoneTF. Unable to enter a string into a TelephoneTF, the script skips to the next TextField (Email) and enters the text field there. By the time it attempts to enter data for the Other field, no more text fields remain.

To fix this problem, it seems you have to examine every reference to the text fields in this window and edit the component references to accommodate the type change. If you had multiple tests, that could be a lot of work. However, if you had generated component declarations for the application, you'd only need to change one file to update all scripts to use the correct types.

6. **Click** Stop **to end inspect mode.**

7. **Quit the Record/Playback window.**

Designing a Suite to Use Generated Declarations

This case illustrates one problem that declarations can handle: making quick updates to accommodate changes to a component type. However, this isn't the only case where declarations are useful. For example, in the case of buttons—where JavaStar references the button using the actual label—you would have a problem if you renamed the button. Your tests would fail and you'd be left with the task of updating all scripts to use the new button label.

Here's how generated declarations provide a solution:

1. At one time, you generate a declarations file for each window of your application. This step is as simple as pointing and clicking—JavaStar does all the work. You can store all the declarations files for a test program in a package for convenience.

2. You edit the declarations file to use abstracted names for the components. Instead of referring to a field by a number, you can reference it by name.

3. You reference the declaration files at the time you record scripts, so that JavaStar automatically imports the declarations into the test it creates and records component references using the new abstracted names.

4. When a component changes in the application, you open the file where the component declaration is stored, edit it to reflect the change, and then save and compile. All scripts that reference that component now automatically reference the new information.

Generating Declarations

1. **From the JavaStar main menu, select** Create Test Script**.**

2. **Enter** namedb **as the** Class **name.**

3. **Click** Start**.**

4. **Once the Record/Playback window is open, click the** Declarations **button.** This opens Generate Declarations instruction dialog, giving you information on how to select components. You can leave this dialog open or dismiss it by clicking OK.

5. **Select the main window of the Name Database as your component.** Move your cursor over the main window and press Ctrl-Alt-F10.

   A new Generate Declarations window opens, prompting you for a package name and a class name.

6. **Type `NameData` in the** Package **field.**
   You're not required to enter a package name. If you don't define a name, JavaStar stores the class containing the declarations in the working directory. If you do define a package, JavaStar stores the declaration classes within a directory named for your package, stored a subdirectory of your working directory.

7. **Type `MainWin` in the** Class **field and click** OK**.**
   The log file in the Record / Playback window (see Figure 4-4) shows the components JavaStar writes out to the class you specified. Generating declarations for a window is that simple.

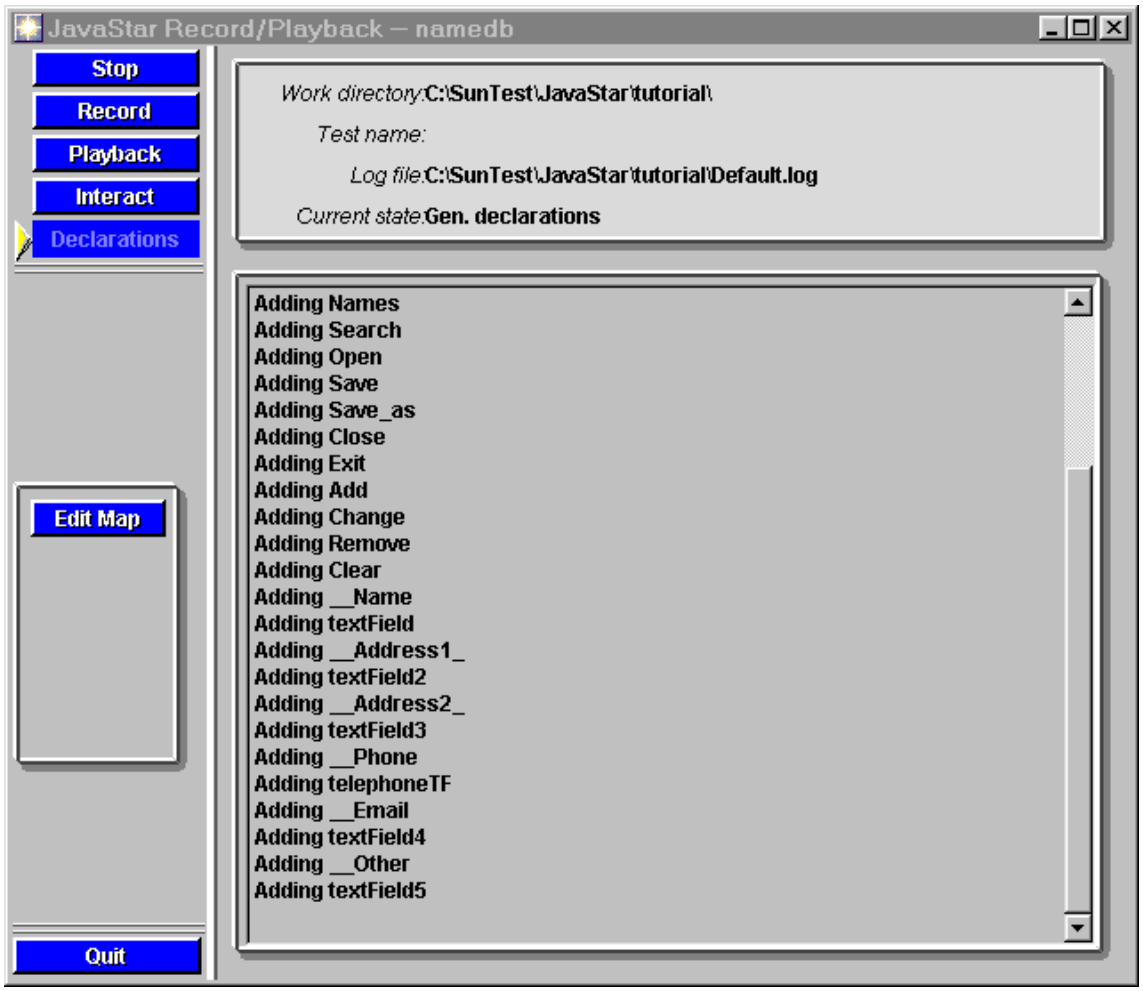


Figure 4-4 Record / Playback window while generating declarations

8. **Generate declarations for the Names and Search windows.**
   To do this:

   a. **Click the Names button to open the window.**
      Put your cursor over any part of the Names window and press Ctrl-Alt-F10.

   b. **In the Generate Declarations window, keep the package name the same and type `NamesWin` for the class name. Click OK.**

   c. **Repeat this process for the Search window, saving the declarations to `SearchWin`.**

9. **Stop the generate declarations process.**
   In the Record / Playback window, click Stop.

10. **Quit the Record/Playback window.**

Modifying Declarations to Use Abstracted Names

The code for the Name Database application does not define names for the text fields. In lieu of names, JavaStar refers to them by their order—for example, `textField1`, `textField2`, and `textField3`.

Ideally, the code would name the components. However, if the code you're testing uses default names for text fields, you can add your own by providing abstracted names in the declarations file. You do not have to do this in order to gain benefits from using declarations, but this provides an extra bonus. Using names makes scripts easier to modify, and logs easier to decipher. Updating the declarations to reflect GUI modifications is also more straightforward.

> ***Note –*** When you change names in a declarations file, be sure to search to find any additional references to the original name, and update these, too.

Whether you use abstracted names or not, you always need to save and compile declarations after generating them. JavaStar creates on the `.java` files when you generate declarations, not the `.class` files.

Editing MainWin Declarations

1. **From the JavaStar main menu, select** Edit Test Script**.**

2. **Load the `MainWin.java` declarations file.**
   You'll find this in the `NameData` subdirectory of `tutorial`.

3. **Locate the declaration for the first text field.**
   Scroll down through the file until you find the line:

```
/* TextField */
public static JSComponent textField(){
return Namedb().member("java.awt.TextField", 0);
}
```

Note – The code to change and the replacement code is set in bold type for these examples. It won't appear in bold in the Script Editor window.

4. **Replace the default textField name with a field-specific identifier.** Change `textField()` to `nameTextField()`. Your code should read:

```
/* TextField */
public static JSComponent nameTextField(){
return Namedb().member("java.awt.TextField", 0);
}
```

5. **Edit the declarations for the other four text fields.**

   a. **For the second text field, replace `textField2` with `address1TextField`.**
   Replace:

```
/* TextField */
public static JSComponent textField2(){
return Namedb().member("java.awt.TextField", 1);
}
```

   With:

```
/* TextField */
public static JSComponent address1TextField(){
return Namedb().member("java.awt.TextField", 1);
}
```

   Note – When replacing "`textField`", don't forget to include the number that follows it.

   b. **For the third text field, replace `textField3` with `address2TextField`.**
   Replace:

```
/* TextField */
public static JSComponent textField3(){
return Namedb().member("java.awt.TextField", 2);
}
```

   With:

```
/* TextField */
public static JSComponent address2TextField(){
return Namedb().member("java.awt.TextField", 2);
}
```

   c. **For the fourth text field, replace `textField4` with `emailTextField`.**
   Remember that the Telephone field is no longer a TextField object, so we're not editing that declaration.

   Replace:

```
/* TextField */
public static JSComponent textField4(){
return Namedb().member("java.awt.TextField", 3);
}
```

With:

```
/* TextField */
public static JSComponent emailTextField(){
return Namedb().member("java.awt.TextField", 3);
}
```

d. For the fifth text field, replace `textField5` with `otherTextField`.
Replace:

```
/* TextField */
public static JSComponent textField5(){
return Namedb().member("java.awt.TextField", 4);
}
```

With:

```
/* TextField */
public static JSComponent otherTextField(){
return Namedb().member("java.awt.TextField", 4);
}
```

6. **Save and compile.**
A dialog is displayed when the compile succeeds.

Editing SearchWin Declarations

1. **Load `SearchWin.java` into the Script Editor.**

2. **Scroll down to locate the declaration for the textField component.**
This is the contains strings field:

```
/* TextField */
public static JSComponent textField(){
return SearchPop().member("java.awt.TextField");
}
```

3. **Change the component name to a field-specific identifier.**
Change `textField()` to `containsStrTextField()`.

```
/* TextField */
public static JSComponent containsStrTextField(){
return SearchPop().member("java.awt.TextField");
}
```

4. **Save and compile.**

Editing NamesWin Declarations

The Names window doesn't contain any text field components, so you don't need to edit the contents of `NamesWin`. You do, however, need to compile the `.java` file.

1. **Load the `NamesWin.java` declarations file into the Script Editor.**
2. **Save and compile the script.**
3. **Close the Script Editor.**

Recording New Scripts that Use Declaration Files

For this part of the lesson you'll create a small script that exercises several components. Because you'll be re-designing `TestNameDB` in the next lesson, this test serves as an example and is not one you'll keep in the revised test suite. *All the tests you create from this point in the tutorial will use the declaration files.*

> ***Note –*** If you want to use declaration files with existing tests, you can modify the code manually to incorporate them. For instructions, see the chapter, "Generating and Using Declarations" in the *JavaStar User's Manual*.

1. **From the main menu, choose** Create Test Script**.**
2. **Start the `namedb` application.**
3. **Click** Record**.**
4. **In the Create script field, enter `SimpleAdd`.**
5. **To the right of the** Record with map files **field, click the** Map list... **button.** The Select Map Classes dialog is displayed. *Map* is another name for declaration files.
6. **Using the file panel in the left portion of the window, navigate to and expand the `\tutorial\NameData` directory.**
7. **Select each of the declaration files and add it to the list.**
   a. **Click on `MainWin.class`.**
      The class name appears in the upper right panel as the current item.
   b. **Click the** Add to list **button.**
      `MainWin.class` is added to the list in the right panel and to the field at the bottom of the window.
   c. **Repeat** Step a **and** Step b **for `NamesWin.class` and `SearchWin.class`.**
8. **In the Select Map dialog, click** OK**.**

9. **In the Record test script dialog, click** OK**.**

10. **In the Name Database window, open the `test.db` file.**
    Click Open, locate and select `test.db`, and click Open again.

11. **In the same window, click** Clear**.**

12. **Enter a new record into the name database.**
    Type data into each field of the record and click Add.

13. **In the Record/Playback window, click** Stop**.**

14. **Examine the `play()` method of the script (shown in the log panel to the right of the Record/Playback window) to see the declarations.**
    Though the mouse coordinates may differ in your code, the beginning lines of `play()` should look similar to this:

```
public void play(String[] args) throws Throwable {

   NameData.MainWin.Open().buttonPress();

   NameData.MainWin.frame().dialog("Open").relativefile(".",
"test.db");

   NameData.MainWin.Clear().buttonPress();

   NameData.MainWin.nameTextField().mousePressed(27,11,16);

   NameData.MainWin.nameTextField().mouseDragged(25,10,0);

   NameData.MainWin.nameTextField().mouseReleased(25,10,16);

   NameData.MainWin.nameTextField().typeString("Alix", 0, 0);

   NameData.MainWin.nameTextField().keyPressed(9,'\t',0); /* Tab
*/
```

    The same test recorded without using the declaration files would look like this:

```
public void play(String[] args) throws Throwable {
   JS.frame("Name Database").button("Open").buttonPress();

   JS.frame("Name Database").dialog("Open").relativefile(".",
"test.db");

   JS.frame("Name Database").button("Clear").buttonPress();

   JS.frame("Name
Database").member("namedb").member("java.awt.TextField",
0).mousePressed(21,5,16);

   JS.frame("Name
```

```
Database").member("namedb").member("java.awt.TextField",
0).mouseReleased(21,6,16);

   JS.frame("Name
Database").member("namedb").member("java.awt.TextField",
0).typeString("Alix", 0, 0);

   JS.frame("Name
Database").member("namedb").member("java.awt.TextField",
0).keyPressed(9,'\t',0);  /* Tab */
```

15. **Quit the Record/Playback window and confirm that you want to end all processes.**
    Click Quit, then click OK in the confirmation dialog.

Summary

How you use declarations is up to you—you can use them just to provide control over components that change during the course of development, or you can also add a level of abstraction that makes tests more readable. Whatever you decide to do, using declarations as a regular part of every test suite makes it easier to change and maintain tests as the application under test evolves.

The next chapter shows you how to re-record the test conditions in `TestNameDB` as re-usable modules.

Using a Modular Approach 5

This chapter introduces the JavaStar test model by guiding you through the analysis and definition of a modular test.

This lesson analyzes the work from the previous lesson to see how the test can continue to be improved to follow the JavaStar model. In the process, you'll be introduced to the Test Composer, a key JavaStar feature supporting modular test development.

This chapter assumes you have basic familiarity with the JavaStar recording controls.

After completing this lesson, you should be able to:

- Analyze a test to see how it would best benefit from a modular approach
- Define a JavaStar Test (JST) using the Test Composer
- Record small scripts that work together in larger tests
- Navigate through the results of a JST in the Results Viewer

Topics:

Setting Up for This Lesson

If you are continuing from the previous lesson, all you need to do to prepare for this lesson is to:

1. **Copy the original** `sesame.db` **over** `test.db`**.**
   This returns the database to a clean state, without the records added by recent test runs.

2. **In the JavaStar main screen, click on the** Mapping **tab of Project Settings.**
   The mapping options panel moves to the forefront..

3. **Enter the declaration filenames into the** Declaration classes **field.**
   If you are working in a Windows environment, use a semi-colon as a delimeter between filenames. For a UNIX environment, use a colon.

   On Windows 95, for this example, you enter:

   ```
   NameData.MainWin;NameData.NamesWin;NameData.SearchWin
   ```

4. **In the Declaration classpath field, type the path to the declaration files.**
   For example, if you are running under Windows and you used the default setup for the JavaStar installation, you would enter a path of:

   ```
   c:\suntest\javastar\tutorial
   ```

   Note – Because NameData is a package, do not point directly to the NameData directory—if you do, JavaStar will try to find a NameData package within the NameData directory, and will return compile errors for your scripts.

5. **Click the** Save **button to save changes.**
   JavaStar will now provide your list of declaration files into the Record with map files field by default, so you don't have to enter it every time you record a new script.

If you have not done the previous lessons:

1. **If you have not already done so, follow the instructions for** Setting up JavaStar **as described in the chapter "**Generating Declarations**."**

2. **If you did not do the lesson in chapter** Generating Declarations**, then**:

   a. **Create a directory within the** `tutorial` **directory called** `NameData`**.**

   b. **Copy the contents of the** `\javastar\tutorial\NameDataDecls` **into** `\javastar\tutorial\NameData`**.**

3. **Copy the contents of the** `\javastar\tutorial\declarations` **directory to** `\javastar\tutorial`**.**

Improving Your Tests with a Modular Approach

The script in **“**Getting Started with JavaStar” tested a variety of functions in the Name Database:

- Loading a database
- Adding a new record
- Verifying the new record using the Search window

These are valid tests, but, as mentioned in the chapter “Moving to the JavaStar Model,” putting so many tests into a single script makes the test difficult to record and even more difficult to re-use. This lesson begins by examining `TestNameDB` closely and seeing where it can be broken down into modules, represented by individual scripts.

Looking at the Name Database Example

If you look at the last test closely, you’ll see that it:

- Loaded a file
  - Opened the file dialog, loaded the file
  - Verified the database name loaded
- Added records
  - Cleared the display
  - Entered field data
  - Clicked the Add button
- Cleared the display
- Verified the name in the Search window
  - Opened the Search window
  - Selected search criteria
  - Entered search data
  - Clicked search
  - Selected search result
  - Verified the number of records returned
  - Viewed the result in the main window
  - Verified the contents of the record as displayed
  - Closed the search window
- Cleared the display

Loading files, adding records, and verifying entries are likely to be operations that you do repeatedly in the test suite. By breaking these operations out into different scripts, you create more manageable files that you can then re-assemble in different configurations, based on how you want to test the application.

In addition, by breaking the tests down further to the second level of bullets, you can develop a number of test modules. You'll have more pieces to work with, but also much more flexibility in how you can combine them to form new tests.

For this lesson, you'll start re-defining the tests for Name Database, starting with a high-level definition in the Test Composer.

Using the Test Composer

The JavaStar Test Composer provides you with a graphical, point-and-click way to define and assemble scripts into JavaStar Test (JST) files.

JavaStar refers to each module in a JST file as a test *node.* Nodes are either scripts or other JST files that execute under the conditions you define.

Nodes are connected by arrows that define their relationship to each other in the overall test graph. Connections appear as either green normal arrows or red exception arrows. These connection types—normal and exception—indicate the conditions under which the test pointed to will run.

For example, if you have a node called PopulateDB that adds a series of records to your database, and if those records are needed for the rest of the tests to continue properly, you probably want to define a normal and an exception condition.

By drawing a green arrow from PopulateDB to ChangeRecord (the first script of the rest of the test) you specify that JavaStar will execute ChangeRecord only if PopulateDB ends normally. If you then draw a red exception arrow from PopulateDB to another test node call LoadPopulatedDB—one that loads a database that already has these records included—you specify that if PopulateDB ends with an exception, the test should run LoadPopulatedDB.

You can then draw a normal, green arrow from LoadPopulatedDB to ChangeRecord, and your test has built-in recovery.

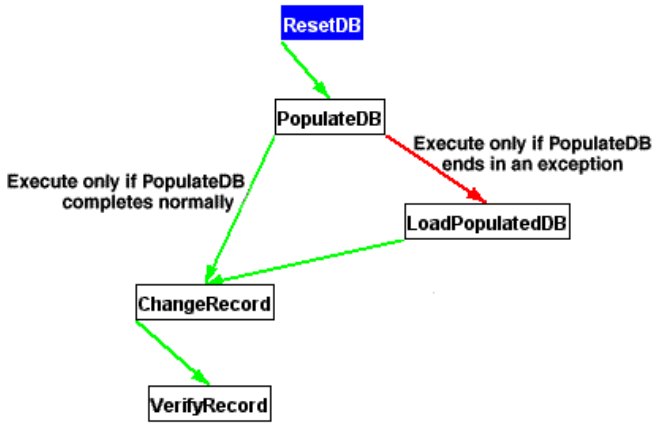


Figure 5-1 A sample JST

When you do *not* define an exception node—for example, if you defined PopulateDB to have only a normal connection to ChangeRecord—JavaStar will terminate playback when the exception is thrown. If you set up a long series of tests to run overnight, and you did not catch a recoverable condition, you've then lost valuable test time.

Defining test conditions is just one of the ways the Test Composer helps you create powerful tests. Using this tool, you can also define any node as a *restart node*. Restart nodes specify that the application will restart before executing the node. This is particularly helpful for recovery nodes.

The Test Composer also provides many controls for passing parameters to individual nodes and to the JST file itself. This dramatically increases the flexibility of your tests, making it easier to create one test that works in a variety of situations, all defined by parameters you specify in the JST file or at the time you run the test. This feature is explored in detail in the next chapter.

Composing Tests

One of the handy features of the Test Composer is that you can compose tests before you write the scripts that make up the test. Defining the JST files first can help you determine which scripts you need to write, whether they will be re-usable, and, later on, when you can re-use a test by adding parameters.

For this part of the lesson, you're going to use the Test Composer to compose three JavaStar Tests:

- Entering a Record
- Verifying Search Results
- The Acceptance Test

The first two tests represent the major areas covered by the script you wrote in the previous chapter. The last JST, the Acceptance test, combines the two previous tests with an OpenFile script. This creates a test that functions just like the long script, but is composed of modules you can re-use in other configurations.

You create the Acceptance test last because, while you can define a JST before you've written the scripts that it references, you can't reference another JST that hasn't yet been written.

Entering a Record

This first JST handles record entry. As you noted when you reviewed the example earlier, this record entry portion of the script did three things:

- Cleared the display
- Entered field data
- Clicked the Add button

Now you'll create a JST with a node for each of these operations.

1. **Open the Test Composer.**
   Click Compose Test in the JavaStar main menu. See Figure 5-2 for a screen shot of this window.

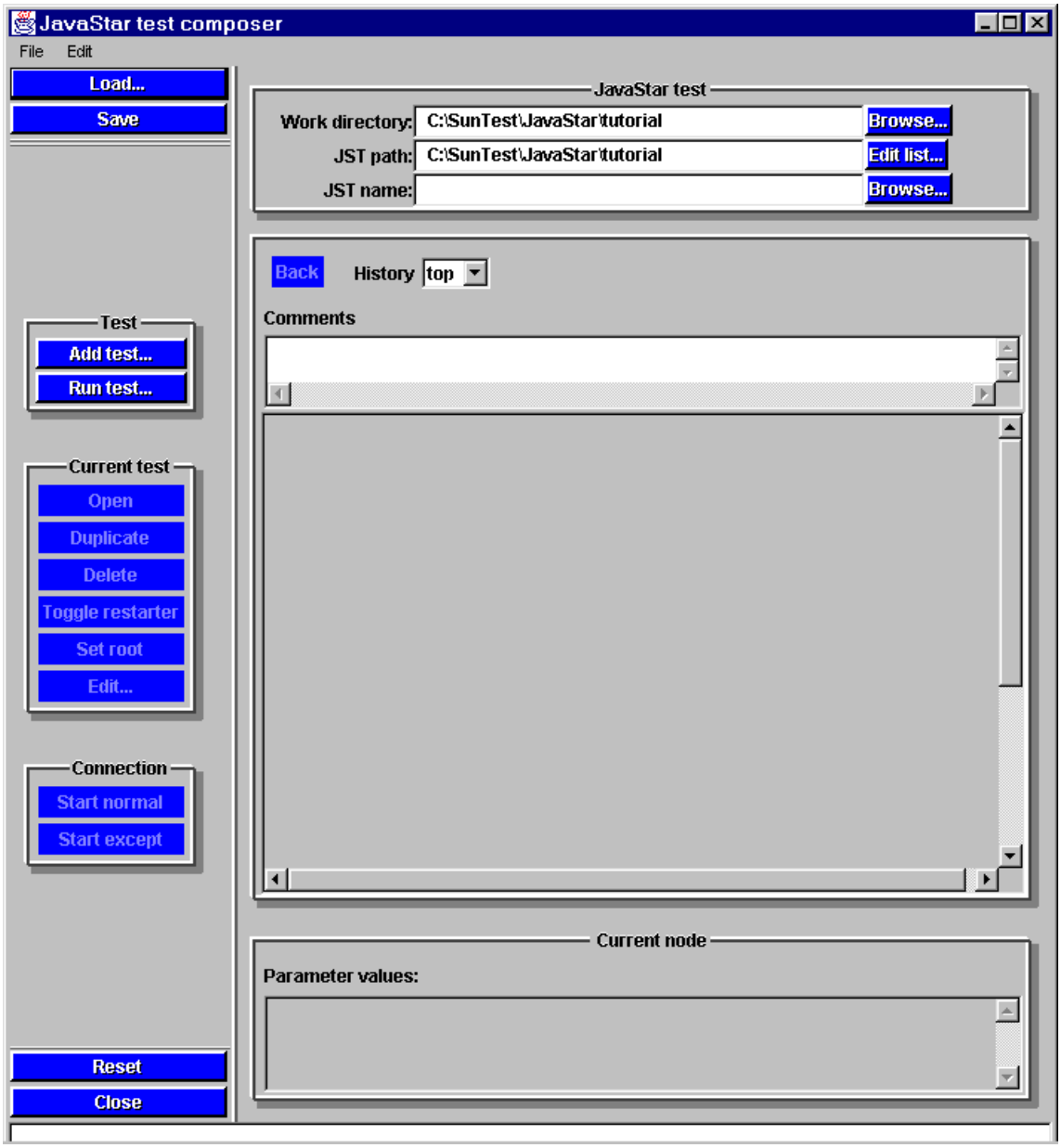


Figure 5-2 JavaStar Test Composer

The blank panel in the middle/right area of the Test Composer window is the work area where you graphically define your test. The buttons along the left control the file and the nodes you create, while the fields above the work area are where you define general information about your test, and where you navigate to other JSTs.

2. **Add your first test node.**
   You can do this in two ways:

   - By clicking on the Add test button
   - By clicking with a right mouse button anywhere in the work area, and choosing Add test from the menu that pops up.

   The only difference between these options is that using the Add test button places the new node in the upper left corner of the work area, while the right mouse button puts the node exactly where you positioned the mouse. If you use the Add test button, you can always click and drag the node later to position it where you want.

3. **In response to the Test Name dialog, enter `ClearDisplay` as the name of your node.**
   Because you are defining tests you haven't yet written, enter the name of the script you plan to write. When you run the test, JavaStar looks for a `.class` file by this name, so be precise. In this case, type `ClearDisplay` and click OK.

   Note – If you wanted to reference another JST file instead of a `.class` file, you would need to use the `.jst` extension.

4. **Create two more test nodes: one named `EnterFieldData` and the other named `Add`.**

5. **Position the nodes so that `ClearDisplay` is toward the top of the work area, with `EnterFieldData` below `ClearDisplay` and `Add` below `EnterFieldData`.**
   You can click and drag a node to re-position it.

6. **Draw a normal connection from `ClearDisplay` to `EnterFieldData`.**
   To create a normal connection:

   a. **Click on the `ClearDisplay` node to select it.**

   b. **From the buttons to the left, choose** Start Normal**.**
   `ClearDisplay` flashes to let you know an operation is in progress.

   c. **Click on `EnterFieldData` to select it as the end point.**
   JavaStar draws a green line from `ClearDisplay` to `EnterFieldData`.

7. **Draw a normal connecting line from `EnterFieldData` to `Add`.**

8. **Duplicate the `ClearDisplay` node.**
   Click on `ClearDisplay` to select it, then click the Duplicate button in the left button panel.

9. **Click and drag the `ClearDisplay` node to a position below the `Add` node.**

10. **Draw a normal connection from `Add` to the second `ClearDisplay` node.**

11. **For Comments, type `Adds 1 record to the database.`**
    Your work area should look like this:

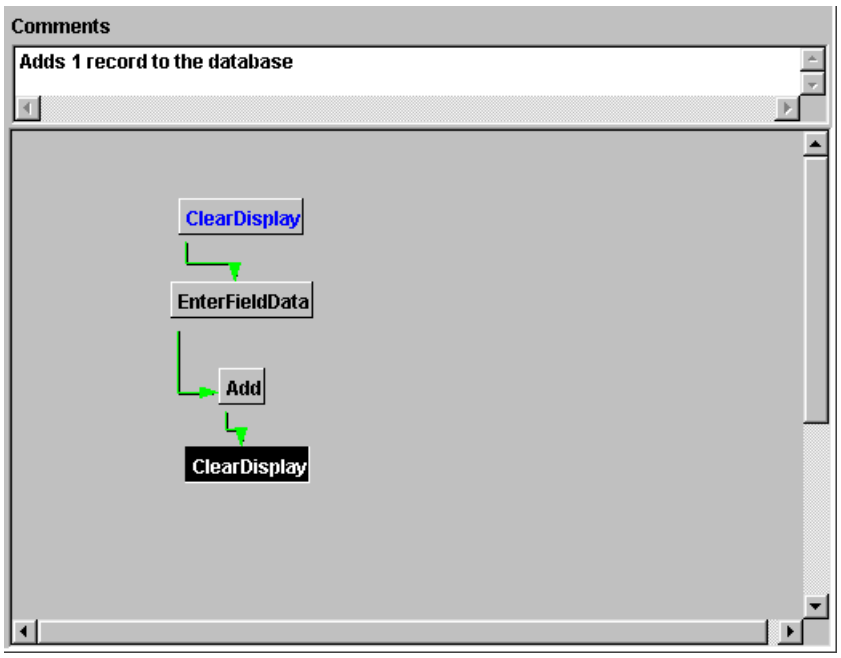


Figure 5-3 AddRecord.jst

The first node you created, `ClearDisplay`, appears in blue to indicate that it is the root or starting node. This is the node that the test will execute first. You can change a root starting node by selecting another node and clicking the Set Root button.

12. **In the** JST name **field (located toward the top of the window), enter `AddRecord.jst`.**
13. **Click the** Save **button.**
14. **Click** Reset **to clear the display and begin a new JST.**

Verifying Search Results

This JST verifies that the name matches the record you added. In the `TestNameDB` script you created in the previous chapter, you:

- Opened the Search window
- Selected the field to search on
- Defined the search string
- Performed the search
- Verified the results
- Closed the window

Because it's rare that you would perform a search and not select a record to view, this JST assumes you're combining these into one script, but leaving the rest of the steps as separate scripts.

1. **Create five nodes: `OpenSearch`, `DefineSearch`, `GetSearchResults`, `VerifyRecord`, and `CloseSearch`.**
2. **Position the nodes so that one is above the other, in the order listed.**
3. **Using the** Start Normal **button, draw connecting arrows from each node to the one below it.**
4. **In the** Comments **field, type `Searches for a record and verifies that the result is correct`.**
   Your work area should look like this:

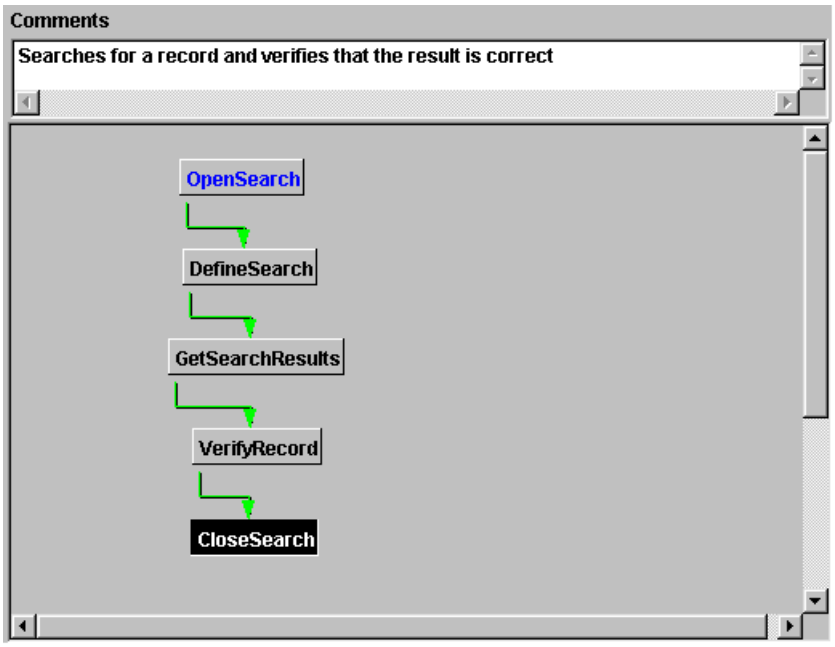


Figure 5-4 VerifySearch.jst

5. **Enter `VerifySearch.jst` in the** JST name **field.**

6. **Click** Save**.**

7. **Click** Reset**.**

The Acceptance Test

The script you wrote in the last chapter will be the model for this acceptance that draws all the test modules (the ones you have yet to write) together. This means the acceptance test needs to:

- Open a database file
- Add records
- Save the database
- Verify search operation
- Clear the display

The steps for adding records and verifying the name are now already defined by other JSTs and only need to be referenced. For loading a file, you need to create a node for another script you'll write. `ClearDisplay` is a script you've already referenced (but not yet written) in `AddRecords.jst`, so you'll use the same name again.

To define the acceptance test:

1. **Click Add Test.**

2. **Enter `OpenFile` as the test name and click** OK**.**

3. **Add another test and specify `AddRecord.jst` as the test name.**
   Click Add test. For Test name, click the browse button, locate `AddRecord.jst`, and click Open. Then click OK.

   JavaStar creates a rectangle with rounded corners to show that this node launches another JST, rather than running a `.class` file.

4. **Add a node named `SaveDB`.**
   This node has squared corners, indicating it points to a script.

5. **Create another node, this one pointing to `VerifySearch.jst`.**
   You can type the names in instead of browsing, but if you forget the `.jst` extension or misspell the name, JavaStar won't find the file you want.

6. **Add a node named `ClearDisplay`.**

7. **Add a node named `CloseDB`.**

8. **Organize the nodes in the order you plan to run them.**
   See the list at the opening of this section.

9. **Create normal connections from each node to the next.**

10. **For** Comments**, type** `Simple acceptance test for Name Database.`
Your work area should look like:

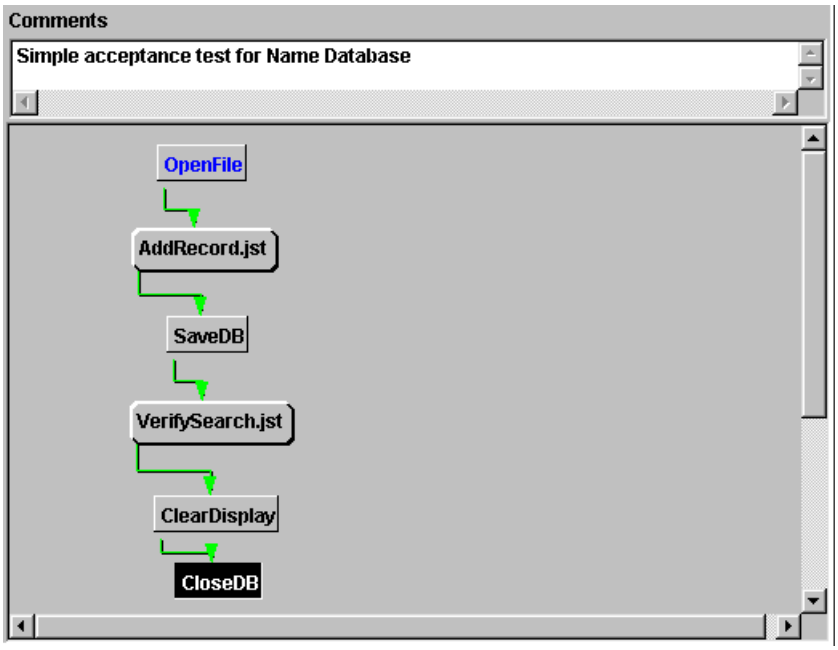


Figure 5-5 Acceptance.jst

11. **In the** JST name **field, enter** `Acceptance.jst`.
12. **Click** Save.
13. **Close the Test Composer.**

Recording Individual Scripts

By setting up the JSTs, you've specified all the scripts you need to write. Currently, you need to create:

- OpenFile
- ClearDisplay (used three times)
- EnterFieldData
- Add
- SaveDB
- OpenSearch
- DefineSearchString
- GetSearchResults
- VerifyRecord
- CloseSearch
- CloseDB

This may sound like a lot of scripts, but each is well-contained and quick to create.

Going into Create Test Script mode

1. **Select** Create Test Script **from the main menu.**
2. **Enter `namedb` as the** Class name **and click** Start**.**
   JavaStar opens the Record / Playback window and the Name Database application. You may need to move the windows around to make sure you can see both the Record / Playback window and the Name Database window while you work.

Recording OpenFile

1. **Start recording a script named `OpenFile`.**
   Click Record and enter `OpenFile` as the test name. Note that the Record with map files field is already filled in for you. Click OK.
2. **In the Name Database main window, click** Open**.**
   The file dialog opens.
3. **Navigate to the tutorial directory, select `test.db` and click** Open**.**
4. **In the Record/Playback window, click** Synchronize **to enter Synchronization mode.**
5. **In the Name Database main window, select the component by clicking on the** Name Database - test.db **text at the top of the window.**
   The selection code appears in the Synchronization panel, and the default method changes to *Using text*.
6. **In the Synchronization window, click** Use default**.**
7. **Enter a purpose in the** Why **field.**
   For Why, type `Proceed only if correct file loaded`.
8. **Click the** Insert synchronization into test **button.**
   This inserts the synchronization code into the script.
9. **Click** Continue**.**
10. **Click** Stop**.**
    This completes the `OpenFile` script.

Recording ClearDisplay

This simple script is used several times by JSTs, giving it immediate re-use potential.

1. **Start record mode and name the test `ClearDisplay`.**
2. **In the Name Database main window, click** Clear.
3. **Stop recording.**

Recording EnterFieldData

1. **Start record mode and name the test `EnterFieldData`.**
2. **Click in the Name text field and type `Count von Count`.**
3. **Pressing** Tab **to advance to each field (except between Phone and Email, where you need to click inside the Email field to continue), complete the rest of the fields as shown:**

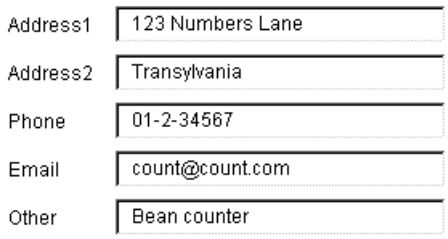


Figure 5-6 Values for data entry fields

4. **Stop record mode.**

Recording Add

1. **Enter record mode using the test name `Add`.**
2. **In the Name Database main window, click** Add.
3. **Stop record mode.**

Recording SaveDB

1. **Enter record mode using the test name `SaveDB`.**
2. **In the Name Database main window, click** Save.
3. **Stop record mode.**

Recording OpenSearch

1. **Enter record mode using the test name `OpenSearch`.**
2. **In the Name Database main window, click** Search**.**
3. **Stop record mode.**

Recording DefineSearch

1. **Start record mode using `DefineSearch` as the test name.**
2. **In the Search window, click on the** select criteria **choice list and choose** address2**.**
3. **Enter the search string into the** contain strings **text field.**
   Type `Transylvania`.
4. **Stop record mode.**

Recording GetSearchResults

1. **Start record mode using `GetSearchResults` as the test name.**
2. **In the Search dialog of the Name Database, click the** Search **button to perform the search.**
3. **Verify the item count**:
   a. **Go into Verification mode.**
      Click Verify.
   b. **Click in the search results list to select that component.**
   c. **Click** Customize**.**
   d. **Select** Using simple methods and fields**.**
   e. **Click the** Select simple methods and data members **button.**
   f. **Scroll through the method list and select `int getItemCount()`.**
   g. **Click** Enter a purpose**.**
   h. **Enter the purpose of the verification and click** Insert verification into test**.**
   i. **Click** Continue**.**
4. **In the results list of the Search window, select the `Count von Count` record.**
5. **Click** View Result**.**
6. **Stop record mode.**

Recording VerifyRecord

1. **Enter record mode using the test name `VerifyRecord`.**
2. **Immediately enter Verification mode.**
   Click Verify.
3. **In the Name Database main window, select the first field to verify.**
   Click in the Name text field to select that component.
4. **Click** Use default **to accept** *using text* **as the method.**
5. **Enter a purpose.**
   In the Why field, type `Verify data entry`.
6. **Click the** Insert verification into test **button.**
7. **For the remaining record fields (including the** Telephone **field):**
   a. **Click in the next field to select the component.**
   b. **Click** Use default**.**
   c. **Click** Insert verification into test**.**
8. **Click** Continue**.**
9. **Stop record mode.**

Recording CloseSearch

1. **Enter record mode using the test name `CloseSearch`.**
2. **In the Search window, click** Close**.**
3. **Stop record mode.**

Recording CloseDB

1. **Enter record mode using the test name `CloseDB`.**
2. **In the Name Database main window, click** Close**.**
3. **Stop record mode.**

Quit Playback/Record

- **Click the** Quit **button in the Playback/Record window and confirm your choice.**

Running Tests

Running a JST is a little different than running a script. When you run a script, the Record / Playback window shows you the script as it executes, highlighting each line. Because a script has only one file, it's more obvious where you are while the test runs. But for JSTs, JavaStar is opening and closing various tests. To help you track where the test is executing, JavaStar opens a JST Runner window.

The JST Runner shows the graph of your JST just as you created it in the Test Composer. As each node executes, the JST Runner flashes that node. When JavaStar encounters a node that is a JST itself, it begins tracking that JST in the JST Runner. When the nested JST finishes executing, the JST Runner displays the upper-level JST and continues where it left off.

The Record / Playback window still shows the code of each script as it executes. The only difference is that now the JST Runner shows you which script is being executed in the Record / Playback window.

Running a JST

1. **Copy the original `sesame.db` to `test.db`, replacing the existing file.**
2. **In the JavaStar main menu, click** Run test**.**
3. **For the test name, type `Acceptance.jst`.**
4. **Click** Start**.**
   JavaStar launches the test application, opens the Record / Playback window, and also opens the JST Runner window to show the progress through the JST as it runs. You may need to move the windows around at the beginning so you can see the JST Runner.

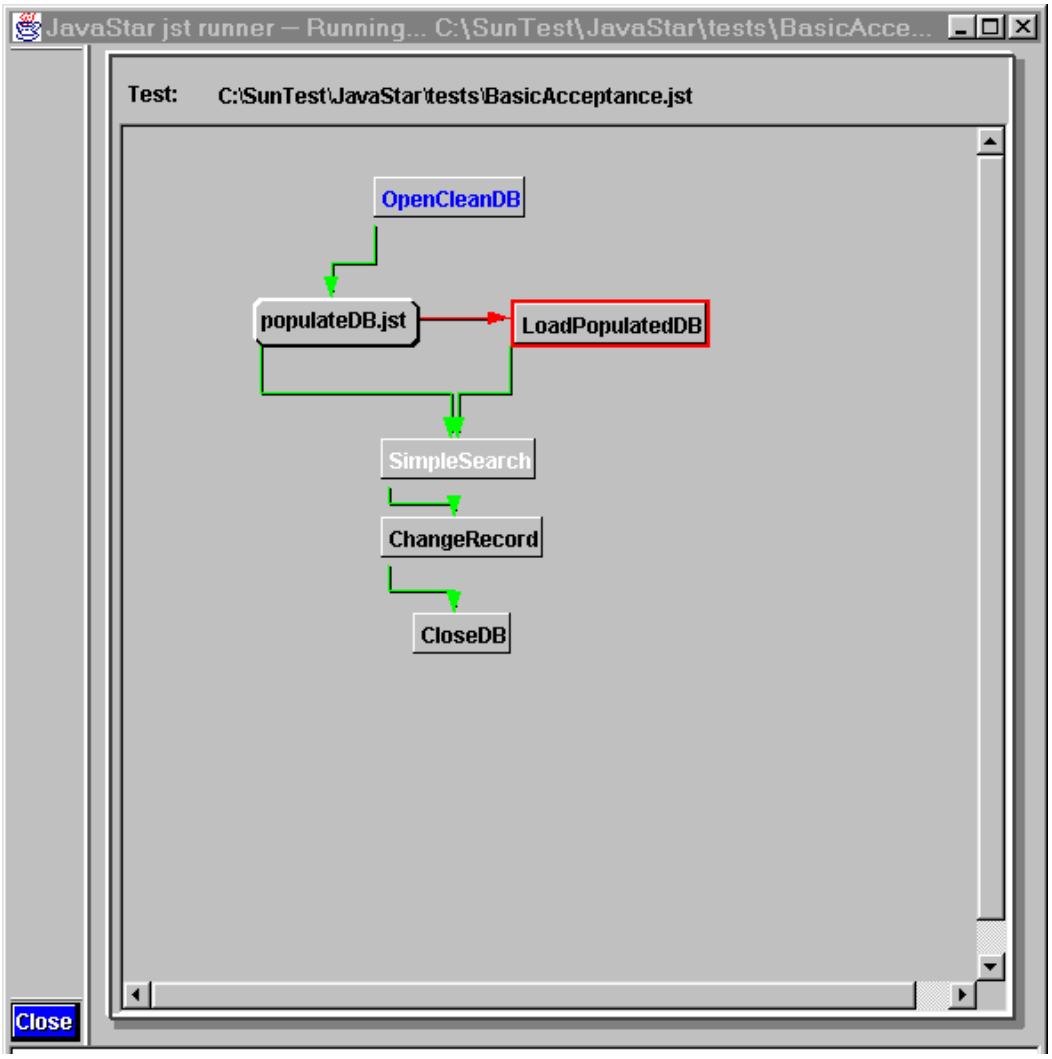


Figure 5-7 JST Runner showing a graphic display of the currently executing test

5. **When the test finishes, quit the Record/Playback window.**
   This closes the application and JST Runner windows as well.

Debugging a Test

If run test stops before the JST finishes, note which node is flashing in the JST Runner window. Check the log file display in Record / Playback window to see what the error is. If it can't find the file, you might have misspelled the node name either when you created the node or the recorded the script.

To debug the problem:

1. **Stop playback in the Record/Playback window.**
2. **Close the JST Runner.**
3. **Click** Playback **and enter the name of the node (JST or script) where the test stopped.**
4. **Examine the test while it executes and try to determine the problem.**
   If you can't find anything wrong with the test, check the previous node. It's possible that the previous node didn't leave the application in the proper state for this node to execute properly. For example, if one of the scripts didn't clear the display before adding a record, the data it enters would then be mixed with that already in the first record of the database.

Viewing Results from a JST

One of the many benefits of using JSTs is that the results you get are as modular as the test itself, and thus easier to read. JavaStar presents the results in the form of an expandable and collapsible tree, with summary information available at each node.

To view the results of the test you just ran:

1. **From the JavaStar main menu, click** View Test Results.
   The Results Viewer opens. By default, this should show the log file for the test you just ran. If you've run another test since running Acceptance.jst, then to load the acceptance results:

   a. **Click the** Open **button in the left panel.**

   b. **Locate the `Acceptance.jst.log` file and click** Open**.**

   The log file appears in the Test Results panel as a hierarchical tree, with the first level of tests expanded.

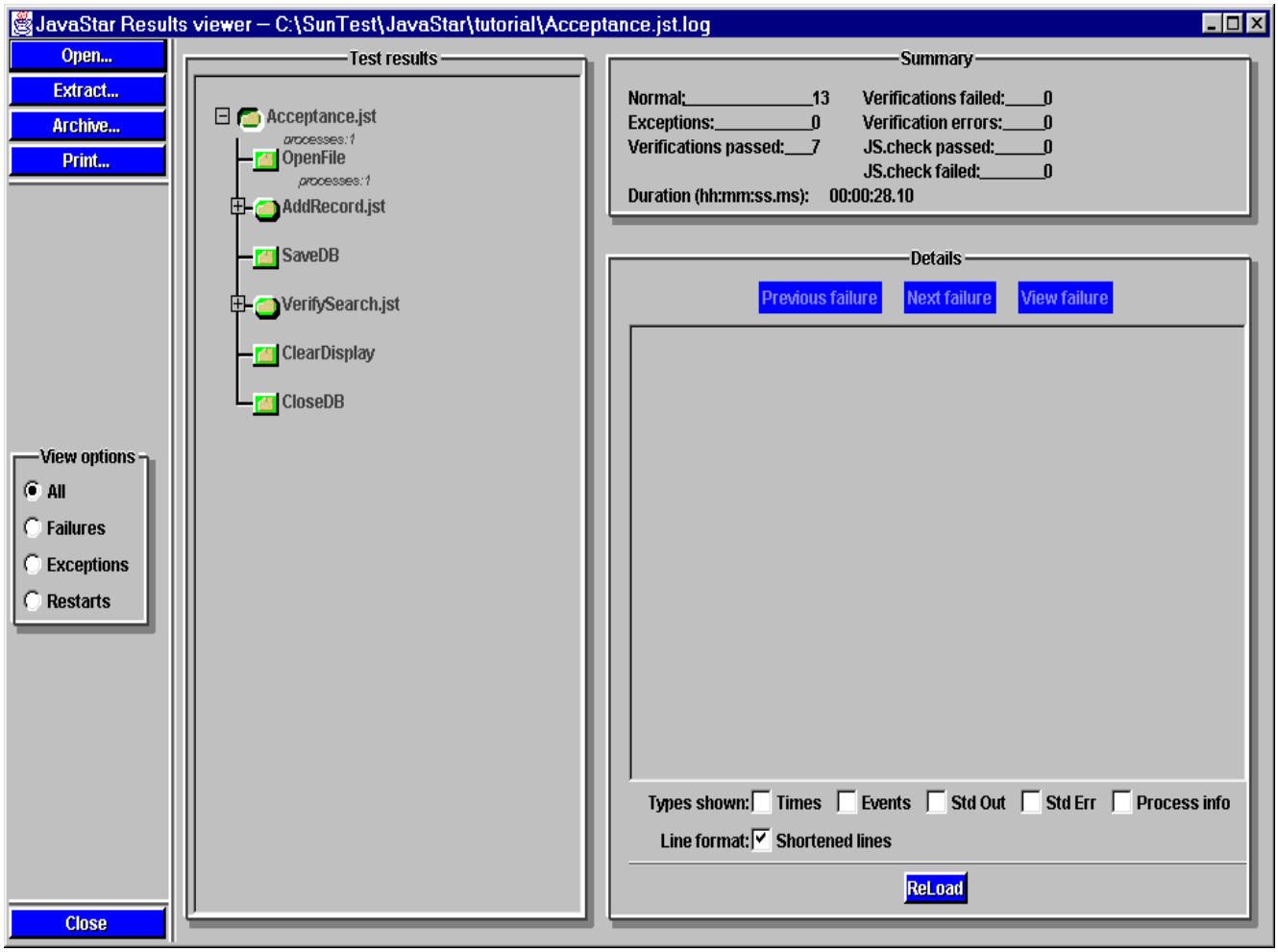


Figure 5-8 Results Viewer showing Acceptance.jst results

2. **Expand the log file results by clicking on the + symbols to the left of `AddRecord.jst` and `VerifySearch.jst`.**
   This expands `Acceptance.jst` to show the nodes. The symbol in front of a node indicates its type: rectangle with square edges is a script, while a rounded rectangle is another JST file.

   If any nodes failed—i.e., they ended with an exception—a notice of the failure appears below the node in red. Green symbols signify normal ends.

   The Summary box here shows that `Acceptance.jst` had 13 nodes that ended normally and 0 that ended with exceptions. Seven verifications passed and none failed. The remaining values should be zero. Had you expanded your tests to use the `JS.check` features of the JavaStar API, these results would be summarized here, as well.

If you select different nodes in the JST, you'll see that the summary information changes to show the summary for that test and any tests below it. The log panel always displays information for the currently selected node, as well.

3. **Close the Results viewer.**
   Click Close.

Summary

This chapter introduced you to the processes of creating tests that work with the JavaStar modular model. You should now be comfortable analyzing your application to define modular tests, creating simple JSTs, running JSTs, and understanding the results from nested JSTs in the Results Viewer.

The next chapter takes the scripts and JSTs you created in this chapter and moves one step further by adding parameters, further increasing re-use potential.

Exercise: Making the Names Test Modular

Now that you know how to redesign a long script into several modules, try this out with the TestNames test from the last chapter's exercise. If you didn't do that exercise, you can find the solution (`TestNames.java` and `TestNames.java`) in the `\tutorial\gettingStarted` directory.

Instructions

For this exercise:

1. Compose a multi-node JST to test the names window.
2. Record the scripts that correspond to each node.
3. Integrate the JST for Names into a copy of the acceptance test. Name this copy `Acceptance2.jst`, so that you don't confuse it with the original version.
4. Run the test to make sure it works.

Solution

There's more than one way to tackle this exercise. One solution breaks the search test down into four modules:

- `OpenNames`—open the Names window
- `SelectName`—select a name from the list
- `VerifyRecord`—the same script you recorded earlier in this chapter, with no modifications
- `CloseNames`—close the Names window

You might have chosen fewer, and that's okay. Using four modules comes in handy later on, when you can parameterize individual nodes.

If you did use four nodes, your search JST in the Test Composer should look somewhat like this:

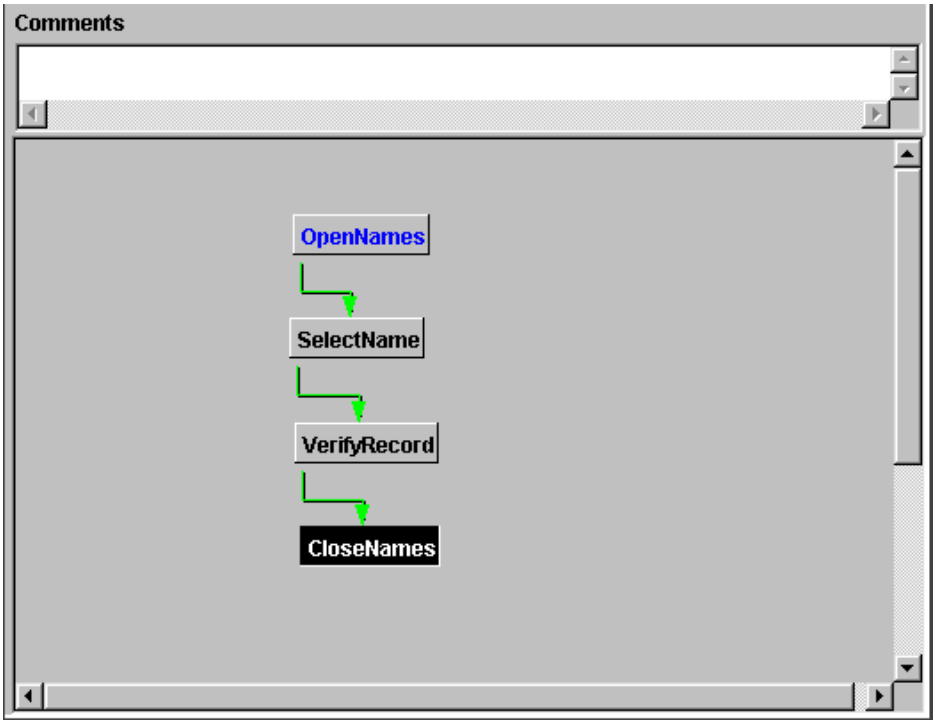


Figure 5-9 VerifyNames.jst

Incorporating the VerifyNames JST into the acceptance test requires only a few steps. You only need to:

1. **Open `Acceptance.jst` in the Test Composer.**
2. **Add a node for VerifyNames.jst (or whatever you named your Names test).**
3. **Draw a normal, green arrow from `ClearDisplay` to `VerifyNames.jst`.**
4. **Save the results as `Acceptance2.jst`.**
   The Acceptance2.jst work area should have looked similar to this:

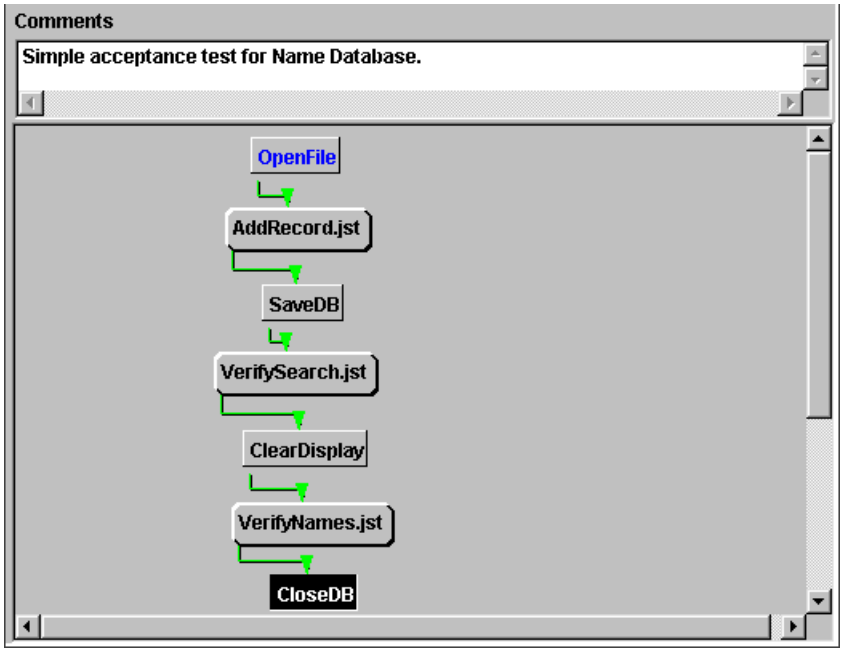


Figure 5-10 Acceptance2.jst—one solution

When you run this test, the summary for the overall Acceptance2.jst run should show 12 verifications instead of the original six.

You can find `VerifyNames.jst` and the scripts that support it in the `\tutorial\modular` directory. Look at the files:

`VerifyNames.jst`
`OpenNames.class` and `OpenNames.java`
`SelectName.class` and `SelectName.java`
`VerifyRecord.class` and `VerifyRecord.java`
`CloseNames.class` and `CloseNames.java`
`Acceptance2.jst`

Adding Parameters for Flexibility 6

This chapter shows you how to use JavaStar to pass arguments within modular tests to increase flexibility and re-use.

Breaking tests down into modules is the first step in making your tests reusable. The second step is replacing hard-coded information—for example, field data and component labels—with parameters. When you do this, you can create fewer modules that can be combined to make more varied tests.

This lesson guides you through the process of modifying the tests you created in the last chapter to use parameters. After following this lesson, you should know how to:

- Analyze tests to determine where to use parameters
- Edit scripts to add parameters
- Edit JSTs to pass parameters to scripts
- Use test arguments when running a test
- View parameter values in the Results Viewer

Topics:

Setting Up for this Lesson

If you are continuing from the previous lesson, all you need to do to prepare for this lesson is to:

- **Copy the original** `sesame.db` **over** `test.db`**, to return the database to a clean state.**

If, however, you have not done the previous lessons:

1. **If you have not already done so, follow the instructions for** Setting up JavaStar **as described in the chapters "**Getting Started with JavaStar**" and "**Using a Modular Approach**"**

   Note – Both chapters contain information about setting up a project file. "Getting Started" describes how to do the initial setup for a project file. "Using a Modular Approach" describes how to modify the mapping information in the project file to use declarations. You need to follow both sets of instructions to continue with this lesson.

2. **If you did not do the lesson in chapter** Generating Declarations**, then**:

   a. **Create a directory within the** `tutorial` **directory called** `NameData`**.**

   b. **Copy the contents of the** `\javastar\tutorial\NameDataDecls` **into** `\javastar\tutorial\NameData`**.**

3. **Copy the contents of the** `\javastar\tutorial\modular` **directory to** `\javastar\tutorial`**.**

Deciding Where to Use Parameters in Scripts

When you begin planning how to "parameterize" a test, start analyzing the test at the script level. On a script-by-script basis, examine where you might want to replace data with parameters. Good candidates for parameters are any place where a script:

- Responds to a prompt
- Selects an item from a menu or list
- Supplies data (such as a string or a date)
- Chooses from a series of options

These aren't the only places where you might want to replace hard-coded data with parameters, but they get you off to a good start.

Looking at the acceptance test you created in the last chapter, there are several places where parameters would increase versatility. Perhaps the most obvious candidates are scripts that require a filename, and scripts that work with field data for a name database record.

`OpenFile` is the only script that uses the database filename: it's hard-coded to load and verify `test.db`. You can change this filename to a parameter and then define the filename when you run the test. This makes this script easier to use in other composed tests, and also gives you ways to adapt `Acceptance.jst` for multiple, varied test runs.

As far as record data is concerned, `Acceptance.jst` is now hard-coded to create, select, and verify the "`Count von Count`" record. This is a somewhat limited test case. You can build other tests with the scripts that make up the acceptance test, but you are still restricted to the "`Count von Count`" data. If, instead, you modify the scripts to take the field values as parameters, your scripts will function independent of the data. Re-use potential then increases enormously.

Examining the scripts of the acceptance test, you can see that four scripts use data from the same record:

- **EnterFieldData**—inputs values for the six fields of a record: Name, Address1, Address2, Phone, Email, and Other.
- **DefineSearch**—uses the value of the Address2 field as a search argument.
- **GetSearchResults**—selects the record from the search results list, selecting the Name value used in EnterFieldData.
- **VerifyRecord**—performs field-by-field verification on a record, comparing the results to the values entered in EnterFieldData.

These four scripts use the same six parameters:

- Name
- Address1
- Address2
- Phone
- Email
- Other

With the addition of the filename for the test database, you'll be editing your tests to use a total of seven parameters.

To change these scripts to use parameters, you'll edit the `.java` files to replace the hard-coded data with argument references. At runtime, a script takes any parameters passed to it and places the parameters within the `args[]` array. This means that for every script, the first parameter becomes `args[0]`, the second `args[1]`, the third `args[2]`, and so on. So, when you edit EnterFieldData, you'll replace the current field values with `args[0]` through `args[5]`. Within OpenFile, you'll replace the `test.db` string with `args[0]`, as well.

Editing the Scripts

This lesson describes how to add parameters, using the script editor included with JavaStar. The JavaStar Script Editor includes a button for saving and compiling the modified code, making it convenient for this exercise. You can use any editor you like to edit the scripts—just make sure you compile the code when you are done.

This section covers:

Editing OpenFile

1. **In the JavaStar main menu, click** Edit Test Script**.**

2. **Open** `OpenFile.java` **for edit.**
   Next to the Script name field, click the Browse button. Locate and select `OpenFile.java`. Double-click the filename or click Open.

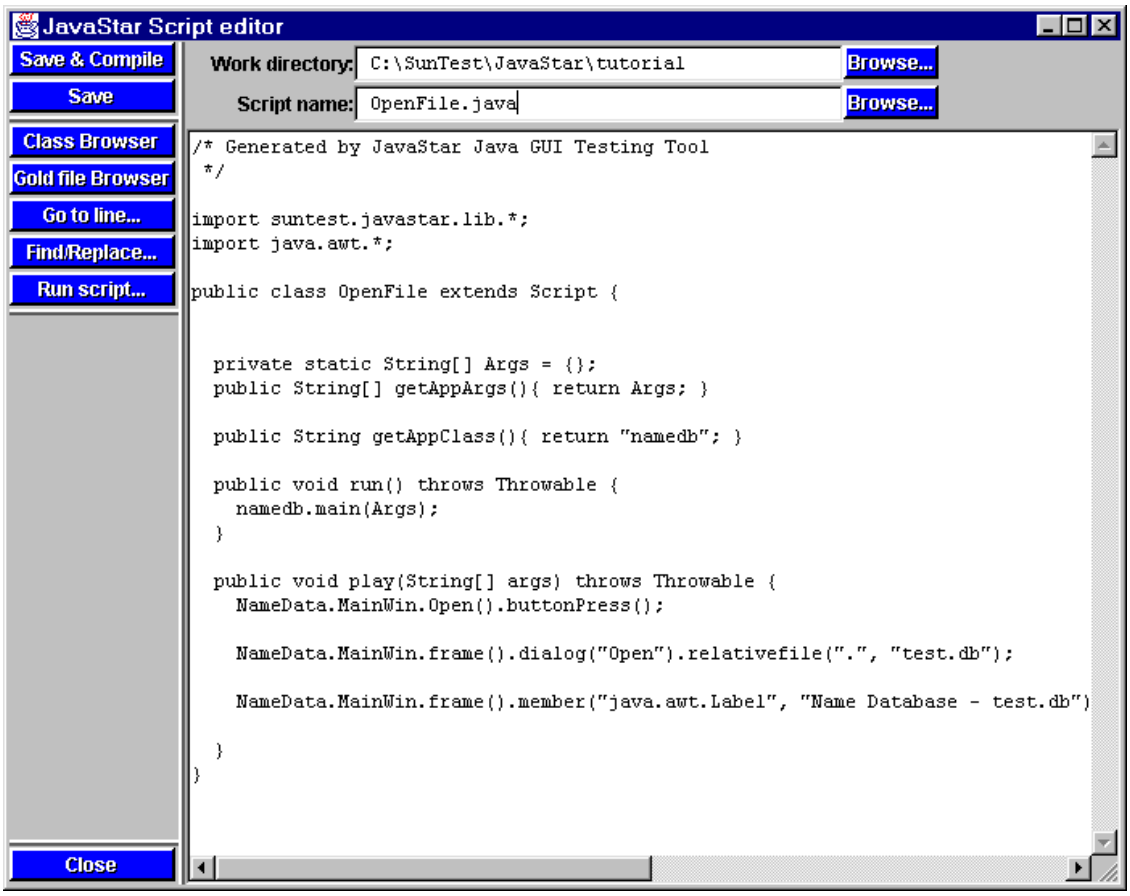


Figure 6-1 JavaStar Script Editor

3. **Scroll down to the play method.**
   The play method opens with the line:

   ```
   public void play(String[] args) throws Throwable {
   ```

4. **Replace the** `test.db` **reference with** `args[0]`**.**
   The current line reads:

   ```
   NameData.MainWin.frame().dialog("Open").relativefile(".
   ", "test.db");
   ```

   Change `"test.db"` to `args[0]`. Leave the `"."` parameter before the filename so that JavaStar knows to look for the database file in the current directory. Remove the quotation marks. The new line should read:

   ```
   NameData.MainWin.frame().dialog("Open").relativefile(".
   ", args[0]);
   ```

5. **Edit the synchronize operation to handle a variable filename.**
   Right now, this script checks the Name Database header in the window to see that it reads `Name Database - test.db`. This is done with this line:

   ```
   NameData.MainWin.frame().member("java.awt.Label", "Name
   Database - test.db").waitFor("Name Database - test.db",
   "Proceed only if correct file loaded");
   ```

   You accommodate the variable filename in this comparison by changing `"Name Database - test.db"` to `"Name Database - " + args[0]`. Be sure to change it in both places on this line—in the `member()` and `waitFor()` method calls—and include the plus-sign.

   ```
   NameData.MainWin.frame().member("java.awt.Label", "Name
   Database - " + args[0]).waitFor("Name Database - " +
   args[0], "Proceed only if correct file loaded");
   ```

6. **Click** Save & Compile.
   JavaStar should return a message that it compiled the script successfully. If not, check for errors—make sure you didn't accidentally delete a parenthesis or comma, or leave out the plus-sign.

Editing EnterFieldData

1. **In the Script Editor, open** `EnterFieldData.java` **for edit.**

2. **Scroll down to the play method.**
   The play method opens with the line:

   ```
   public void play(String[] args) throws Throwable {
   ```

3. **Edit the first data entry line.**
   Find the line:

   ```
   NameData.MainWin.nameTextField().typeString("Count Von
   Count", 0, 0);
   ```

   On this line, change `"Count von Count", 0, 0` to `args[0]` so that it reads:

   ```
   NameData.MainWin.nameTextField().typeString(args[0]);
   ```

   Note that in addition to replacing the text string with the argument, you're deleting the position values `(0,0)` that follow. The JavaStar API provides two versions of the of the `typeString()` method—one requiring the string to replace, selection start, and selection end parameters, and the other requiring only the string to replace. Here you're replacing the default version recorded in the script with a simpler method call. That just replaces the entire contents of the text field.

4. **Edit the remaining data entry lines:**

   a. **Change the reference to** `"123 Numbers Lane", 0 ,0` **to** `args[1]`**.**
      Find:

```
NameData.MainWin.address1TextField().typeString("123
Numbers Lane", 0, 0);
```

Replace with:

```
NameData.MainWin.address1TextField().typeString(args[1]);
```

b. Change the reference to `"Transylvania", 0 ,0` **to** `args[2]`**.**
Find:

```
NameData.MainWin.address2TextField().typeString("Transylv
ania", 0 ,0);
```

Replace with:

```
NameData.MainWin.address1TextField().typeString(args[2]);
```

c. Change the reference to `"01-2-34567", 0 ,0` **to** `args[3]`**.**
Find:

```
NameData.MainWin.telephoneTF().typeString("01-2-34567", 0,
0);
```

Replace with:

```
NameData.MainWin.telephoneTF().typeString(args[3]);
```

d. Change the reference to `"count@count.com", 0 ,0` **to** `args[4]`**.**
Find:

```
NameData.MainWin.emailTextField().typeString("count@count.
com", 0, 0);
```

Replace with:

```
NameData.MainWin.emailTextField().typeString(args[4]);
```

e. Change the reference to `"Bean Counter", 0 ,0` **to** `args[5]`**.**
Find:

```
NameData.MainWin.otherTextField().typeString("Bean
counter", 0, 0);
```

Replace with:

```
NameData.MainWin.otherTextField().typeString(args[5]);
```

5. Click Save & Compile.

Editing DefineSearch

1. **In the Script Editor, open** `DefineSearch.java` **for edit.**
2. **In the Play method, edit the line that specifies the search string.**
   Find the line:

   ```
   NameData.SearchWin.containsStrTextField().
   typeString("Transylvania", 0, 0);
   ```

   (This code appears as a single line in the Script Editor.)

   Replace the line with:

   ```
   NameData.SearchWin.containsStrTextField().typeString(args
   [0]);
   ```

   Note – Argument references always start at 0 within a script. The argument value refers to the arguments position in the array of arguments passed to the script at runtime. It does not refer to an absolute position for all arguments passed into the JST.
3. **Save and compile.**

Editing GetSearchResults

1. **In the Script Editor, open** `GetSearchResults.java` **for edit.**
2. **In the Play method, edit the line that selects the** `Count von Count` **record by name.**
   Find the line:

   ```
   NameData.SearchWin.list().select(0,"Count Von Count");
   ```

   Replace `"Count von Count"` with `args[0]` so the line reads:

   ```
   NameData.SearchWin.list().select(0,args[0]);
   ```
3. **Edit the code to remove the selection position.**
   Currently the code passes a position and a string to the `select` method. The JavaStar API provides another version of this method that selects an item using only the string. If you now plan to pass the string name to the script, you don't know what the position of that string will be within the list when this is executed as part of a test, so you need to perform the selection based solely on the string value.

   To make this modification, delete `0,` so the line reads:

   ```
   JS.frame("Search").member("java.awt.List").select(args[
   0]);
   ```
4. **Save and compile.**

Editing VerifyRecord

1. **In the Script Editor, open** `VerifyRecord.java` **for edit.**
2. **In the Play method, edit the line that selects the** `Count von Count` **record by name.**
   a. **Find the line:**

      ```
      NameData.MainWin.nameTextField().verify(this,"Count von
      Count", "Verify expected field text");
      ```

   b. **On this line, change** `"Count von Count"` **to** `args[0]` **so that it reads:**

      ```
      NameData.MainWin.nameTextField().verify(this, args[0],
      "Verify expected field text");
      ```

3. **Edit the remaining verification lines.**

Replace…	With…
`"123 Numbers Lane"`	`args[1]`
`"Transylvania"`	`args[2]`
`"01-2-34567”`	`args[3]`
`"count@count.com"`	`args[4]`
`"Bean Counter"`	`args[5]`

4. **Save and compile.**

Deciding Where to Define Parameters in the JST

Once you have scripts edited to use parameters, you’re ready to compose new JSTs or modify existing tests to pass parameters and define constant values. This section shows you how to modify the acceptance test to incorporate parameters to support the newly edited scripts.

> ***Note –*** You don’t have to create a JST to use parameters. If you have a script that runs start-to-finish without needing any other script, and you want to use parameters, you can go ahead and replace portions of the code with argument references. Then, when you run the script, you just need to specify each of the parameters in the Test arguments field of the Run test window. This works for small, contained test cases—for anything more complex or varied, composed tests are where you’ll access the power.

Just as you examined the scripts to see where you want to add parameters, now you’ll examine the acceptance test to see how you want to supply those parameters to the scripts.

An outline of `Acceptance.jst` (something similar to the outline you seen in the Results Viewer when you expand all nodes) shows the nodes as:

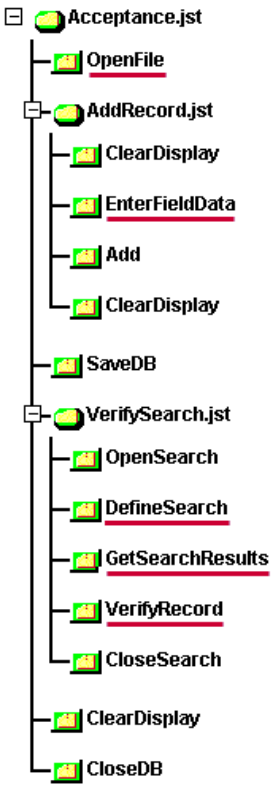


Figure 6-2 An outline view of `Acceptance.jst`

The scripts that you edited to take parameters are underlined in this illustration. At the very least, you need to set up these nodes to pass parameters. But you can do more than that.

JavaStar supports three ways of supplying parameters to a node:

- **Constants**—when you pass a constant, you define the value of the parameter in the node itself. This isn't the same as defining it in the script—the node passes the value to the script at runtime.
- **Parent parameters**—when you define an argument as a parent parameter, JavaStar knows to get the data from arguments passed to the test at runtime or, in the case of a multi-level JST, to inherit these arguments from the "parent" of this node.
- **Property file**—when you define a property file as the source of the argument, JavaStar looks for a Java property file with the name you define. This is useful when you want to reference arguments that change

depending on the platform where you're running the test. You can provide a different property file (same name, different contents) for each platform you test under, and the test will run on each without modifications.

For the acceptance test to work properly, `EnterFieldData`, `DefineSearch`, `GetSearchResults` and `VerifyRecord` require identical data (though `DefineSearch` and `GetSearchResults` use only portions of the record). Because of this, it makes sense to set these parameters not at the level of particular test nodes, but at the level of the first common parent.

You can do this using parent parameters, by setting up an inheritance scheme so that the nodes for these scripts inherit the parameters from their parent node, and the parent nodes then inherit the parameters from *their* parent (`Acceptance.jst`). You then supply the parameter values at runtime.

`OpenFile` requires the filename for the test database. You can pass this as a constant or inherit it from the parent. For this example, you'll use a parent parameter.

Figure 6-3 shows the outline again, this time with all the nodes that pass parameters marked in bold. Based on this plan, all parent nodes will pass parameters down to child nodes. EnterFieldData will inherit parameters from `AddRecord.jst`, and `AddRecord.jst` will inherit these same parameters from the test arguments passed to `Acceptance.jst`.

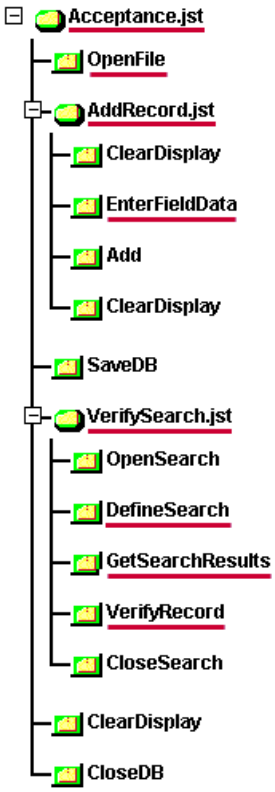


Figure 6-3 `Acceptance.jst` with parameter nodes underlined.

How you define the parameter structure for `Acceptance.jst` doesn't affect how you use these scripts with another JST. You might want to create a JST that populates a Name Database with records, and that might mean calling `AddRecord.jst` repeatedly. In that case, you'd probably define each AddRecord node to pass the parameter values as constants, rather than define all of them as test arguments.

> ***Note –*** Another way to populate a database using AddRecord would be to edit a script yourself to read data in from a property file, then use a combination of `setProperty()` and `getProperty()` to set field values. You'll learn more about adding custom code to scripts in the chapter "Using the JavaStar API."

Editing JSTs to Use Parameters

Now you'll go into Compose Test to set the JSTs to accept and pass parameters. You'll be working with seven parameters that you'll pass to the acceptance test at runtime:

`$0 Name`
`$1 Address 1`
`$2 Address 2`
`$3 Phone`
`$4 Email`
`$5 Other` (for the Other field in the name database record)
`$6 Test database filename`

A Note About Argument Identifiers

The argument numbers you specify in the Edit Node dialog for a particular node represent the positions of the arguments that node will inherit from its parent node. These numbers do not reflect the position numbers used by the individual scripts. Each script takes whatever arguments you pass to it and numbers them sequentially within its own `args[]` array, starting at zero.

Because this exercise uses parameters that are passed from one node to the other in the same order, it's hard to see what this would mean, except in the case of OpenFile. The OpenFile script uses the last parameter passed to `Acceptance.jst` at runtime—`args[6]`. When you specify the argument number in the Test Composer, you refer to argument #6. But the OpenFile script references `args[0]` in the actual code, because it takes whatever parameters you pass it (whether it's `args[0]`, `args[3]`, or `args[6]`) and renumbers them starting at `args[0]`.

This keeps your script independent of the JST's implementation. You don't have to change a script depending on how you want to get nodes from the parent. All the Edit Node dialog needs is information on which arguments you want it to get from the parent, and it will do the rest.

Editing the OpenFile Node

To edit the OpenFile node pass parameters to the script:

1. **In the JavaStar main menu, select** Compose test**.**
2. **Load** `Acceptance.jst`**.**
   Click the Browse button. Locate `Acceptance.jst` in the `Tutorial` directory, then click Open.
3. **Select the** `OpenFile` **node and click the** Edit **button.**
   The Edit Node dialog opens.

4. **In the Edit Node dialog window, select** Parent parameter **from the pull-down menu.**
   The Value field changes to Argument #.

5. **In the** Argument # **field, type** `6`.
   For this node, you'll use the seventh parameter passed to the acceptance test, after the first six field values. The seventh parameter occupies position 6 in the args array, since the first parameter is numbered `0`.

6. **Click the** Add (after) **button.**
   This adds the parameter to the list. This is the only parameter you need for this script.

7. **Enter a comment to describe the parameter.**
   In the Comments field, type `$6 = Database filename`.

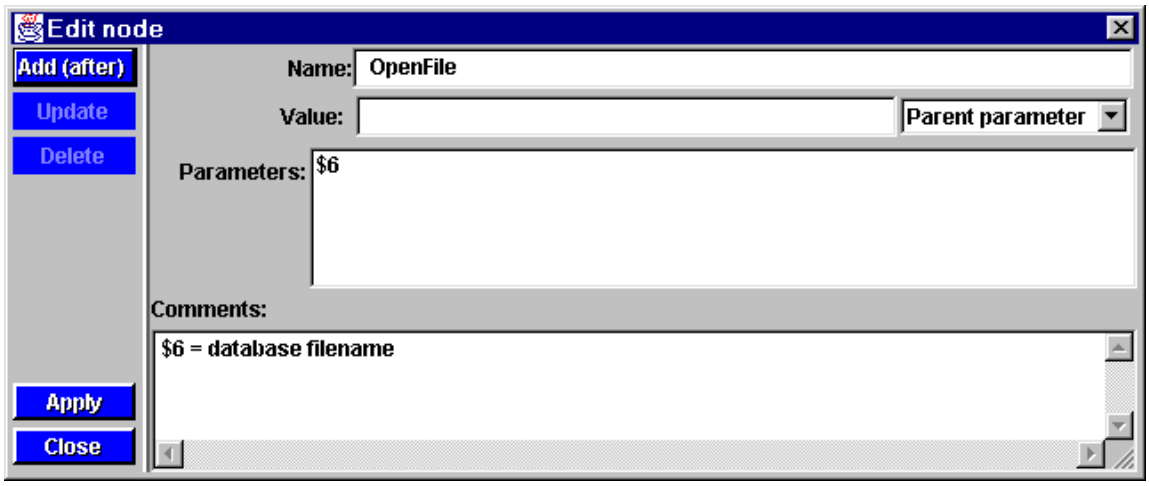


Figure 6-4 Edit Node dialog for `OpenFile`

8. **Click** Apply **and click** Close**.**

You have now set up this node of the test so that it takes the seventh argument passed to `Acceptance.jst` and passes it to its own script. The script, in turn, refers to the single passed parameter as `args[0]`.

Editing the AddRecord .jst Node

Now you need to edit the AddRecord node to prepare it to pass the correct parameters to EnterFieldData.

1. **Select the** `AddRecord.jst` **node and click the** Edit **button.**
   The Edit Node dialog opens.

2. **In the Edit Node dialog window, select** Parent parameter **from the pull-down menu.**
   The Value field changes to Argument #.

3. **In the** Argument # **field, type** `0`.

4. **Click the** Add (after) **button.**
   This adds the parameter to the list.

5. **Add arguments** `1` **through** `5`.
   Type the next number, then click Add (after). Continue until you have six parameters, numbered `0` through `5`.

   If you make a mistake, you can correct it by double-clicking on the parameter from the Parameters list and using the Update and Delete buttons.

6. **In the** Comments **field, list each argument and the field it maps to.**
   This is an important step—if you don't note the purpose of each argument here and later forget, you'll need to read the script code to decipher it.

   Type:

```
$0 Name
$1 Address 1
$2 Address 2
$3 Phone
$4 Email
$5 Other
```

You're going to need to use these same comments in other nodes. You can save yourself some typing by selecting this text and copying it to the clipboard for later use.

The Edit Node dialog should look like this:

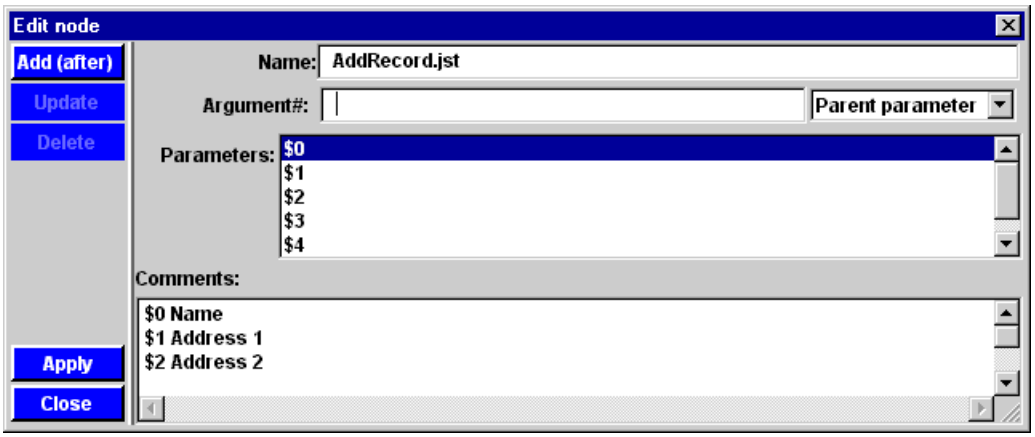


Figure 6-5 Edit Node for `EnterFieldData`, after parameters added

7. **Click** Apply**.**
8. **In the Edit Node dialog, click** Close**.**
9. **Click** Save **to record changes to** `Acceptance.jst`**.**
   JavaStar informs you that `Acceptance.jst` exists and asks you for confirmation to overwrite the file.
10. **Click** OK**.**

Editing EnterFieldData Node

1. **With the** `Acceptance.jst` **open, open** `AddRecord.jst`**.**
   Click on the `AddRecord.jst` node to select it. Click Open.
2. **Select the** `EnterFieldData` **node and click the** Edit **button.**
   The Edit Node dialog opens.
3. **Add arguments** `0` **through** `5`**, one for each of the text fields.**
4. **Add a comment describing what each of these parameters maps to.**
   If you copied the comments from EnterFieldData to the clipboard, you can paste the text in.
5. **Click** Apply **to record your edits.**

6. **Close the Edit Node dialog.**
   Click Close.

7. **In the Test Composer, click** Save**.**

Editing VerifySearch.jst

Because you have the `Acceptance.jst` open already, it is easiest to define parameters for `VerifySearch.jst` next.

1. **In** `AddRecord.jst`**, click the** Back **button to navigate back to** `Acceptance.jst`**.**
   The Back button is located in the Test Composer, above the Comments field.

1. **Select the** `VerifySearch` **node and edit.**

2. **Add six parent parameters (0-5).**

3. **Paste or re-type the comments you used for the EnterFieldData node.**

4. **Click** Apply**.**

5. **In the Edit Node window, click** Close**.**

6. **In the Test Composer, click** Save**.**

Editing the DefineSearch Node

Now you need to edit the nodes within `VerifySearch.jst`. You can start with DefineSearch.

1. **Navigate to** `VerifySearch.jst`**.**
   With the VerifyName node still selected, click the Open button. This loads `VerifySearch.jst`.

2. **Edit the DefineSearch node.**
   Click on `DefineSearch` node. Click the Edit button.

3. **For** Argument #**, type** 2**.**
   This test uses the value for the Address2 field, which is the third parameter (`args[2]`) you pass to `VerifySearch.jst`.

4. **Click** Add (After).

5. **In the Comment field, type** `Value to search for`**.**

6. **Apply the changes and close the window.**

Editing the GetSearchResults Node

1. **Still in** `VerifySearch.jst`**, click on** `GetSearchResults` **node.**
2. **Edit the** `GetSearchResults`**.**
   Click on `GetSearchResults` node. Click the Edit button.
3. **For** Argument #**, type** `0`**.**
   This test uses the record name, the first parameter passed to `VerifySearch.jst`.
4. **Click** Add (After).
5. **In the Comment field, type** `Name to select`**.**
6. **Apply the changes and close the window.**

Editing the VerifyRecord Node

1. **Click on** `VerifyRecord` **node.**
2. **Click** Edit**.**
3. **Add arguments 0-5.**
4. **Add comments to describe the six arguments.**
5. **Apply the changes and close the window.**
6. **Save the JST.**
7. **Close the Test Composer.**

The acceptance test is now set to take six parameters at the command line and pass these down to the three scripts you edited.

Running a Test With Arguments

Running tests with arguments is virtually the same as running a test without arguments. The only difference is that if your test inherits parent parameters from the top level of the JST, you need to supply these parameters in the Test arguments field of Run Test.

1. **Refresh the test database.**
   Delete `test.db`. Copy `sesame.db` to `test.db`.
2. **From the JavaStar main menu, select** Run test**.**
3. **For** Test name**, enter** `Acceptance.jst`

4. **In the test arguments field, enter the name of the argument strings you want to pass.**
   Specify five strings (values Name, Address 1, Address 2, Phone, Email, Other—in that order) in this field. Put strings in quotes and separate with a space.

   For example:

   ```
   "Elmo" "45 Sesame St." "Anytown USA" "555-2857"
   "tickleme@elmo.com" "none" "test.db"
   ```

   Do not include line breaks in the parameter list.

   Note – The quotation markes around each argument string aren't really needed if the argument string contains no spaces or special characters. In this case, quotation marks are needed for the second and third strings because they contain spaces, but all other arguments could be typed without the quotation marks. Special characters include `\n`, `\r`, `\t`.

5. **Click** Start.

6. **When the test finishes, quit the Record/Playback window.**

Viewing the Results

The log file for parameterized tests includes the values of the parameters you passed at runtime. To view the parameter information for this test run:

1. **From the JavaStar main menu, select** Show results**.**

2. **Load the log file from the test you just ran.**
   If you used the default log name, it will be called `Acceptance.jst.log`.

3. **Expand the JST to see all the nodes.**
   Click on the plus signs to open nodes.

4. **Select a node where you passed parameters.**
   Note that when you select a node, the Results Viewer shows you what parameters were passed.

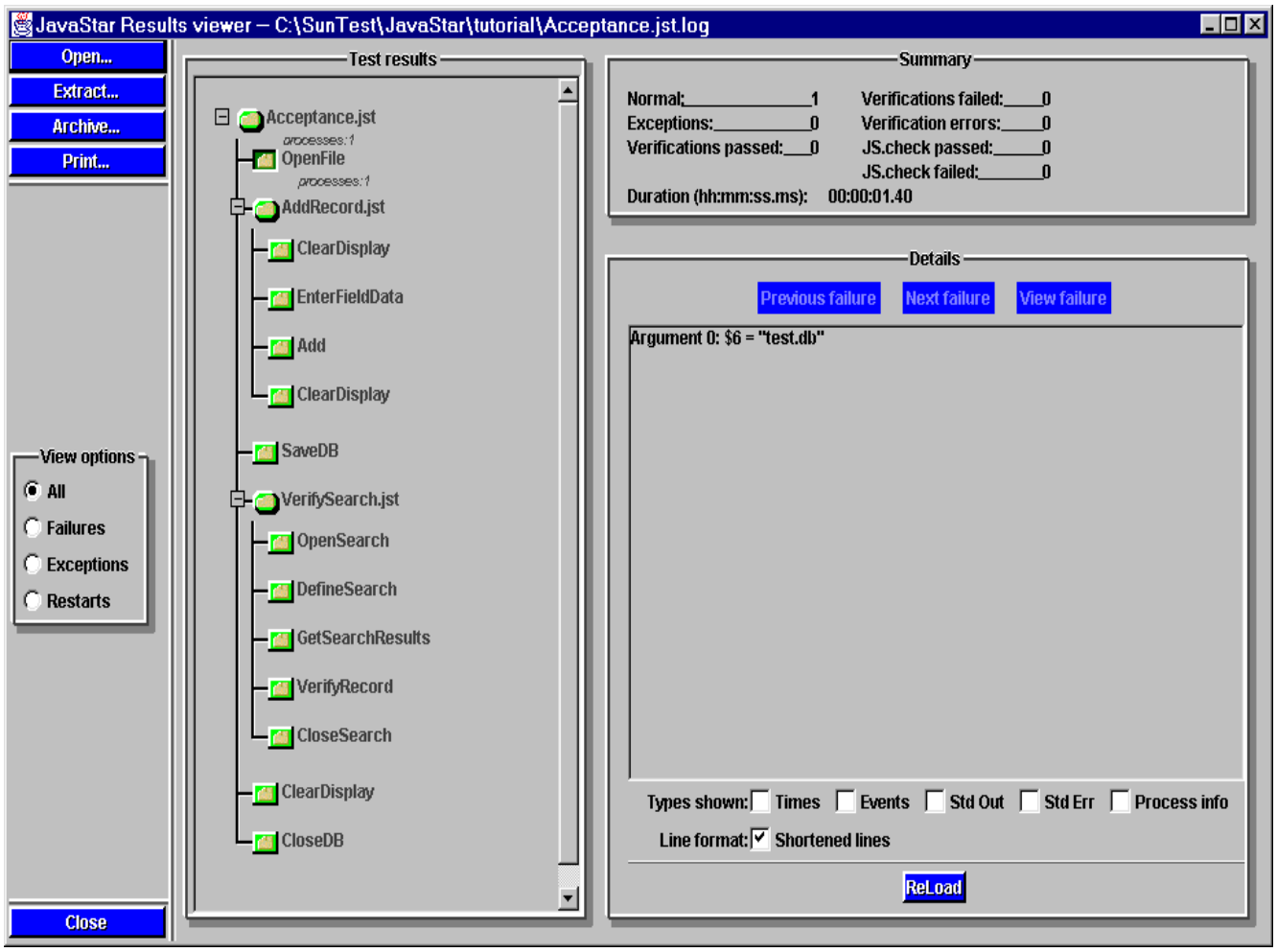


Figure 6-6 Results for `Acceptance.jst`, showing the argument mapping for a node.

Other Possibilities for Adding Parameters

Parameters aren't only useful for data entry and comparisons—you can also use them to determine the component in a window you want to take action on.

For example, in the acceptance test are three nodes—ClearDisplay, Add, and Close—that click a button in a window. In their current states, these scripts are re-usable (in fact, the test uses ClearDisplay three times) but only if you want to click that particular button in that specific window.

To make a more versatile script to replace these three, you can write a script that clicks a button in a window. The label of the button to click and the name of the frame where the button should be can be passed as parameters. This reduces three scripts to one (easier to maintain) and make the single script highly re-usable.

Using Property Files as a Source for Arguments

This lesson focused on using parent parameters for test arguments, but another possibility is using a Java property file. A property file is simply a text file containing a list of variables and their assigned values. You can create a single property file that holds as many test arguments as you want, then reference them using the Property Name option in the Edit Node window.

This exercise shows you how to modify your existing test to read the field arguments for the database from a property file. The remaining lessons in this tutorial use the version of the test you just finished, so be aware that you'll need to switch back to the previous version when you're done. You can copy the files from the `tutorial/modular` directory or create your own backup.

Reading a Single Property

If you want to read a single property or single set of properties (where a set is one value per property for a series of property names) you can do this easily by setting up your script to accept arguments and editing the JST node for that script to accept properties as parameters.

In the case of the acceptance test example, you might want to modify your test so that the script that enters a record (EnterFieldData) gets the data from a property file, instead of by inheriting parent parameters. That way, you wouldn't need to enter the field values each time you run the test—you would only need to change the property file when you wanted to use different data.

Part of the work of modifying the acceptance test is already done, because earlier in this chapter you edited EnterFieldData to accept arguments. You had also edited the EnterFieldData node of AddRecord.jst to read parent parameters. Now, you only need to create a property file and modify that node in the JST to read properties instead of parent parameters.

Creating a Property File

A property file is a text file that uses the format:

```
<property name> = <value>
```

For this example, you'll create a property file, that uses the format:

```
name = <name value for first record>
address1 = <address1 value for first record>
address2 = <address2 value for first record>
telephone = <telephone value for first record>
email = <email value for first record>
other = <other value for first record>
```

1. **Create a text file that contains the text:**

```
name=Elmo
address1=45 Sesame St.
address2=Anytown USA
telephone=555-2857
email=tickleme@elmo.com
other=none
```

2. **Save the file (in the** `tutorial` **directory) as** `FieldData.prop`.

Editing a Node to Read Properties

1. **In the Test Composer, load** `AddRecord.jst`.
2. **Select the** `EnterFieldData` **node and click the** Edit **button.**
   Because you edited the EnterFieldData node earlier in this chapter, it will contain a list of parent parameters, $0 through $6, one for each field of a record.
3. **Click on the first parameter ($0) in the list.**
4. **In the pulldown menu to the right of the** Arg # **field, change** Parent parameter **to** Property name**.**
   Note that the Arg # field changes to the Property name field.
5. **In the** Property name **field, change** `0` **to** `name` **and click the** Update **button.**
6. **Repeat this process for the remaining parameters.**
   Change each parameter from Parent parameter to Property name. Edit the arguemnts using the data:

For Arg #...	Set property name to...
1	address1
2	address2
3	telephone
4	email
5	other

7. **Update the contents of the** Comments **field.**
   Select and delete the existing text. Replace it with:

   ```
   Read properties from FieldData.prop
   ```

8. **Click** Apply **and click** Close**.**

9. **In the Test Composer, click** Save**.**

Your next step would be to edit the remaining parameterized nodes to read from the property file, but for the purposes of simplicity, this exercise focuses soley on the EnterFieldData node.

Specifying the Property File at Run Test Time

The Run Test dialog contains a field named Property File. When you run the acceptance test, be sure to enter `FieldData.prop` as the property file name. From the command line, use the `-prop` option.

You still need to pass the same test arguments described in the section, "Running a Test With Arguments." This is because. for the sake of brevity, this exercise modifies only EnterFieldData to use properties. DefineSearch, GetSearchResults, and VerifyRecord still use parent parameters. If this were a real test case, rather than an example, you would modify these three tests to use properties. Once you had done this, you would only need to pass the name of the test database for OpenFile.

Reading Multiple Properties

If, instead of reading a single property, you want to read a series of property values (for the same property name) you can do this by writing Java code that uses the `JS.getProperty()` method to retrieve values from the file.

In the case of the acceptance test, this would be useful if you wanted to populate the test database with multiple records, instead of using the property file to get data for just one record.

You can modify the tutorial example by:

- Creating a Property File with Sets of Data
- Moving the Code that Inserts Data into a New Script
- Editing EnterFieldData to Read Properties from the File
- Clearing Parameter Settings from the JST Node
- Specifying the Property File at Run Test Time

Creating a Property File with Sets of Data

Create a property file (or, if you already have a file named `FieldData.prop`, edit the existing file) to use the format:

```
total = <total number of records>
name_<#> = <name value for first record>
address1_<#> = <address1 value for first record>
address2_<#> = <address2 value for first record>
telephone_<#> = <telephone value for first record>
email_<#> = <email value for first record>
other_<#> = <other value for first record>
```

where you increment <#> for each set of records.

1. **Create a text file that contains the text:**

```
total = 7
name_0=Elmo
address1_0=45 Sesame St.
address2_0=Anytown USA
telephone_0=555-2857
email_0=tickleme@elmo.com
other_0=none

...
name_6=Susan
address1_6=101 Learning Lane
address2_6=Anytown USA
telephone_6=555-0277
email_6=susan@sesameSt.com
other_6=Educator
```

   Where you replace "..." with five addition records (following the numbering format) containing your own data.

2. **Save the file (in the** `tutorial` **directory) as** `FieldData.prop`.

Moving the Code that Inserts Data into a New Script

Before you change EnterFieldData.java to read properties directly from the property file, you need to save the original contents of EnterFieldData—where you actually insert the data into the record—to another file. For this example, you'll use `Insert` as the filename.

When you later edit EnterFieldData, you'll provide new code that calls Insert, passing the values for a single record set as parameters.

To create the `Insert` script:

1. **From the JavaStar main menu, choose** Edit Test Script**.**

2. **Open the file `EnterFieldData.java`.**

3. **Change the script name from `EnterFieldData` to `Insert`.**
   This is similar to doing a "save as" operation in other applications.

   a. **Click on the** Find/Replace **button.**
      The Find / Replace dialog is displayed.

b. **In the** Find **field, type** `EnterFieldData`**.**

c. **In the** Replace **field, type** `Insert`**.**

d. **Click** Replace All**.**
JavaStar replaces all occurrences of `EnterFieldData` with `Insert`.

e. **In the** Script name **field (located at the top of the window) change the name to** `Insert.java` **and click the** Save & Compile **button.**
This saves the modified file as `Insert.java`, but leaves the original `EnterFieldData.java` file intact.

4. **Add code that calls the play() method of Add**

```
Add a = new Add();
a.play(new String[1]);
```

Delete Add Node from AddRecord.jst

Now that Insert handles the add operation for each record, you need to delete the Add node from AddRecord.jst and reconnect the remaining nodes.

1. **In the Test Composer, open** `AddRecord.jst`**.**
2. **Select the Add node and click the** Delete **button.**
3. **Select the** `EnterFieldData` **node and click the** Start Normal **button.**
4. **Click on the** `ClearDisplay` **node to complete the connection.**
5. **Save the JST file.**

Editing EnterFieldData to Read Properties from the File

Now you need to edit EnterFieldData so that, instead of being passed property values, it reads the properties directly from the property file.

1. **In the JavaStar Script Editor, open** `EnterFieldData.java` **again.**
2. **Replace the body of the** `play()` **method with code that reads properties from the property file and calls** `Insert()`**, passing the properties as arguments.**
Here's one example of how you might edit the `play()` method.

```
public void play(String[] args) throws Throwable {

   Insert o = new Insert();
   int total = 0;

   total = Integer.parseInt(JS.getProperty("total"));
   // read total # of records
```

```
    for(int i = 0; i < total; i++) {
     String name = JS.getProperty("name_" + i);
     String address1 = JS.getProperty("address1_" + i);
     String address2 = JS.getProperty("address2_" + i);
     String telephone = JS.getProperty("telephone_" + i);
     String email = JS.getProperty("email_" + i);
     String other = JS.getProperty("other_" + i);

     if (name == null || address1 == null || address2 == null||
     telephone == null|| email == null|| other == null)
     throw new NoSuchFieldException("Invalid Key Value");

   // construct args
   String[] record = new String[6];
   record[0] = name;
   record[1] = address1;
   record[2] = address2;
   record[3] = telephone;
   record[4] = email;
   record[5] = other;
   // call play method of insert with args
   o.play(record);
    }
}
```

This example code:

a. Reads the number of “records” (sets of properties corresponding to a record) in the property file.

b. Creates a loop that repeats for the number of records to be read and:

   i. Reads each property for the current record being processed.

   ii. Checks to see if any of these return null, and if so, throws an error.

   iii. Creates an array containing the properties for a single record.

   iv. Calls `Insert()` and passes the array of properties as a parameter.

From here, the Insert() method handles inserting the data into a `namedb` record and adding the record to the database.

3. **Click** Save & Compile**.**

4. **Close the Script Editor.**

Clearing Parameter Settings from the JST Node

1. **From the JavaStar main menu, select** Compose Test**.**
   The Test Composer is displayed.

2. **Open** `Acceptance.jst`**.**

3. **Click on the** `AddRecord.jst` **node, then click the** Edit **button.**
   The Edit Node dialog opens.
4. **Click on the first parameter in the list.**
5. **Still in the Edit Node dialog, click the** Delete **button.**
   This deletes `$0` from the parameter list.
6. **Delete the remaining parameters** `$1` **through** `$6`**.**
7. **Delete the contents of the** Comments **field.**

Editing the Test to Read the Properties Directly

Because your script is now reading parameters directly from the property file, you don't need to pass the data as parameters using the JST. To edit the EnterFieldData node

1. **In the Test Composer, open** `AddRecord.jst`**.**
2. **Edit the EnterFieldData node to remove all parameter settings.**
   You can do this either by deleting the node and recreating it (with the same connections to the other AddRecord nodes) or, to edit using the Edit node window:
   a. **Select the** `EnterFieldData` **node and click the** Edit **button.**
   b. **In the Edit Node window, select a parameter from the list, then click the** Delete **button.**
   c. **Repeat** Step b **for each remaining parameter.**
   d. **Delete the contents of the** Comments **field.**
3. **Click** Apply**.**
4. **In the Edit Node window, click** Close**.**

Specifying the Property File at Run Test Time

As with the case of reading a single set of properties, you need to define a property file name at run test time. In the EnterFieldData code, `JS.getProperty()` reads the property you specify from whatever property file you specify when you run the test.

If you're running the test using the Run Test dialog, type `FieldData.prop` into the Property File field. If you are running the test from the command line, use the `-prop` option.

Note that you still need to pass the same test arguments described in the section, "Running a Test With Arguments." This is because. for the sake of brevity, this exercise modifies only EnterFieldData to use properties.

DefineSearch, GetSearchResults, and VerifyRecord still use parent parameters. If this were a real test case, rather than an example, you would modify these three tests to use properties. Once you had done this, you would only need to pass the name of the test database for OpenFile.

As the test executes, you can see that EnterFieldData types seven records into the database entry fields, and executes a button click on Add after it finishes entering each record.

Summary

Now that you know how to make your tests more versatile with parameters, completing your introduction to the JavaStar model. The next chapter shows you how to use JavaStar from the command line—a useful feature if you plan to run your tests automatically using shell scripts or batch files.

Exercise: Adding Parameters to the VerifyNames tests

This test builds on the optional exercise from the last chapter.

Instructions

This time:

1. Edit the scripts you wrote for `VerifyNames.jst` (or whatever you called your Names test) and add parameters.
2. Modify `VerifyNames.jst` to support parameter passing.
3. Incorporate `VerifyNames.jst` into `Acceptance.jst`, naming the modified JST `Acceptance2.jst`.

If you did not do the exercise for the last chapter, copy the following files from `\tutorial\modular`:

```
VerifyNames.jst
OpenNames.class and OpenNames.java
SelectName.class and SelectName.java
VerifyRecord.class and VerifyRecord.java
CloseNames.class and CloseNames.java
Acceptance2.jst
```

Solution

This solution uses a modified version of `Acceptance.jst` from this lesson and saves that as `Acceptance2.jst`. If you did the exercise this way, you didn't have to add all the parameter information for `AddRecord` and `VerifySearch` to the previous version of `Acceptance2.jst`.

Only one script needed parameters to work within the current model, and that's `SelectName`. This solution changes the `SelectName` node within `VerifyNames.jst` to inherit `args[0]` (the name field of the record) from `Acceptance2.jst`. You can see the solution shows the `SelectName` script referencing `args[0]` in place of `"Count von Count"`. If you did this, you also should have removed the position reference in the `select()` call to make the script more versatile.

`VerifyRecord` is the same script already modified earlier in this chapter, so you only needed to change `VerifyRecord` node within `VerfiyNames.jst` to inherit `args[0]` through `args[5]`.

Changing the `VerifyNames` node in `Acceptance2.jst` to also inherit `args[0]` through `args[5]` completes the solution. You can view these files in the `\tutorial\parameters` directory.

JavaStar from the Command Line 7

This chapter introduced you to the JavaStar command line options and shows you how you can use them to control your environment, run tests, filter logs, and more.

Topics:

About this Lesson

Part of the benefit from creating automated scripts is that you don't have to be present to run them. So far, you've selected all JavaStar options and started execution from inside the JavaStar GUI. This requires your interaction, and probably isn't how you'd want to work on a daily basis. Now you'll take a look at how to run JavaStar from the command line, using the wealth of command line options. From there, you can create a shell script or batch file to run tests in sequence (or repeatedly, with parameter changes) and analyze the results later.

Setting Up for this Lesson

1. **Copy the original `sesame.db` database over `test.db`.**
2. **If you did not do the lessons in the previous chapter, or if you no longer have those files in your `tutorial` directory, copy the contents of the `parameters` directory into the `tutorial` directory.**
3. **Launch JavaStar.**

Using Command Line Flags

Each of the playback features provided by the JavaStar GUI can be set through command line flags. The best way to see how this works is to map a test run to a command line sequence.

In the GUI, you can define a test run that:

- Runs the `Acceptance.jst` test
- Sends six string arguments to the test
- Shows both the program under test and the JavaStar Record/Playback window while running.
- Uses a log file named `accept1.log` instead of the default

You can do this in the Run Test window by:

- Specifying the test name and the six arguments in the fields under General
- Selecting Show Application and playback window under View (this is the default setting)
- Enter `accept1.log` in the Log filename field under Advanced.

At the command line, you would specify the same test run by typing:

```
java javastar -play -gui -jst Acceptance.jst -log accept.log
-testargs "Name" "Address 1" "Address 2" "Phone" "Email"
"Other" "test.db"
```

This command line sequence sets each of these options. The order is only loosely defined; the `-testargs` flag must always fall at the end, but the other options can be defined in any order. The commands set the options as follows:

Command	Description
`java javastar`	Starts JavaStar.
`-play`	Sets the mode to playback
`-gui`	Sets JavaStar to display the JavaStar GUI and the application under test during the test run.
`-jst Acceptance.jst`	Specifies that you are running a JST and the name is `Acceptance.jst`.
`-log accept.log`	Sets the logfile to `accept.log`. If you didn't set this, the log filename would default to `Acceptance.jst`.log.
`-testargs "Name" "Address 1" "Address 2" "Phone" "Email" "Other" "test.db"`	Passes the specified arguments to the test.

JavaStar provides controls that correspond to playback options, as well. These options are useful for controlling the environment in which the test runs. For example, the delay (`-scale`) and time out (`-timeout`) options are helpful when you run a test on a platform that has a slower response time than the system used to record the test. If your test involves interaction with a canvas, delay times can be critical to getting a proper response. But even if you recorded with delays on, your test might be too fast on a very slow system.

By scaling the delay factor up—increasing delay times—and by increasing the timeout value, you allow the system extra time to respond to the test events. This provides you with the flexibility you need to test in a wide range of environments.

Running the Acceptance Test

1. **If you have the JavaStar GUI running, exit JavaStar.**
   This isn't required—you can run the JavaStar GUI at the same time you run tests from the command line—but, for the purposes of example, it simplifies what you see on screen.

   Note – At this point, you should be using the NameDB application in the `namedb2` directory. Be sure `namedb2`, not `namedb1`, is in your `CLASSPATH`. You can check your `CLASSPATH` setting by typing

   ```
   java javastar -sysinfo
   ```

2. **In a UNIX shell or at an MSDOS prompt, enter the command to run the acceptance test.**
   Type:

   ```
   java javastar -play -gui -jst Acceptance.jst
   -log accept.log -testargs "Susan" "45 Sesame St."
   "Anytown USA" "555-2857" "susan@susan.com" "none"
   "test.db"
   ```

   There are no line breaks in a command line call to JavaStar.

   This command:

Action	Command Line Flag
Runs JavaStar	`java javastar`
Goes into Play mode	`-play`
Displays the JavaStar GUI	`-gui`

Action	Command Line Flag
Runs the acceptance JST	`-jst Acceptance.jst`
Records output to `accept.log`	`-log accept.log`
Sends test arguments for the `Susan` record	`-testargs "Susan" "45 Sesame St." "Anytown USA" "555-2857" "susan@susan.com" "none" "test.db"`

Filtering the Acceptance Test Log

If you're going to run tests automatically, it makes sense to generate reports automatically, too. Of course, JavaStar already does this by creating a log file, but this file contains *all* the information JavaStar records about the test—probably more information than you want. Using the log control options at the command line, you can manipulate this file during the test run or afterwards.

For this example, you'll filter the log for the test you just ran in the previous exercise.

1. **Filter the log to show only summary and time information.**
   At the command line, type:

   ```
   java javastar -logfilter ST accept.log filtered.log
   ```

   The `S` option of `-logfilter` specifies summary information, while the `T` option includes time stamps.

2. **Compare the sizes of the two log files.**
   The filtered log is only a fraction of the size of `accept.log`.

3. **Compare the contents of the logs in a text editor.**
   The original `accept.log` contains a lot of information—details of the test environments, each event executed, test arguments, and so on—that you might not need for an initial report. The filtered log contains only the summary for each test and the time it executed.

Summary

Running JavaStar from the command line is relatively simple. It's important to become familiar with the different options so that you can take advantage of them in automated test runs. For more details on how to ways JavaStar from the command line, see Using Command Line Options in the *JavaStar User's Guide*. If you want a complete list of the options availabe, either refer to the JavaStar Command Reference appendix to the *JavaStar User's Guide*, or `-help` option of JavaStar at the command line.

Part 2 — Advanced JavaStar

Using the JavaStar API 8

This chapter presents JavaStar tools and guidelines for using the added facilities found in the JavaStar Application Program Interface, or API. You use the API by adding your own Java code to existing JavaStar scripts or writing scripts from scratch.

While JavaStar provides you with fully-functional tests without requiring you to use the API, your tests will be stronger if you know what is available and use the methods to enhance your testing.

The JavaStar API includes classes and methods that assist test development. You can modify your scripts to use API features that aren't available when you record GUI interaction. You can also access custom components directly and call any public methods you need for verifying the state of your GUI or performing actions. Because Java is a powerful language, your tests can use more of the object-oriented features it provides when you add your own code.

In addition, some people prefer to write their tests from scratch. That way, they can add their own methods to the script class as they go along. If you're one of these people, you'll find JavaStar provides you with many tools to help you create flexible scripts. Before writing a script, however, you'll need to familiarize yourself with the script format that JavaStar expects.

After reading this chapter, you should know:

- some of the tools available for you in the JavaStar API
- the format of JavaStar scripts
- how to start customizing existing scripts
- what you need to do to create a script from scratch

In addition, you should know how to do these common functions using the API:

- verify that a menu component is enabled
- use any method of a component for verification
- share information among scripts
- keep scripts platform-independent when opening files
- replace label text errors found by `GUINotFoundExceptions` with internal checks that do not cause an exception

This chapter does not attempt to teach you Java—it assumes that if you want to add custom code, you're already familiar with the syntax of that code—but it does guide you through an exercise of adding Java statements to a script.

Topics:

About the JavaStar API

The JavaStar API contains interface classes, class static functions for playback, exception classes, and error classes. JavaStar uses this API to create scripts. All JavaStar scripts are implemented from the `Script` class of the API.

When you edit a JavaStar script or write your own from scratch, you can use this API to create powerful, consistent tests that work with JavaStar's log features.

The API class you'll probably use the most is the `JS`. This class contains a collection of static functions for use in playback. These functions allow you to perform a number of necessary functions, including:

- assertions—checking values as a test comparison (`JS.check()`)
- inserting delays into the test (`JS.delay()`)
- locating frames and dialogs, using exact titles or regular expressions (`JS.frame`, `JS.frameRX()`, `JS.dialog()`, `JS.dialogRX()`)
- getting and setting property values (`JS.getProperty()`, `JS.setProperty()`)
- getting and setting the typing rate (`JS.getTypingRate()`, `JS.setTypingRate()`)
- finding named components and menu components by name (`JS.lookup()`, `JS.mlookup()`)
- recording notes to the log file (`JS.note()`)
- inserting a break point in your test (`JS.pause()`)
- waiting for a condition you define to be reached (`JS.waitFor()`, combined with the `Waiting` interface)
- wrapping a component in a JSComponent (`JS.wrap()`)

Two other important classes are:

- `JSComponent`
- `JSMenuComponent`

You use `JSComponent` to send events to GUI components or to locate related components. You use `JSMenuComponent` to contain menu components so that you can test the contents.

You implement the `Waiting` interface when you use the `JS.waitFor()` method to define the condition you want to reach before continuing, similar to a custom synchronization.

Along with these classes, the API provides a number of exception and error classes that you can use to handle problems. For information on these classes, and for more details about those already described, see the *JavaStar API Reference.*

Anatomy of a Test Script

Each test script implements the `Script` class to define the application it needs to run, start-up information, argument details, and the actual test itself. When you create or edit a Script, you must be sure to maintain the Script structure, or your test may not compile or run. A Script is a Java program; it must compile and run in Java. In addition, you must ensure that the methods JavaStar requires are in place.

```
/* Generated by JavaStar Java GUI Testing Tool
 */


import suntest.javastar.lib.*;
import java.awt.*;
// You may add import statements here
public class EmptyScript extends Script {
private static String[] Args =
{"file:/D:\\JavaStar\\examples\\SwingSet\\Application\\SwingSetA
pplet.html"};
  public String[] getAppArgs(){ return Args; }
public String getAppClass(){ return
"suntest.javastar.applet.JSAppletViewer"; }
public void run() throws Throwable {
    suntest.javastar.applet.JSAppletViewer.main(Args);
  }
public void play(String[] args) throws Throwable {
//YOU ADDRESS THE API HERE
  }
}
```

When you edit a script or create one manually:

- If you change the name of the script, be sure to change the class name as well as the file name.
- Always include the four class methods (`getAppArgs()`, `getAppClass()`, `run()`, and `play()`).
- You can add import statements, but don't delete the lines that import the `suntest.javastar` or `java.awt` packages.
- Be sure that `getAppClass()` returns a string that matches the name of your application's main class. This is an issue only if you're creating the script manually.
- You can safely edit the `play()` method or add other methods.

In general, you only need to add import statements and edit or add to the `play()` method. If you generate an empty script through the Record / Playback window, all other required fields and methods will be in place.

An Example Application

There is an application called "TCTester" that you can use for this lesson. This application creates a test file to be used to test a temperature scale converter. To set up this application, create a project first.

Throughout this lesson you will be working in the tutorial directory. This directory is found in your <JavaStar home> directory. Please adjust the path for your installation. For example, in Windows, this may be:

```
C:\JavaStar\tutorial
```

1. **Change your working directory to**
   `<JavaStar home>/tutorial/API/TCTester/`
2. **Create a project for this tutorial where:**
   - application with its associated files is in:

   `<JavaStar home>/tutorial/API/TCTester/Application/TCTestCaseBuilder.class`
   - the tests, test paths, and JST is:

   `<JavaStar home>/tutorial/API/TCTester/Tests`
   - the test results are in:

   `<JavaStar home>/tutorial/API/TCTester/TestResults`
   - the declarations are in:

   `<JavaStar home>/tutorial/API/TCTester/Tests/TCTesterDeclarations/Main.class`

If you need help creating a project, please see "Setting Up Project Files".

Verifying Menu Components

After you have completed the project settings, create a script to test that the menu components are properly enabled. To do this, create a template for the script, then fill in the proper code using some of the API commands.

Let's see what happens when you try to select a menu component from the "Record / Playback" window.

1. **Click on** Create a Test Script**, and on** Start **to begin the application.**
   If your project settings are correct, the application, `TCTestCaseBuilder`, should be in place.
2. **Click on** Interact**.**
3. **From the application, pull down the** File **Menu. The** New**,** Open**, and** Exit **menu items are enabled.**
4. **Click** Inspect **to view the JavaStar code for these components.**
   The "Inspect" window does not show these components. It remains on the main application.
5. **Click** Quit **to close the Record/Playback window.**

Java 1.1.x AWT `MenuComponents` are specific to a platform. If JavaStar tried to address these components directly through the GUI itself, the resulting scripts would be platform dependent. Robust scripts should run on all platforms.

Because of this, there needs to be another means to address the `Menu Components`. There is: using the JavaStar API. You can use methods in the `JSMenuComponent` class provided with JavaStar to get the actual `MenuItem`, then use the `MenuItem's` methods to do the verification.

This is a common use for the API: obtaining the component itself, then checking information about the component that is not available to you from the verification function within the "Record / Playback" window.

Using the API to Obtain a Component

Although our example uses a menu component, parallel commands exist for a regular component. The menu class is `JSMenuComponent`; a regular component is `JSComponent`.

You can use the declarations generated for this application. To learn more about generating declarations, see "Generating Declarations".

Here is an example of locating a menu component:

```
JSMenuComponent newMenuComponent;
newMenuComponent = TCTesterDeclarations.Main.New();
```

This is not the AWT `MenuItem`, though. Use the `getValidUnique()` method of `JSMenuComponent` (or `JSComponent`) to extract the actual component from the JavaStar wrapper. As `getValidUnique()` returns a generic `Component`, you must downcast the component to the proper class.

```
MenuItem newMenuItem = (MenuItem) newMenuComponent.getUnique();
```

Once you have the `MenuItem`, you can use its methods to verify. However, you'll need a means to report the results of that verification to the test.

Using an Internal Verification

You can make output assertions from within a script using the `JS` static method, `JS.check()`. `JS.check()` takes two parameters: a `boolean` that is the condition you want to check, and a `String` that represents the purpose of the check.

The initial condition should be presented such that if it is *true*, the check will pass. For example, you assert that if the New menu item is enabled, this check should pass. You do this in code by writing:

```
JS.check(newMenuItem.isEnabled(), "New should be enabled");
```

Now that you've got the idea of what is needed, let's review the mechanics of getting this code into the script.

Step 1: Creating the Base Script

You could code the entire script by hand, but it would be subject to errors of omission for the items JavaStar provides that you shouldn't have to think too much about. It's easier to use the record features of JavaStar to generate a base script.

If you were really verifying the state of the application at start-up, you'd probably do the verifications of all buttons and text fields directly from the Record / Playback window, then edit the script to add the menu verifications. Feel free to do that if you have the time. Our directions are just for how to do the menu verification.

1. **Click on** Create a Test Script**, and** Start **the application.**
2. **Record the script. Name it** `VerifyFileNotOpen`**.**
3. **Immediately** Stop **the recording.**
4. Quit **the "Record/Playback" window.**

This created a script with an empty `play()` method, except for obtaining a component. Now you need to edit that script to put in the code to verify the menu item.

Step 2: Edit and Test the Script

Use the Script Editor within JavaStar to enter the required code. See "Editing Tests" in the *JavaStar User's Guide* for more information on using this editor.

1. **Click** Edit Test Script**, then browse to locate this script:**

   `tutorial/API/TCTester/Tests/VerifyFileNotOpen.java`

2. **Locate the** `play()` **method.**

3. **Add the code to verify that the** New **menu item is enabled:**

   ```
   MenuItem newMenuItem;
   JSMenuComponent newMenuComponent;
   newMenuComponent = TCTesterDeclarations.Main.New();
   newMenuItem = (MenuItem) newMenuComponent.getValidUnique();
   JS.check(newMenuItem.isEnabled(), "New should be enabled");
   ```

4. **If you want, delete the unnecessary declaration for the button.**

5. **Click** Save and compile**.**

6. **Run the script (you can do this from the editor by clicking** Run Script**).** It should run successfully with one passed check.

7. **Click** Quit **to close the Record/Playback window and** Close **to close the editor.**

Now that you've got a start on using the API, try a lesson that is a bit more complicated.

Opening Files

If you have been working on the other lessons in this tutorial, you've spent a lot of time on `OpenFile`. This lesson has a different application, but still has a file to open, so you're going to spend more time; however, you can build on what you've learned and use methods in the API to make your file handling even more rigorous.

You are going to work with these methods in this part of the lesson:

- `JSComponent.relativefile()` and `JSComponent.choosefile()`, which simulate file dialogs
- `JS.note()`, which adds messages to the test results file
- `JS.getProperty()` and `JS.setProperty()`, which share data between scripts

Examining the Recorded Open

Start by recording a script that opens a file.

1. **From the main JavaStar window, click** Create a Test Script**, and click** Start **to run the application.**
2. **Click** Record**. Name the script** `openAction`**.**
3. **From the application, pull down the** File **menu, and select** Open**.**
4. **Open the file**:
   `tutorial/API/TCTester/newfilew`
5. **Verify that the label on the left reads "**`newfilew`**."**
6. **Stop the recording, and** Quit **the "Record/Playback" window.**

This is pretty much the same as you did before. You edited that script to replace the hard-coded file name with an argument. That is better, but you can do better still. Here are some of the problems that remain with a file-opening script, even if you make the file name a variable:

- The script may fail, due to either a test set-up failure (no such file exists) or an application failure, and it will not be evident which is the problem.
- The script remains platform-dependent if a relative path is hard-coded.
- It is very easy to break a script dependent on the relative position of the test file to the working directory.
- The script may fail with an obscure `GUINotFoundException`, which may disguise a very different problem.

Next, you will examine each of these in detail and see solutions available to lessen the problems with scripts that address files.

Separating Test Set-up Verification

Two things are being tested in the `openAction` script:

- that the test is in the proper initial state
- that the application works

For better reuse, use two scripts.

The first, `VerifyTestFile`:

- ensures the arguments have been entered
- checks that the test file exists where specified
- notes initial state characteristics about the file for use in other scripts, such as its size

The second, `openAction`:

- actually runs the application and sees if it can successfully open the file

If `VerifyTestFile` fails, it means that test file wasn't set up properly. If `openAction` fails, the application has a problem opening the file. This makes error assignment and debugging much clearer than leaving it all in one script. `VerifyTestFile` becomes very reusable for any application using that file.

Removing Platform and Path Dependence

Examine your `openAction` script by viewing it in the script editor.

1. **Click** Edit Test Script. **Browse to find:**

   `tutorial/API/TCTester/Tests/openAction.java`

2. **Scroll to the** `play()` **method. Notice the call to the file dialog simulation:**

```
JS.frame("Temperature Converter Test Input")
  .dialog("java.awt.FileDialog", "Open")
  .relativefile("..\\", "newfilew");
```

3. **Click** Close**.**

In a previous exercise, "Adding Parameters for Flexibility", you replaced the file name `"newfilew"` with an argument that is passed to the test at run time. While this improves the script and makes it more flexible, please note that the directory, the first argument to the `JSComponent.relativefile()` method, remains hard-coded. This string is dependent on two things, each of which makes this script fragile and less portable:

- The file is relative to the current working directory. This script can only be run in the exact same relative position to the test file.
- The path is platform-dependent. Notice in our example above, you see two backslashes ("\"). This script depends on a Windows environment to run.

To correct this, make the directory, as well as the file, an argument. It would be preferable to specify the path as absolute, rather than relative. You'll do this in a few moments.

Removing False Exceptions

Should the `openAction` script fail to open a test file that is known to exist, it is most likely a problem in the application code. The proper behavior is to note that error against the application.

However, as the code uses a variable label to show that the file has been open, the verify on this label will fail with a `GUINotFoundException`.

The ideal correction to this is to change the application code itself to use a `setName()` for this component. Then, if JavaStar is instructed to use component names, the text of the label will not become the actual name of the component.

However, the `TCTester` application does not use `setName()`. We can compensate by catching the `GUINotFoundException` within the script, and posting an error to the log by using `JS.check()`.

```
try {
  JS.frame("Temperature Converter Test Input")
    .member("java.awt.Label", filename)
    .verify(this,filename, "label should reflect file name");
} catch (GUINotFoundException oops) {
  JS.check(false, "File label not correct");
}
```

Now let's actually create the `VerifyTestFile` and `openAction` scripts.

Building `VerifyTestFile`

Create a template script and edit it to code the required instructions.

1. **Click** Create a Test Script**. Start the Application.**
2. Record**. Name the script** `VerifyTestFile`**.**
3. **Immediately click** Stop **to stop recording.**
4. **Click** Quit **to close the "Record/Playback" window.**
5. **Edit the script. Browse to open**:

   `tutorial/API/TCTester/Tests/VerifyTestFile.java`

6. **Import the IO classes you will need.**
   Add:

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

   to the import statements.

7. **Assign the two arguments to variables with better names. Validate to ensure the arguments are not empty or null. Put this code at the beginning of the** `play()` **method.**

   ```
   String fileDir, fileName;
   try {
        // find the arguments
        fileDir=args[0];
        fileName=args[1];
        // make sure they are not empty
        if (fileDir.length() == 0 || fileName.length() == 0) {
           JS.note("Empty argument found");
           JS.note("Directory: " + fileDir);
           JS.note("File Name: " + fileName);
           throw new IOException();
        }
     } catch (Exception oops) {
   ```

```
        JS.note ("Unable to resolve file name arguments");
        throw oops;
    }
```

`JS.note()` is the API method that allows you to post a string to the test result file. Here you use it to add a better explanation for what caused the exception.

8. **Enter the code to ensure the file exists. The code below has the required calls to the API.**

```
// verify the file is ok
     TestCaseFile tf = new TestCaseFile(fileDir,fileName);
     if (!tf.fileExists()) {
        JS.note ("Could not find " +fileDir + fileName);
        throw new IOException();
     }
```

9. **Gather other information about the file for use in later scripts.**
   It is better to get that data now, rather than try to access the file while the application has it open, as you may accidentally cause a problem trying to open it twice. Here is the code to gather the information through the file API.

```
// get the absolute path
     String absFileDir = tf.getParent();
     String fileSize = String.valueOf(tf.getSize());
```

10. **Place the file information in a property file that can be accessed by other scripts.**
    JavaStar maintains an internal property file that all scripts can access within any one execution of a test. To place items in this file, assign them a key name, and use `JS.setProperty(`*String keyname*`,` *String value*`)` to update the file. Here is the code that sets the values in the property file and notes to the log what those values are.

```
     JS.setProperty("FILEDIR", absFileDir);
     JS.setProperty("FILENAME", fileName);
     JS.setProperty("FILESIZE", fileSize);
     JS.note("Properties set: ");
     JS.note ("FILEDIR: " + absFileDir);
     JS.note ("FILENAME: " + fileName);
     JS.note ("FILESIZE: " + fileSize);
```

11. Save and compile **the script. Don't close the editor yet.**

Edit `openAction`

1. **From the Script Editor, browse to find the script:**
   `tutorial/API/TCTester/Tests/openAction.java`

2. **Locate the** `play()` **method. All your updates will go within the brackets of this method.**

```
public void play(String[] args) throws Throwable {
//YOU ADDRESS THE API HERE
}
```

3. **Get the file directory and file name properties, and assign them to variables. Place this code at the beginning of the `play()` method.**

```
// Code to get the properties we need
String filedir, filename;
filedir=JS.getProperty("FILEDIR");
filename=JS.getProperty("FILENAME");
```

4. **Locate the call to `relativefile()` and correct it.**
There is a parallel method, `choosefile()` that uses an absolute path. Change to this method, and use the values from the properties for the input parameters.

```
// change from relativefile to choosefile,
// from literals to properties
TCTesterDeclarations.Main.tCTestCaseBuilder()
  .dialog("java.awt.FileDialog", "Open")
  .choosefile(filedir, filename);
```

5. **Locate the verification of the label, and change it to use the variable filename.**

```
JS.frame("Temperature Converter Test Input")
  .member("java.awt.Label", filename)
  .verify(this,filename, "label should reflect file name");
```

6. **Enclose that verification in a try/catch loop that catches an exception and posts an error to the log instead.**

```
try {
  JS.frame("Temperature Converter Test Input")
    .member("java.awt.Label", filename)
    .verify(this,filename, "label should reflect file name");
} catch (GUINotFoundException oops) {
  JS.check(false, "File label not correct");
}
```

7. **Save and compile the script. Close the Script Editor.**

Now that you have built the scripts, you can build a test that contains them.

Build the JST

By creating a test that contains the pair, you have the effect of a single script similar to the `OpenFile` you created for the `Name` application. This test can be used within other tests, and the properties it sets can be addressed by the other scripts.

If you are not familiar with using the test composer, please see "Composing Tests" in the *JavaStar User's Guide*.

1. **Click** Compose Test**. Enter** `OpenTCFile` **as the jst name.**
2. **Add the scripts** `VerifyTestFile` **and** `openAction`**.**
3. **Map the first two jst arguments, 0 and 1, to the first two arguments of** `VerifyTestFile`**. Use the Edit button.**
4. **Create a normal flow between** `VerifyTestFile` **and** `openAction`**.**
5. **Save the test.**
6. **Close the Test Composer.**

Now you are ready to test your test.

Run the Test

1. Run Test**. Browse to find**:
   `tutorial/API/TCTester/Tests/OpenTCFile.jst`
2. **Enter the arguments** `/tutorial/API/TCTester` **for the directory, and** `fileOfThree` **for the file name. Adjust the directory path as needed to be correct for your platform.**
3. **Start the test. It should run to completion.**
   Examine the test results to see the notes you have made.
4. **Try the test several other times using different arguments, such as the absolute path for the directory, an invalid directory, an invalid file name, etc.**

At this point, you have learned to build a robust file-handling test.

Summary

You should now be well on your way to designing effective tests and finding the best ways to automate them. The next step is for you to take what you've learned and apply it to your application and your planned test scenarios. For help with how to perform specific tasks or to learn what various JavaStar options do, see the *JavaStar User's Guide*. For more details on the JavaStar API, read the *JavaStar API Reference.*

Using Non-Component Locators 9

This chapter describes develop and use locators to access non-Java AWT components as part of your JavaStar tests. You may need a locator if GUI objects that you need to access as widgets are not visible to JavaStar.

The next two chapters deal with specific locators for the JFC, and with writing locators.

Topics:

About this Lesson

When you test an application that was created using a toolkit that does not extend Java AWT, you need a locator object to translate GUI location data for JavaStar. In the case of the Marimba Bongo, Netscape IFC, and the KL Group's JClass toolkits, JavaStar provides locators. For other toolkits, you need to write your own.

This lesson introduces the concept of locators and describes how to write your own. Because locators vary in implementation (the actual content of the methods used) depending on the toolkit, this lesson does not guide you through the detailed process of creating a specific locator. Instead, it provides the information you need to evaluate your toolkit and use the JavaStar API to create a locator appropriate to the toolkit.

After completing this lesson, you should be able to:

- Determine when you need a locator
- Use your own locator or a pre-existing one to record tests
- Use the JavaStar non-component classes to create a locator that conforms to the structure JavaStar expects
- Evaluate your toolkit to see what methods it provides to assist you in developing your locator

Using an Existing Locator

This section describes the steps to follow to use a locator while recording a test, but doesn't give you an exercise to run. This is because in order to run a locator and see it work, you need to have an application that uses non-components (the JavaStar term for widgets that JavaStar can't "see").

If you would like to do an exercise using a non-component locator, please take the next t utorial lesson, Testing JFC Components.

To use a locator when running a test, make sure the locator directory is in your Additional classpath setting.

This procedure assumes you have your application running and have the Record / Playback window open.

1. **In the Record/Playback window, click** Record**.**
   The Record test script window is displayed.
2. **In the** Test name **field, provide the name of your script.**
3. **In the** Non component locators **field, either type in the locators you want or click** Locator List **to browse for them.**
   If you type the locators in, be sure to give the package name (if any) and put a delimeter between each locator. On Windows systems, use a semi-colon as a delimeter; on UNIX systems, use a colon.

   If you use the Select Non-Component Locators dialog window to find the locators, you need to navigate to the correct directory using the Select an Item panel to the left. If you are using a JavaStar-supplied locator, or if you are storing your locators in the same directory as the JavaStar locators,

   a. **Click on the JavaStar folder icon along the top of the Select an Item panel.**
   b. **Click on the plus sign to expand the folder, then scroll down to find the** `contrib` **folder.**
   c. **Expand the contrib folder and double-click on the locators sub-directory to make it the only folder displayed.**
   d. **Expand the locators folder.**
   e. **Click on a locator you want to reference, then click the** Add to List **button in the Edit List panel to the right.**
   f. **Continue for any other locators you need to add.**
      See Figure 9-1 for an example of how the window should look.
   g. **Click** OK**.**

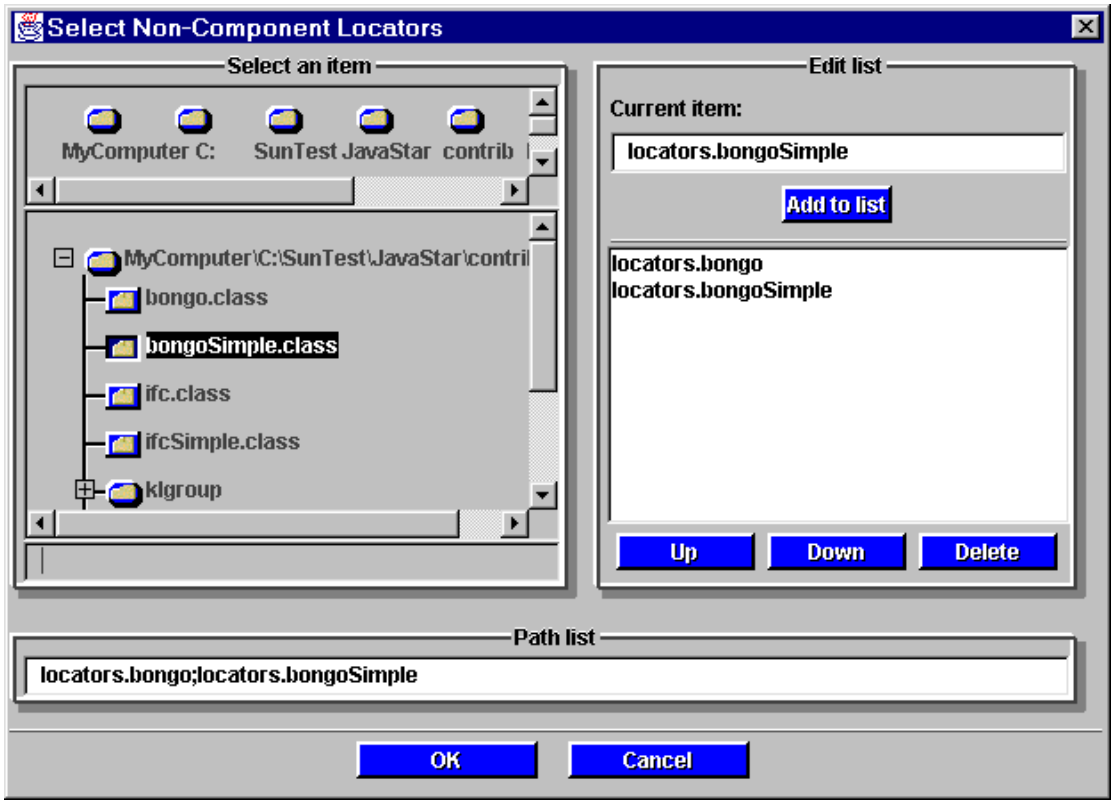


Figure 9-1 Select Non-Component Locators dialog window.

4. **If you need to specify files for** Record with map files**, or to toggle on** Record with delays**, do so now.**

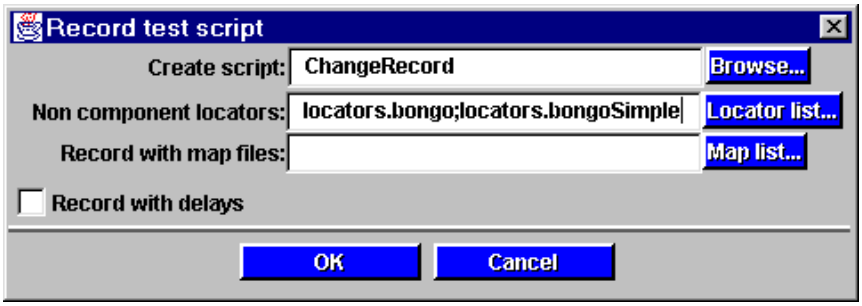


Figure 9-2 Record test script dialog with locator information provided

5. **Click** OK**.**

How a Locator Works

Non-Component Locators (NCLs) are objects of the `nonComponentLocator` class. This class contains two methods—`findObject()` and `getNamedObjectData()`—that you implement with code specific to the toolkit you are using.

This section describes:

- Recording a Script with an NCL
- Running a Test with an NCL

Recording a Script with an NCL

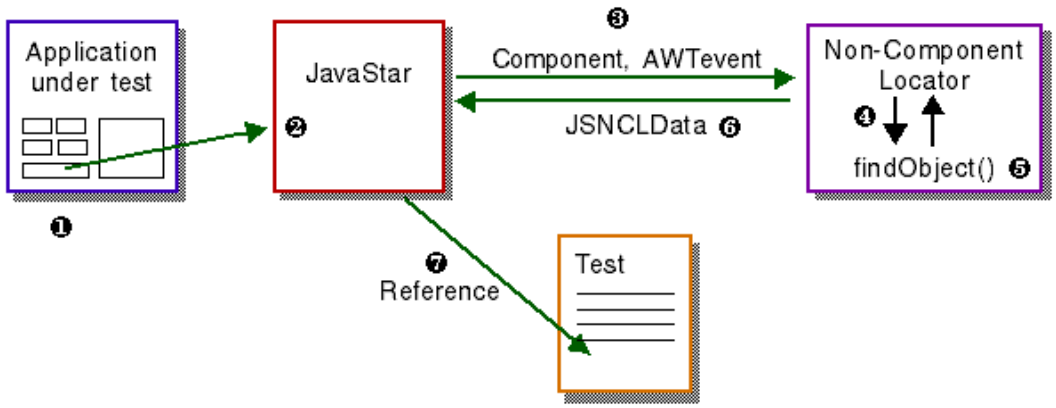


Figure 9-3 Recording with a non-component locator

As you record a script, JavaStar monitors your interaction with the application under test. If you defined your test to use an NCL as you record, the test is recorded using the procedure:

1. Test operator performs an event on a non-component.
2. JavaStar monitors the event, noting the parent `Component` (the parent deriving from the JavaAWT library) and the `AWTevent`.
3. JavaStar sends the `Component` and `AWTevent` to the NCL
4. The NCL runs the data through its own acceptance criteria, and if it passes, it calls `findObject(Component, AWTevent)`.
5. `findObject()` identifies the non-component as an object of the toolkit, retrieves the location of the object, then determines how best to refer to this object using a String.
6. The NCL returns the reference as part of a `JSNCLData` object.

7. JavaStar takes the non-component reference supplied by the NCL and inserts this code into the test currently recording.

Running a Test with an NCL

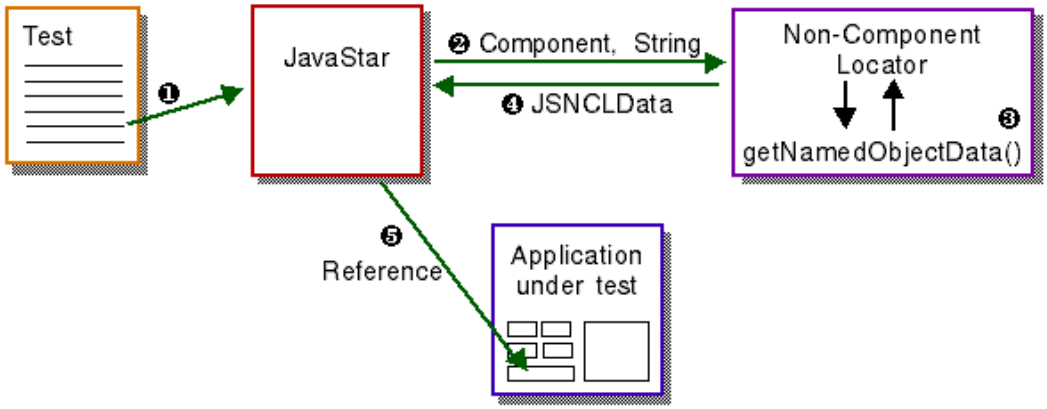


Figure 9-4 Running a test with a non-component locator

1. JavaStar receives an instruction from the test that references a non-component using a String.
2. JavaStar calls the `getNamedObjectData()` method for the NCL. It passes the Component and the String referencing the non-component, as provided by the test code.
3. `getNamedObjectData()` performs the reverse operation as `findObject()`—it uses the String passed in as an argument to determine the location (coordinates) of the object.
4. `getNamedObjectData()` sends the location information back to JavaStar inside an `NCLData` object.
5. JavaStar executes the event at the location specified.

Summary

Now that the basic concepts of an NCL are understood, you can take the next more detailed lessons on using and building NCLs.

Testing JFC Components 10

The Java Foundation Classes (JFC) are not officially part of the Java language until the release of Java 1.2. However, many people are developing projects using components based on those classes. The JFC is now available under the name Swing. JavaStar can be used to test applications and applets that contain Swing, or JFC, components.

This chapter allows you to record tests on an application that uses many of the JFC components. You use the provided Non-Component Locators.

After reading this chapter, you should know how to:

- verify a simple JFC component
- set up project to use the JFC
- use provided NCL's and text maps to address complex components
- obtain a JFC component and use the JavaStar API for specialized tests

Topics:

Setting Up to Test the JFC

Before you can do this tutorial, you must install Swing 1.0.2 or higher. This is needed to run any application using Swing components. You can locate this product at the Java web site. Be sure to update your classpath to include `<swing>/swingall.jar`.

We have copied the `SwingSet` application to this directory: `tutorial/API/TCTester/Application`.

You can set up a project to run the `SwingSet` applet.

1. **Set the** App **to the applet file URL:**

   `tutorial/API/TCTester/Application/SwingSetApplet.html`

2. **Establish the test working directory, JST path, and test path, as:**

   `tutorial/API/TCTester/Tests`

3. **Set the test results directory to:**

   `tutorial/API/TCTester/TestResults`

4. **Move to the** Mapping **pane. You will need to use a text map and a non-component locator to test this application. You will find both in the directory where you installed JavaStar. Edit the list for the NCL to include:**

   `<javastar>/contrib/jfc/newNCL.class`

5. **Ensure the text map list includes:**

   `<javastar>/contrib/jfc/JfcTM.class`

6. **Save the project by clicking on** Save As... **and saving it in:**

   `tutorial/API/TCTester/SwingSet.jpr`

Now that `javastar` can find the application, its needed Swing classes, and the necessary NCL's and text maps, you are ready to record a test.

Testing a Simple JFC Component

Actually, you don't need the NCL to test simple JFC components that don't contain other components. JavaStar can locate these components. To demonstrate this, test a slider to see if it is set at the correct value.

1. **Click** Create Test Script**. Use the application that you set up in the project. Click** Start**.**
   It takes a while for the `SwingSet` applet to load.

2. **Record the script. Name it** `testSlider`**.**

3. **Return to the** `SwingSet` **applet. Select the tabbed pane "**Slider**".**
   Notice that the NCL is used to locate a tabbed pane. Here is the JavaStar code:

   ```
   JS.lookup("Main SwingSet Panel").member
      ("com.sun.java.swing.JTabbedPane").getNonComponent
         ("jfc.newNCL","14:Slider").multiClick(17,15,16,1);
   ```

4. **Verify the value shown on a slider. Click** Verify **on the Record/Playback window. When asked to select an object, select any slider.**
   You should see the `JSlider` object in the verify panel. The code is something like this (it varies depending on which slider you chose).

   ```
   JS.lookup("Main SwingSet Panel").member
      ("com.sun.java.swing.JTabbedPane").member
         ("SliderPanel").member("com.sun.java.swing.JSlider", 6)
   ```

5. **JavaStar will suggest using the attributes in the gold file as the default. Click** Customize **to use a method of a** `JSlider` **to check just the value.** If you used the gold file, JavaStar would compare all significant attributes of the slider.

6. **Click on the** Using simple methods and data members **radio button. Click on** Select simple methods and data members **to see a list of the members.**

7. **Scroll through the list and select the method** `getValue()`**, and click** Enter a purpose**.**

8. **Click** Insert the verification into test**. The code in the script should be similar to this:**

```
JS.lookup("Main SwingSet Panel").member
   ("com.sun.java.swing.JTabbedPane").member
      ("SliderPanel").member
         ("com.sun.java.swing.JSlider",6).
   verifyAnyMethod(this,false,true,"getValue",new Integer(60),
                     "ensure the proper value");
```

9. **Click** Continue**, then stop the recording.**

10. **Playback the recording.**

As you can see, testing a simple JFC component is not very different from testing an AWT component. You can do everything you could with an AWT component. As you'll see in the next section on Menu components, you can do more with some items.

Testing Menus and Toolbars

Verifying the contents of an AWT menu required the use of the JavaStar API. However, you can verify JFC menus and toolbars while recording. This section allows you to try that verification.

1. **If you are still in the Record/Playback window with SwingSet running, you can just record a different script here. If you are not in that window, Click** Create test script **and start the applet.**

2. **Record a script. Name it** `testMenu`**.**

3. **Select the** Menus & ToolBars **tabbed pane from the Swingset applet.**

Check a Menu Bar Label

1. **Verify that the** Colors **menu item has the correct label. Click** Verify**, and select the** Color **menu.**

2. **JavaStar will suggest using the Gold File as a default. Since we just want to check the label, click** Customize**.**

3. **Select** Using simple methods and data items**, and click** Select simple methods and data items.
4. **Choose the method `getLabel()`, and click** Enter a purpose**.**
5. **Enter a purpose if you'd like, and click** Insert verification into test**.**
6. **Click** Continue**.**

Check a Menu Item's Mnemonic

Continue to record the test. Verify that the proper mnemonic was used for a menu item.

1. **Pull down the** File **menu on the tabbed pane. Use the sample menu on the pane, not the real one for the SwingSet applet.**
   Leave the menu pulled down, and return to the Record/Playback window.
2. **Click** Verify**. Return to the applet, and select the** Save **menu item.**
3. **Return to the Record/Playback window. Click** Customize**.**
4. **Select** Using simple methods and data items**, and select the `getMnemonic()` method.**
5. **Click** Enter a purpose**, and** Insert verification into test**.**

Check a Toolbar

The ability to use abstract buttons with images, rather than text, in them makes the creation of a button icon easy when using JFC components. You can verify these buttons. Verify to see if a button is enabled. This continues the test we are building, and selects a button on a toolbar within the Menus and Toolbar tabbed pane.

1. **Click** Verify**.**
2. **Select a toolbar button on the tabbed pane. We chose the** Paste **button at the end of the bar.**
   Notice the code that was generated. The button was labeled with the word "`paste`" rather than a sequential number. The text map we used chooses the tool tip for a label of some components.

   ```
   JS.lookup("Main SwingSet Panel").
      member("com.sun.java.swing.JButton", "paste")
   ```
3. **Return to the playback window, and click** Customize**.**
4. **Click** Using enabled**.**
5. **Click** Enter a purpose**, and** Insert verification into test**.**

6. **Click** Continue**, and stop the recording.**

7. **Playback the recording.**

Now try a more complex component that will use the non-component locator, and use the JavaStar API to test it.

Testing a Complex Component

To try a more complex component, take a look at the sample tree that is in SwingSet. Verify that the main root of the tree has more than one child. To do this, start a recording, locate the tree, then use the edit facility of the Record / Playback window to get the component.

1. **Record a new script. Name it `testTree`.**

2. **Select the tabbed pane "**TreeView**" on the SwingSet applet.**

3. **Return to the Record/Playback window. Click** Edit**. The script editor will appear in the window.**

4. **Select** Insert reference **from the bottom of the panel.**
   A dialog appears. This dialog lets you create a variable name, then select the component you'd like to reference.

5. **Name the variable by typing its name in the** Name **text field. Call it `theRoot`.**

6. **Select the component by returning to the applet and selecting the top of the tree, "**Music**."**
   You see the component appear in the reference window.

7. **In the Reference window, click** Apply**, and** Close**.**
   This inserts the code to get the component in the top of the `play()` method of the script.

   This is the code that is generated:

```
com.sun.java.swing.tree.DefaultMutableTreeNode theRoot =
(com.sun.java.swing.tree.DefaultMutableTreeNode)
   (JS.lookup("Main SwingSetPanel")
      .member("com.sun.java.swing.JTree")
         .getNonComponent("jfc.newNCL","0:Music")
            .getReference());
```

8. **Add the code to do the check. Place it beneath the reference.**

```
JS.check(theRoot.getChildCount() > 1,
     "root must have more than one child");
```

9. **Compile the code. When the compile succeeds,** Save **the script.**

10. **Click** Continue **to close the editor.**

11. **Stop the recording.**

12. **Playback the script.**

You can continue, if you'd like, testing any of the JFC components found in the SwingSet.

Summary

JFC Components can be tested using JavaStar and Swing. Most of the components can be addressed and used as any other components. Some require the use of Non-Component Locators and text maps that are distributed with JavaStar.

Writing Non-Component Locators 11

There are some components that JavaStar cannot identify without the use of a non-component locator (NCL). These include:

- third-party components, such as those provided by the KLGroup
- components that are not officially part of Java in this release, such as the JFC
- components developed by your organization

JavaStar provides non-component locators for the JFC and many of the third party components. However, you may need to write one yourself if you are using components that are not in the AWT.

This lesson explains in detail how NCLs work and walks you through the creation of an NCL.

Topics:

Understanding the Need for an NCL

Items that are directly contained by an AWT component do not require NCL's to be found. JavaStar can locate them. However, if those items themselves contain other sub-components, JavaStar needs more knowledge to determine what actual object is needed by the tester.

For example, let's consider a `JTree`, one of the Swing (soon to be JFC) components. As you saw in the last lesson, JavaStar could find the `JTree` without an NCL. However, it could not find a row, or node, of the tree. Without the NCL, testing could only be done on the tree as a whole.

Figure 11-1 is a screen shot of the tree we are going to work with for most of this lesson. It is from the SwingSet example distributed with Swing.

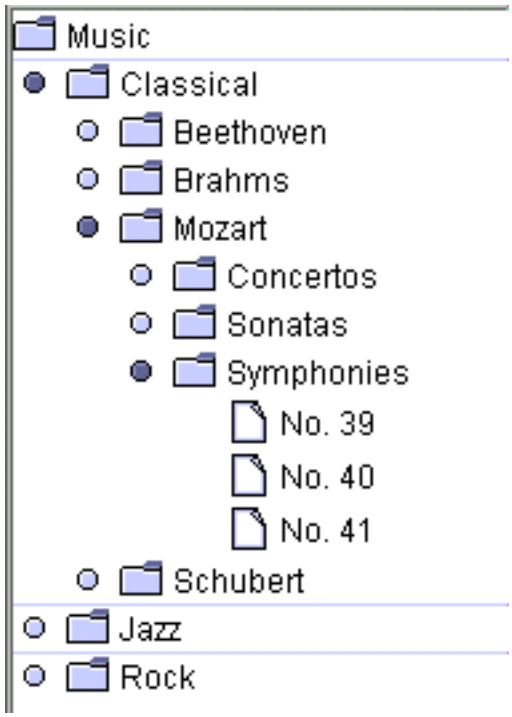


Figure 11-1 A `JTree`

Using `JSNCLData`

`JSNCLData` is the class that describes a non-component. Table 11-1 shows the definition of the class and the values for a single component, the Mozart line seen above.

Table 11-1 Data members of `JSNCLData`

Variable	Declaration	Value for "Mozart"
Name	`public final String` *Name*	"4:Music##Classical##Mozart"
P	`public final Point` *P*	P.x: 40 P.y: 72
Ref	`public final Object` *Ref*	An instance of the DefaultMutableTreeNode class.

The `Name` is the unique identifier for that component within the `JTree`. It is created by the NCL. The name serves two purposes. First, it provides a means to locate the component itself. In this case the number 4 is the index into the JTree that will locate that row. Second, the name is a means to confirm that the

component is the same as the one found at the original. The string "4:Music##Classical##Mozart" can be parsed. The string following the colon can be reconstructed into a path. This path must match the one found at index 4, or JavaStar will know that the component which was originally addressed no longer exists at that index.

The `Point` is the position of that sub-component within the containing component recognized by JavaStar. The containing component is the `JTree`. The upper-left location of the rectangle bounding the Mozart row is represented at that point. With this information, JavaStar can interpret any click on that component as relative to that point.

The `Reference` is the component itself or a means of locating the component. In the case of the `JTree`, it is the `DefaultMutableTreeNode` corresponding to the Mozart row. JavaStar can then inspect the available methods and data members of this component, and make them available for use in verification and synchronization.

Now that we've analyzed the `JSNCLData`, let's take a look at how one NCL, the `Tree.java` NCL, creates that data.

Using the `JSNonComponentLocator`

The purpose of a Non-Component Locator when recording is to interpret a click on a containing component, and provide its `JSNCLData`.

The purpose of a Non-Component Locator when playing back is to locate a component given its unique name, and return its current `JSNCLData`. If an entire tree is moved to a new location, JavaStar can still locate the proper component within that tree, as its bounds are relative to the tree. If the row is no longer at the 4th index position, however, JavaStar will not be able to find the component.

Anatomy of a `JSNonComponentLocator`

You can find the API for this class in the *JavaStar API Reference.*

There are two methods that must be provided:

1. `JSNCLData findObject(Component c, AWTEvent e)`
2. `JSNCLData getNamedObjectData(Component c, String name)`

Finding the `JSNCLData` *while Recording*

The `findObject()` method must determine what object is at a given mouse position. To do this it must:

1. **Ensure that the component and event are appropriate for this NCL.**
   JavaStar will pass the pair to each NCL on the list of locators until a value is returned or it has iterated the entire list. So the NCL for a `JTree`, for example, must be sure that the component passed is an instance of a `JTree`, and the event is a mouse event. Otherwise, it should return null. Here is the code in `Tree.java` that ensures this is a `JTree` and a mouse event:

```
if(e instanceof MouseEvent){
  int x = ((MouseEvent)e).getX();
  int y = ((MouseEvent)e).getY();
  if(c instanceof JTree){
    JTree t = (JTree)c;
    return findObject(t,x,y);
    }
}
return null;
```

Note the cast of the component to the `JTree` when it is certain that the component is a `JTree`.

2. **Locate the nearest component at this position.**
   The NCL must determine the component that is appropriate for that event, and at the position within the containing component. How it does that depends on the methods available in the component. For a `JTree`, the NCL uses the `getClosestPath()` method to locate the full path, then gets the row index for that path. Here is the code:

```
public static JSNCLData findObject(JTree t, int x, int y){

TreePath p = t.getClosestPathForLocation(x,y);
  if(p!=null){
    int r = t.getRowForPath(p);
    return getTreeDat(t,r);
  }
  return null;
}
```

3. **Create a unique name for the contained component.**
   There are two parts to the name: its location within the containing component, and a confirming string that can be used to ensure the same component is at that location. Think of the location as a street address of a home. Think of the confirming string as the names of the people in the home. To find people, you need to both locate their address and ensure they still live there. When a script is played back, that location must exist and that same component must be in that location.

   For the `JTree`, finding the location is easy. Mozart is the fourth row, and the NCL uses the integer 4 as the index. The confirmation is not quite as easy. Simply using the string “Mozart” may not ensure it is the correct component, as there may be many “Mozart” entries in the tree. To be unique, the string must contain the path to Mozart. The `Tree.java` uses the various methods available to get the row path, parse it to individual

objects, and concatenate a path, separated by the ## delimiter, of the "`toString()`" titles of each object. The resulting name, then, is 4:"Music##Classical##Mozart".

4. **Locate the position of the contained component within the container.** There needs to be a consistent way to state the boundaries of that component relative to its container. Figure 11-2 shows a logical rectangle around the row. The upper boundary of that rectangle is at point (40,72) within the `JTree`. A click at (44,84) lies within that row.

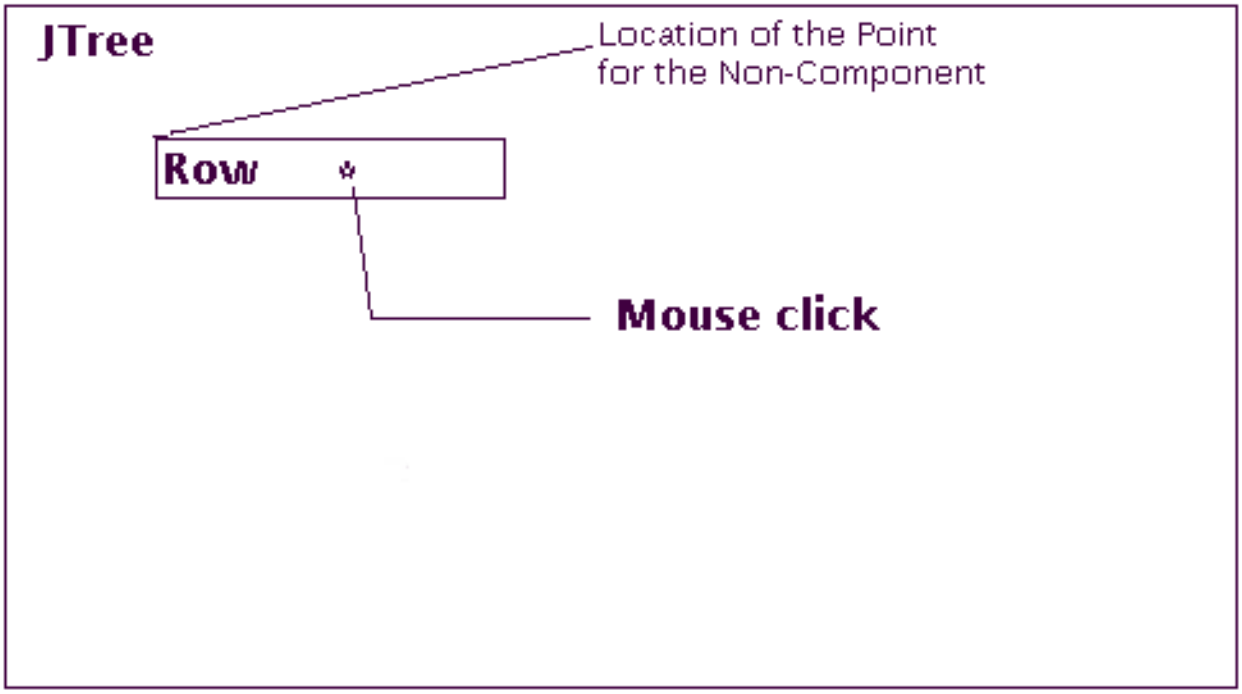


Figure 11-2 A Row contained within a Tree

The Point of the Non-Component is a fixed position from which JavaStar can consistently calculate the relative position of the multi-click. In the case of the Mozart example, the point is at the x, y coordinates of 40,72.

If JavaStar calls `findObject()` with a mouse click at 44,84, the NCL finds the row at 40,72. JavaStar sees that point in the returned `JSNCLData`, and translates the mouse click to be relative to that row. The mouse click coordinates viewed in the script, then, are 4 (44 minus 40), 12 (84 minus 72).

The row bounds of 40,72 were obtained by using the `getRowBounds()` method of `JTree`.

5. **Get the actual component.**
   The last part of the `JSNCLData` is the component itself. The NCL provides the component at the appropriate level. For a `JTree`, that was determined to be the `TreeNode` at that level. If this click was not on a tree node, no object, and no `JSNCLData`, would be returned.

The other “half” of the NCL is to be able to generate the `JSNCLData` based on the unique name, rather than a coordinate.

Retrieving a Named Non-Component

When a script is played back, JavaStar needs to find the actual component clicked upon to evaluate whether a verification continues to pass against that component. JavaStar has the unique name found at the time of recording, and the event logged against the component. It needs to verify that the appropriate non-component exists, and invoke the necessary methods against that component to evaluate a verification.

The NCL must provide a method, `getNamedObject()`, to get the JSNCLData for a component based on its name. The NCL is provided the containing component, and the unique name created at the time of recording the script.

In our example, `getNamedObject()` is passed the JTree, and the string "4:Music##Classical##Mozart".

These are the things that must occur:

1. **Ensure that the component is appropriate for this NCL.**
   In this case, the component must be a `JTree`.
2. **Parse the string to the location, and the confirmation.**
   The location is 4, in our example, and the confirmation is the string version of the row path.
3. **Confirm the location exists.**
   In this case there must be a fourth row in the `JTree`. If there isn't, the method returns null.
4. **Confirm the proper component is at that location.**
   The NCL regenerates the path string in the exact same manner as it did when recording, and ensures it is the same as the one given to the method. If it is, we have the component. If not, null is returned.
5. **Create the NCL Data for that component.**
   This is done exactly as it was for `findObject()`. See steps 3-5 of "Finding the JSNCLData while Recording", above.

Once JavaStar has the component, it can run required methods on it to verify that the tests pass.

Exercise

If you would like to try to build an NCL as an exercise, try one for a `JList`.

Before you code, you need to consider how you will name the item, locate its bounds, and obtain the component. Since we don't expect you to be a `JList` expert, here's some help.

The name of the component will be its index in the list, a colon, and a text string. We have provided a method `findName()` that returns that additional string given a `JList` and an index to it. You can use the `Parser` utility just as it was used in the `JTree` locator. To find the index, use the `locationToIndex()` method of `JList`. You can find the `JList` API in this HTML file:

```
<Javastar>/examples/SwingSet/Application/doc/api/
    com.sun.java.swing.JList.html
```

Once you have the index, you can get the bounds by using the `JList` method `indexToLocation()`.

Finding the actual object to return is a bit complicated with a `JList`, especially this example as it uses embedded images. We have provided a method `getObject()` for you to use. It returns the object given a `JList` and an index.

Given that information, and a template with the provided methods and comments, follow along with the previous example and construct a `JList` NCL.

Here is the setup information.

Setting up the Exercise

The tutorial directory for this example is:

```
<Javastar>/tutorial/WriteNCL/
```

The template for the JList Locator is:

```
<Javastar>/tutorial/WriteNCL/NCLs/template/TheListLoc.java
```

Copy this to the NCL directory:

```
<Javastar>/tutorial/WriteNCL/NCLs/
```

Write the NCL

1. **Use any editor to complete the NCL. You may use the JavaStar editor.** Follow the steps in "Anatomy of a JSNonComponentLocator", seen eariler in this lesson. Use the solution at "Solution", shown later in this lesson, as a guide.
2. **Compile the NCL.**

Test the NCL

You must have Swing 1.0.2 or higher installed and in your classpath. The product can be downloaded from the Java web site at:

```
http://java.sun.com
```

1. **Start JavaStar.**
2. **Build a project.**
   a. **Name it `WriteNCL.jpr` and click** Save As**. Save it in the `<Javastar>/tutorial/WriteNCL directory.`**
   b. **Click on the App tab. Select the Applet radio button. Browse to locate this HTML file and Click Open.**

      `<JavaStar>/examples/SwingSet/Application/SwingSetApplet.html`
   c. **Click on the Mapping tab.**
      Select the NCLs WNCL and TheListLoc from this directory:

      `<JavaStar>/tutorial/writeNCL/NCLs`

      Select the text map `TheTextMap` from this directory:

      `<JavaStar>/tutorial/writeNCL/NCLs`

      Click on the Test tab.

      Set the Tests, Test Path and JST Path directory to:

      `<JavaStar>/tutorial/writeNCL/Tests`

      Set the test results directory to:

      `<JavaStar>/tutorial/writeNCL/TestResults`
   d. **Click the** Save **button to save your work.**
3. **Click** Create a Test Script**.**
   You'll see the "Record / Playback" window. Wait a bit for the applet to load.
4. **Click** Record
5. **Name the script** `tryList`**, and click** OK**.**
6. **From the SwingSet applet, click on the tabbed pane,** ListBox**.**
   You'll see a list of food items appear.
7. **Click on the** Burgers **item in the list.**
8. **Click on the** Fries **item in the list.**
9. **Return to the "Record/Playback" window. Click** Edit**.**
   You will see the script. Check to ensure that the non-component locator was in place for these items. You should see:

   ```
   getNonComponent("TheListLoc","0:  Burger")
   ```
10. **Click** Continue **to close the editor.**
11. **Click** Stop **to compile the script.**
12. **Click** Playback **to play the script back. It should find all components.**

Solution

```
import suntest.javastar.lib.*;
import com.sun.java.swing.*;
import java.awt.*;
import java.awt.event.MouseEvent;
/*
 *
 * Non Component Locator for JList in JFC Swing
 */

public class TheListLoc implements JSNonComponentLocator {

  public JSNCLData findObject(Component c, AWTEvent e){
// ensure that this is a list and a mouse event
   if(!(c instanceof JList) || !(e instanceof MouseEvent)){
      return null;
   }
// get the point from the mouse event
      Point mousePoint = ((MouseEvent)e).getPoint();
// cast the component to a JList
      JList theList = (JList)c;
// obtain the index based on the location
    int theIndex = theList.locationToIndex(mousePoint);
// make sure the index exists
    if(theIndex != -1){
// use a common method to tranlate an index entry in a list to
NCLData
      return getListData(theList,theIndex);
    }
    return null;
  }

public JSNCLData getNamedObjectData(Component c, String wname){
// make sure this is a list
    if (!(c instanceof JList)){
   return null;
   }
// cast it to a JList
      JList theList = (JList)c;
```

```
// create a parser for this string
      Parser P = new Parser(wname);
// make sure the string is valid
      if (!P.Valid){
   throw new BadRegularExpressionError(wname + " is not a valid
format");

      }
// get the index and the confirming name from the parser

    int theIndex = P.Index;
    String theConfirmingName = P.Value;
// get the model for the list
    ListModel theListModel = theList.getModel();
// confirm that the index is within the list
    int numberInList = theListModel.getSize();

    if(theIndex<=-1){
   return null;
    }
    if(theIndex>=numberInList){
   return null;
    }
    if(theConfirmingName == null){
   return getListData(theList, theIndex);
   }
//assert that the index is within the model, and that there is a
confirming name to check for
    String name =
Parser.cleanToString(theListModel.getElementAt(theIndex));
    if(theConfirmingName.equals(findName(theList, theIndex))){
// use the common method to translate a list and an index to that
list to NCLData
   return getListData(theList,theIndex);
   } else {
      return null;
   }
  }

  public static JSNCLData getListData(JList aList, int anIndex){
   // get the model for the list
```

```
    ListModel aListModel = aList.getModel();
   // make sure the index is within the model size
    if(anIndex >=aListModel.getSize()) return null;
   // get the location of the index point
    Point theNCPoint = aList.indexToLocation(anIndex);
   // get the element for the object
   Object theNCObject = getObject(aList,anIndex);


   // get a clean string (no special characters) for this object
    String theObjectName = findName(aList, anIndex);
   // prepend the index and create the name
    String theNCName = anIndex + ":" + theObjectName;
    return new JSNCLData(theNCName, theNCPoint, theNCObject);
  }
static Object getObject(JList aList,int anIndex) {
   ListModel aListModel = aList.getModel();
   Object theNCObject = aListModel.getElementAt(anIndex);
   if (theNCObject instanceof Integer) {
      ListCellRenderer renderer = aList.getCellRenderer();
      if (renderer != null) {
        Component comp = renderer.getListCellRendererComponent
           (aList, theNCObject,
            anIndex, aList.isSelectedIndex(anIndex),
            aList.hasFocus());
        if (comp != null) {
           theNCObject = comp;
        }
      }
   }
   return theNCObject;
}
static String findName(JList L, int idx) {
    ListModel M = L.getModel();
    Object o = M.getElementAt(idx);
    String name = null;
    if (o instanceof ImageIcon) {
      name = ((ImageIcon)o).getDescription();
    } else if (o instanceof Integer) {
      ListCellRenderer renderer = L.getCellRenderer();
```

```
      if (renderer != null) {
   Component comp = renderer.getListCellRendererComponent(L, o,
idx, L.isSelectedIndex(idx), L.hasFocus());
   if (comp != null) {
     o = comp;
     if (comp.getName() != null && comp.getName().length() > 0) {
       name = comp.getName();
     } else {
       if (comp instanceof JLabel)
         name = ((JLabel)comp).getText();
       else if (comp instanceof Label)
         name = ((Label)comp).getText();
     }
   }
      }
    }
    return (name != null ? name : Parser.cleanToString(o));
  }
}
```

Summary

Building a non-component locator requires in-depth knowledge of the component you are trying to test and its contained components. Some Java skills are required as well. However, they are not necessarily difficult once you have determined the means of naming, determining bounds, and locating the component.

Index

P

R

S

T

V