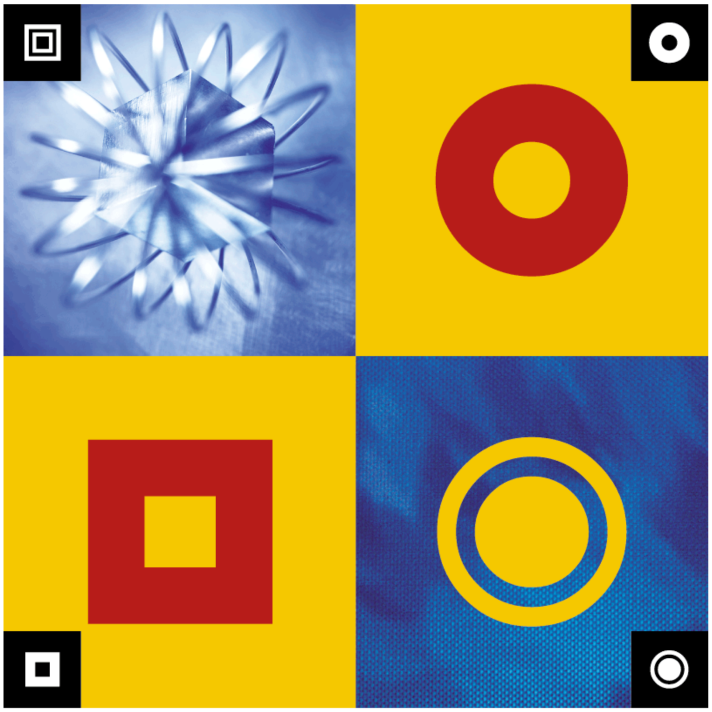
MERCURY INTERACTIVE

WinRunner

JAVA™ ADD-IN



USER'S GUIDE

WinRunner Java™ Add-in

User's Guide

Version 7.6

MERCURY INTERACTIVE

WinRunner Java Add-in User’s Guide, Version 7.6

Mercury Interactive Corporation
1325 Borregas Avenue
Sunnyvale, CA 94089 USA
Tel: (408) 822-5200
Toll Free: (800) TEST-911, (866) TOPAZ-4U
Fax: (408) 822-5300

If you have any comments or suggestions regarding this document, please send them via e-mail to documentation@merc-int.com.

WRJAVAUG7.6/01

Table of Contents

1

Introduction

Welcome to the WinRunner Java Add-in. This guide explains how to use WinRunner to successfully test Java applications and applets. It should be used in conjunction with the *WinRunner User's Guide* and the *TSL Online Reference.*

This chapter describes:

- ➤ Using the Java Add-in
- ➤ How the Java Add-in Identifies Java Objects
- ➤ Activating the Java Add-in

Using the Java Add-in

The Java Add-in is an add-in to WinRunner, Mercury Interactive's automated GUI testing tool for Microsoft Windows applications.

The Java Add-in enables you to record and run tests on cross-platform Java applets and applications. You can record and run user actions on Java objects in Internet Explorer or Netscape, in Java Web Start, in Sun's AppletViewer, and in standalone Java applications.

To create a test for a Java application or applet, use WinRunner to record the operations you perform on the applet or application. As you work with Java objects, WinRunner generates a test script in TSL, Mercury Interactive's C-like test script language.

With the Java Add-in you can:

- Record operations on standard Java objects just as you would any other Windows object with WinRunner.
- Configure the GUI map to recognize custom Java objects as push buttons, check buttons, static text, or text fields.
- Use various TSL functions to execute Java methods from the WinRunner script.
- Use the **java_fire_event** function to simulate a Java event on the specified object.

How the Java Add-in Identifies Java Objects

WinRunner learns a set of default properties for each object you operate on while recording a test. These properties enable WinRunner to obtain a unique identification for every object that you test. This information is stored in the GUI map. WinRunner uses the GUI map to help it locate frames and objects during a test run.

WinRunner identifies standard Java objects as push button, check button, static text, list, table, or text field classes, and stores the relevant physical properties in the GUI Map just like the corresponding classes of Windows objects. If you record an action on a custom or unsupported Java object, WinRunner maps the object to the general object class in the WinRunner GUI map unless you configure the GUI map to identify the object as a custom Java object, by choosing **Tools** > **Java Custom Object Wizard**. A custom Java object can be configured as a push button, check button, static text, text field, and so forth, and you can configure the physical properties that will be used to identify the object. For more information on GUI maps, refer to the "Configuring the GUI Map" chapter in the *WinRunner User's Guide*.

You can view the contents of your GUI map files in the GUI Map Editor by choosing **Tools** > **GUI Map Editor**. The GUI Map Editor displays the logical names and the physical descriptions of objects. For more information on GUI maps, refer to the "Understanding the GUI Map" section in the *WinRunner User's Guide*.

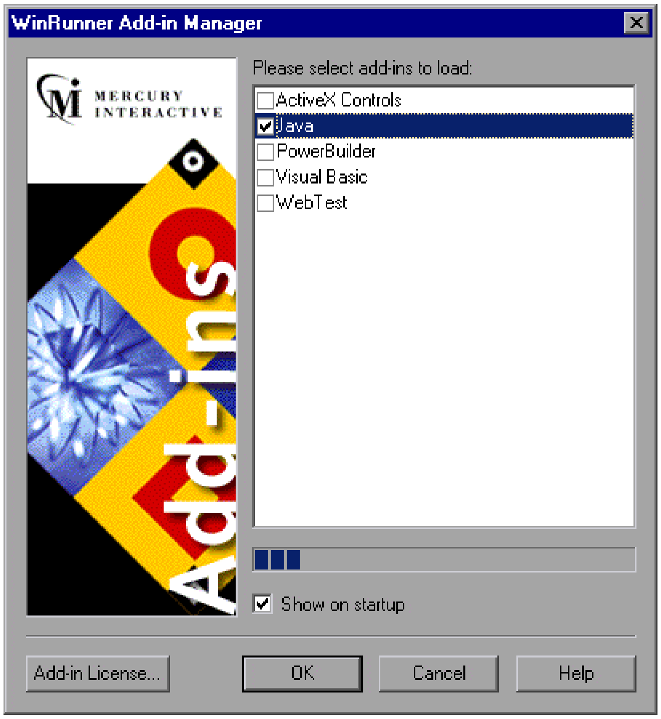
Activating the Java Add-in

Before you begin testing your Java application or applet, make sure that you have installed all the necessary files and made any necessary configuration changes. For more information, refer to the *WinRunner Java Add-in Installation Guide*.

Note: The RapidTest Script Wizard option is not supported by the Java Add-in. For more information about the RapidTest Script Wizard, refer to the *WinRunner User's Guide*.

To activate the Java Add-in:

1 Select **Start** > **Programs** > **WinRunner** > **WinRunner**. The WinRunner Add-in Manager dialog box opens.



Please select add-ins to load:

- [ ] ActiveX Controls
- [✓] Java
- [ ] PowerBuilder
- [ ] Visual Basic
- [ ] WebTest

[✓] Show on startup

2 Select **Java**.

3 Click **OK**. WinRunner opens with the Java Add-in loaded.

Note:

If the Add-In Manager dialog box does not open:

1 Start WinRunner.

2 In **Tools** > **General Options** > **General** category > **Startup** sub-category, check **Display Add-in Manager on startup**. In the **Hide Add-in Manager after ___ seconds** box, enter the number of seconds for which the Add-in Manager is displayed. (The default value is **10** seconds.)

3 Click **OK**.

4 Close WinRunner. A WinRunner message box opens asking whether you want to keep the changes you made. Click **Yes**.

For more information on the Add-in Manager, refer to the *WinRunner User's Guide*.

2

Testing Standard Java Objects

This chapter describes how to record standard Java objects and enhance scripts that test Java applications and applets.

This chapter describes:

About Testing Standard Java Objects

With the Java Add-in, you can record or write context sensitive scripts on all standard Java objects from the supported toolkits. You can also use TSL functions that enable you to add Java-specific statements to your script.

Recording Context Sensitive Tests

Whenever you start WinRunner with the Java Add-in loaded, support for the Java environments you installed will always be loaded. For more information about selecting Java environments, refer to the *WinRunner Java Add-in Installation Guide*.

You can confirm that your Java environment has opened properly by checking the Java console for the following confirmation message: "Loading Mercury Support (version x.xxx)".

Note: If the browser's Java console and a Java plug-in are open simultaneously, the Java Add-in will not function properly, since this scenario results in two virtual machines and WinRunner cannot distinguish between them. Typically, the last virtual machine created within a process is supported. If this happens, close the browser and console, and then reopen the browser before running the tests.

If your Java application or applet uses standard Java objects from any of the supported toolkits, then you can use WinRunner to record a Context Sensitive test, just as you would with any Windows application.

As you record, WinRunner adds standard Context Sensitive TSL statements into the script. If you try to record an action on an unsupported or custom Java object, WinRunner records a generic **obj_mouse_click** or **win_mouse_click** statement. You can configure WinRunner to recognize your custom objects as push buttons, check buttons, static text, edit fields, and so forth, by using the Java Custom Object Wizard. For more information, refer to Chapter 4, "Configuring Custom Java Objects."

Enhancing Your Script with TSL

WinRunner includes several TSL functions that enable you to add Java-specific statements to your script. Specifically, you can use TSL functions to:

- Set the value of a Java bean-like property.
- Activate a specified Java edit field.
- Find the dimensions and coordinates of list and tree items in JFC (Swing toolkit).
- Select an item from a Java pop-up menu.
- Configure the way WinRunner learns object descriptions and runs tests on Java applications and applets.
- Configure the way WinRunner records Swing/JFC table cell editors.

You can also use TSL functions to invoke the methods of Java objects and to simulate events on Java objects. These are covered in Chapter 3, “Working with Java Methods and Events.”

For more information about TSL functions and how to use TSL, refer to the *TSL Reference Guide* or the *TSL Online Reference*.

Setting the Value of a Java Bean-Like Property

You can set the value of a Java bean-like property with the **obj_set_info** function. This function works on all properties that have a set method.

Tip: You can also use the **java_set_field** function to set the value of a Java field, even if it does not have a set method. The **java_set_field** function has the following syntax: **java_set_field (** *object*, *field_name*, *value* **);**. For more information, see “Using the java_set_field Function” on page 25.

The **obj_set_info** function has the following syntax:

obj_set_info (*object*, *property*, *value* **);**

The *object* parameter is the logical name of the object. The object may belong to any class. The *property* parameter is the object property you want to set and can be any of the properties displayed when using the WinRunner GUI Spy. Refer to the *WinRunner Users Guide* for more information on the GUI Spy or for a list of properties. The *value* parameter is the value that is assigned to the property.

Note: When writing the *property* parameter name in the function, convert the capital letters of the *property* to lowercase, and add an underscore before letters that are capitalized within the Java bean-like property name. Therefore, a Java bean-like property called **MyProp** becomes **my_prop** in the TSL statement.

For example, for a property called **MyProp**, which has the method **setMyProp(String)**, you can use the function as follows:

```
obj_set_info(object, "my_prop", "Mercury");
```

The **obj_set_info** function will return ATTRIBUTE_NOT_SUPPORTED for the property **my_prop** if one of the following statements is true:

- The object does not have a method called *setMyProp*.
- The method **setMyProp()** exists, but it has more than one parameter, or the parameter is not of one of the following types: String, int (or Integer), boolean (or Boolean), or float (or Float).
- The value parameter is not convertible to one of the above Java classes. For example, the method gets an integer number as a parameter, but the function's value parameter was a non-numeric value.
- The **setMyprop()** method creates a Java exception.

Activating a Java Edit Object

You can activate an edit field with the **edit_activate** function. This is the equivalent of a user pressing the ENTER key on an edit field. This function has the following syntax:

edit_activate (*object* **);**

The *object* parameter is the logical name of the edit object on which you want to perform the action.

For example, if you want to enter John Smith into the edit field Text_Fields_0, then you can set the text in the edit field and then use **edit_activate** to send the activate event, as in the following script:

```
set_window("swingsetapplet.html", 8);
edit_set("Text Fields:_0", "John Smith 2");
edit_activate("Text Fields:_0");
```

Finding the Location of a List Item

You can find the dimensions and coordinates of list and tree items in JFC (swing toolkit) with the **list_get_item_coord** function. This function has the following syntax:

list_get_item_coord (*list*, *item*, *out_x*, *out_y*, *out_width*, *out_height* **);**

The *list* parameter is the name of the list. The *item* parameter is the item string. The *out_x* and *out_y* parameters are the output variables that store the x- and y- coordinates of the item rectangle. The *out_width* and *out_height* parameters are the output variables that store the width and height of the item rectangle.

For example, for a list called "ListPanel$1" containing an item called "Cola", you can use the function as follows to find the location of the Cola item:

```
set_window("swingsetapplet.html");
tab_select_item("JTabbedPane", "ListBox");
list_select_item("ListPanel$1", " Cola");
rc = list_get_item_coord("ListPanel$1", " Cola", x_list_src, y_list_src,
    width_list_src, height_list_src);
```

Selecting an Item from a Java Pop-up Menu

You can select an item from a Java pop-up menu using the **popup_select_item** function. This function has the following syntax:

popup_select_item ("*menu;item*" **);**

The *menu;item* parameter indicates the logical name of the component containing the menu and the name of the item.

Note that *menu* and *item* are represented as a single string, and are separated by a semicolon.

When an item is selected from a submenu, each consecutive level of the menu is separated by a semicolon in the format "menu; sub_menu1; sub_menu2;...sub_menun; item." The item must be specified in a chain composed of menu objects from the GUI map and ending in the name of the menu item as it appears in the application. For example, the function **popup_select_item ("Copy");** does not use the correct syntax; while **popup_select_item ("MyEdit;Copy");** is correct.

The **popup_select_item** statement does not open the pop-up menu; you can open the menu by a preceding TSL statement. For example:

```
obj_mouse_click ("MyEdit", 1, 1, RIGHT);
```

Note: When using the **popup_select_item** function on AWT toolkit pop-up menus, the action that opens the menu must be performed during the test run using the USE_LOW_LEVEL_EVENTS variable. For more information, see page 14.

Configuring Java Variable Settings

You can configure how WinRunner learns descriptions of objects, records and runs tests on Java applications or applets, or otherwise affect record or run-related settings, with the **set_aut_var** function. This function has the following syntax:

set_aut_var (*variable*, *value* **);**

Note: Variable names are not case sensitive.
Variable values may or may not be case sensitive, as specified below.

The following variables and corresponding values are available:

EDIT_REPLAY_MODE	Controls how WinRunner performs actions on edit fields. Use one or more of the following values: "S"—uses the setText () or setValue () methods to set a value of the edit object. "P"—sends KeyPressed event to the object for every character from the input string. "T"—sends KeyTyped events to the object for every character from the input string. "R"—sends KeyReleased event to the object for every character from the input string. "F"—generates a FocusLost event at the end of function execution. **Note:** EDIT_REPLAY_MODE variable values are case sensitive. **Default value:** "**PTR**" Note that the default value sends a triple event to the edit field (KeyPressed-KeyTyped-KeyReleased), just as an actual user would generate a key stroke.

EXCLUDE_CONTROL_CHARS	Specifies the characters to be ignored from the setText () call by the edit_set command when REPLAY_MODE_EDIT contains "S". For example: `set_aut_var ("EXCLUDE_CONTROL_CHARS", "\t");` means that the tab character will not be included in the setText () method call when EDIT_REPLAY_MODE contains "S".
MAX_TEXT_DISTANCE	Sets the maximum distance in pixels, to look for attached text. **Default value: 100**
RECORD_BY_NUM	Controls how items in list, combo box, table, tab control, and tree view objects are recorded. The variable can be one of the following values: "list", "combo", "table", "tab", "tree", or a combination of these values separated by a space. If one of these objects is detected, numbers are recorded instead of the item names or row/column header names. ("table" is supported for KLG or JCTable objects. "tab" is supported for JFC, Vcafe, and KLG 3.x.) To return to recording these items by name, set the variable value as an empty string. **Note:** RECORD_BY_NUM variable values are case sensitive.
RECORD_WIN_OPS	Determines whether window operations (move and resize) are recorded. Use one of the following values: "ON" (or any non-zero numeric value) "OFF" **Default value: "OFF"**

SKIP_ON_LEARN	Controls how WinRunner learns a window. Mercury Interactive classes listed in the variable are ignored when WinRunner learns objects in a window from the GUI Map Editor. May contain a list of Mercury Interactive classes, separated by spaces. By default, only objects with the WinRunner class "object" are skipped. **Note:** SKIP_ON_LEARN variable values are case sensitive. **Default value: "object"**
TABLE_EXTERNAL_EDITORS_LIST	Specifies a list of editor class names that should never be treated as part of a JTable object but rather as separate objects. The specified editors should be ones that by default are treated as part of a JTable object (using table functions) but either do not work correctly with the table functions, or work correctly but special actions cannot be performed on them. This variable is available only for JTable Swing toolkit tables. For more information, see “Recording on Swing/JFC Table Objects” on page 15. Use one or more of the following values: Editor class names, separated by a space, tab, newline, or return character. **Note:** TABLE_EXTERNAL_EDITORS variable values are case sensitive.
TABLE_RECORD_MODE	Sets the record mode for a table object (CS or ANALOG). Use one or more of the following values: "CS"—indicates that the record mode is Context Sensitive.

	"ANALOG"—records only low-level (analog) table functions: **tbl_click_cell**, **tbl_dbl_click_cell**, and **tbl_drag**. (JFC JTable objects, KLG 3.6 table objects, and KLG 4.x/5.0 JCTable objects only.)
	Default value: "**CS**"
TREEVIEW_PATH_SEPARATOR	Specifies the default separator "**;**" used to separate entries in a path to a node of a TreeView control.
	Note: If you specify more than one character, for example "**#$**", then WinRunner treats either of the characters as a separator (but not both of them in sequence).
	Default value: ";"
USE_LOW_LEVEL_EVENTS	Controls whether WinRunner simulates user input by Java events or by the mouse and keyboard drivers. When a test runs using this mode, the cursor moves on the screen, as if performing the recorded user operations.
	Use one or more of the following values:
	"all"—indicates that WinRunner simulates all mouse clicks and keyboard strokes for all types of Java objects by the mouse and keyboard drivers.
	WinRunner class names separated by a space indicate that WinRunner uses mouse and keyboard drivers to simulate user input on object of the class names listed. For example, "push_button edit" uses mouse and keyboard drivers to simulate user input on all buttons and edit boxes. To return to simulating user input by Java events, set the variable value as an empty string.

The low level events mode should be used only when the Java Add-in fails to correctly perform an action on your application. In this mode, the test run resembles user behavior, and therefore may succeed where the regular mode fails. It is recommended to only use this mode in specific statements, and not for the entire test.

When running a test on AWT pop-up menus, it is required to use low level events mode in most cases. Note that this mode is less context sensitive, therefore it is recommended to use it only when necessary.

Note: USE_LOW_LEVEL_EVENTS variable values are case sensitive.

Recording on Swing/JFC Table Objects

When you record an operation that changes the data in a cell of a Java table, WinRunner generally records the end result of the data in the cell in the form of a **tbl_set_cell_data** function.

Note: tbl_set_cell_data is not used when the **TABLE_RECORD_MODE** variable is set to **ANALOG**. For more information on the **TABLE_RECORD_MODE** variable, see “Configuring Java Variable Settings” on page 11.

Recording on Standard Cell Editors in Swing JTable Tables

The Java Add-in provides built-in support for several standard Swing JTable cell editor types. This means that by default, WinRunner records operations on these standard cell editors using **tbl_set_cell_data** functions.

Recording on Custom Cell Editors in Swing JTable Tables

When a JTable contains a custom (non-standard) cell editor, the default **tbl_set_cell_data** function cannot be recorded. For example, if a cell contains both a check box and a button that opens a dialog box, then a **tbl_set_cell_data** function may not always provide an accurate description of the operation(s) performed inside the cell.

If you record an operation on a custom cell editor, WinRunner records a function that reflects the operation you performed on the object inside the cell. For example, if the cell editor contains a custom edit box, WinRunner records a statement like the following, depending on the operation that was needed to activate the cell while the test was being recorded:

```
set_window("SwingSetApplet", 8);
tbl_set_selected_cell ("Inter-cell spacing:_1", "#0", "Last Name");
edit_set("Type Here:", "Andrews");
```

or

```
set_window("SwingSetApplet", 8);
tbl_activate_cell ("Inter-cell spacing:_1", "#0", "Last Name");
edit_set("Type Here:", "Andrews");
```

instead of:

```
set_window("SwingSetApplet", 1);
tbl_set_cell_data("Inter-cell spacing:_1", "#0", "Last Name", "Andrews");
```

Modifying the Default JTable Recording Behavior (Advanced)

In most cases, the default recording behavior for JTables works well and maximizes the readability of your test. However, if you are not satisfied with the value that WinRunner records for the **tbl_set_cell_data** function of a particular editor, or if the test does not run correctly, you can set that editor to be recorded, like a custom cell editor, in terms of the operation performed on the object inside the cell.

To do this, use the **TABLE_EXTERNAL_EDITORS_LIST** variable with the **set_aut_var** function to specify specific cell editor type(s) that should always be treated as separate objects, and not as part of a table object. You specify the editors as a space-separated list of the relevant toolkit classes.

For more information on the **set_aut_var** function and its variables, see "Configuring Java Variable Settings" on page 11.

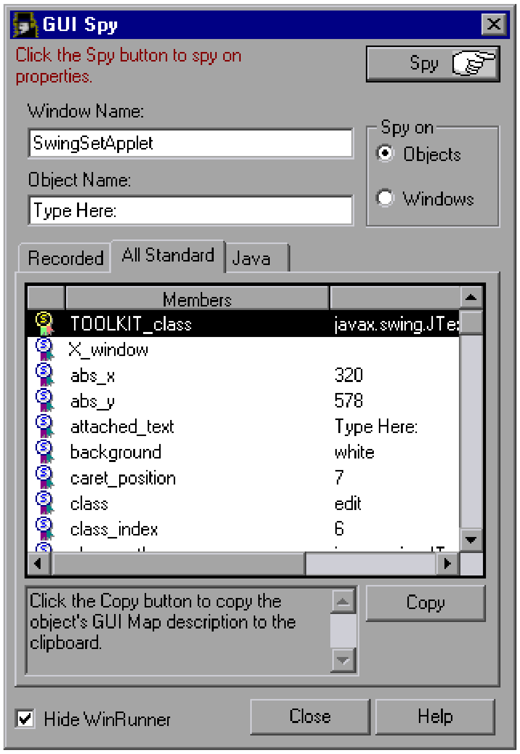
Finding the Toolkit Class of a JTable Editor

If you do not know the value of the toolkit class for an editor for use with the **TABLE_EXTERNAL_EDITORS_LIST** variable, you can find it either by using the GUI Spy or by running a short script in WinRunner to retrieve the value.

To find the toolkit class of a JTable cell editor using the GUI Spy:

1. Open the table and activate the cell editor. For example, make sure the cursor is blinking inside an edit field, or display the drop-down list of a combo box.
2. With the appropriate cell activated, use the GUI Spy to point to the active cell. For more information on using the GUI Spy, refer to the *WinRunner User's Guide*.

3 Display the **All Standard** tab of the GUI Spy.



Window Name: SwingSetApplet

Object Name: Type Here:

	Members	
	TOOLKIT_class	javax.swing.JTe
	X_window	
	abs_x	320
	abs_y	578
	attached_text	Type Here:
	background	white
	caret_position	7
	class	edit
	class_index	6

Click the Copy button to copy the object's GUI Map description to the clipboard.

4 In the **Members** list, find **TOOLKIT_class**. The toolkit class value is displayed next to the **TOOLKIT_class** item.

5 Enter the toolkit class value (the case-sensitive fully qualified Java class name) into your TABLE_EXTERNAL_EDITORS_LIST variable.

Finding the Toolkit Class of a JTable Editor by Running a WinRunner Script

For some cell editors, it is difficult or impossible to capture an activated cell with the GUI Spy because the cell does not stay activated for a long enough period of time. For example, with a check box, once the check box has been selected or cleared, the cell editor is no longer active.

If you need to find the toolkit class value to use for these types of cell editors, you can run a a short script similar to the following script in WinRunner to retrieve the value.

```
set_window("Table Demo", 1);
java_activate_method("Inter-cell spacing:_1", "editCellAt", editable, 4, 2);
    # row 4, column 2
if (editable != "false") {
    java_activate_method("Inter-cell spacing:_1","getEditorComponent",
        component);
    java_activate_method(component, "getClass", class);
    java_activate_method(class, "getName", name);
    pause(name);
} else {
    pause("Cell is not editable");
}
```

3

Working with Java Methods and Events

This chapter describes how to invoke the methods of Java objects. It also describes how to simulate events on Java objects.

This chapter describes:

About Working with Java Methods and Events

You can invoke object methods during your test using the **java_activate_method** function or static (class) methods using the **java_activate_static** function**.** You can view the methods of Java objects in your application using the GUI spy or the Java Method Wizard. You can also generate the appropriate TSL statement for activating the method you select.

You can access object fields using any of the following functions: **java_get_field**, **java_set_field**, **java_get_static**, or **java_set_static**.

You can also simulate events on Java objects using the **fire_java_event** function.

Invoking Java Methods

You can invoke a Java method for any Java object using the **java_activate_method** function. You can invoke a static method using the **java_activate_static** function.

Using the java_activate_method Function

You can use the **java_activate_method** function to invoke object methods during your test.

The **java_activate_method** function has the following syntax:

java_activate_method (*object*, *method_name*, *retval* **[,** *param1*, ... *param8* **]);**

The *object* parameter is the logical name of the object (for a visible, GUI object) or an object returned from a previous **java_activate_method** function or any other function described in this chapter. For more information on return values, see "Working with Return Values (Advanced)" on page 26. The *method_name* parameter indicates the name of the Java method to invoke. The *retval* parameter is an output variable that holds a return value from the invoked method. Note that this parameter is required even for void Java methods. *param1...8* are optional parameters to be passed to the Java method.

The Java method parameters may belong to one of the following Java data types: boolean, int, long, float, double, or string, or they may be any other Java object returned from a previous **java_activate_method** function or any other function described in this chapter. For more information about using returned objects in your script, see "Working with Return Values (Advanced)" on page 26.

Note: If the function returns boolean output, the *retval* parameter returns the string representation of the output: "true" or "false".

For example, you can use the **java_activate_method** function to perform actions on a list:

```
# Add item to the list at position 2:
java_activate_method("list", "add", retval, "new item", 2);

# Get number of visible rows in a list:
java_activate_method("list", "getRows", rows);

# Check if an item is selected:
java_activate_method("list", "isIndexSelected", isSelected, 2);
```

The TSL return value for the **java_activate_method** function can be any of the TSL general return values. For more information on TSL return values, refer to the *TSL Reference Guide*.

Using the java_activate_static Function

You can invoke a static method of any Java class using the **java_activate_static** function.

The **java_activate_static** function has the following syntax:

java_activate_static (*class_name, method_name, retval* **[,** *param1, ... param8* **]);**

The *class_name* parameter is the fully-qualified Java class name. The *method_name* parameter indicates the name of the static Java method to invoke. The *retval* parameter is an output variable that holds a return value from the invoked method. *param1...8* are optional parameters to be passed to the Java method.

The Java method parameters may belong to one of the following Java data types: boolean, int, long, float, double, or string, or they may be any other Java object returned from a previous **java_activate_static** function or any other function described in this chapter. For more information about using returned objects in your script, see “Working with Return Values (Advanced)” on page 26.

Note: If the function returns boolean output, the *retval* parameter will return the string representation of the output: "true" or "false".

For example, you can use the **java_activate_static** function to invoke the toHexString static method of the Java class Integer.

```
java_activate_static("java.lang.Integer", "toHexString", hex_str, 127);
```

Accessing Object Fields

You can access object fields using the **java_get_field** or **java_set field** functions. You can use the **java_get_static** or **java_set_static** functions to access static fields.

Using the java_get_field Function

You can use the **java_get_field** function to retrieve the current value of an object's field.

The **java_get_field** function has the following syntax:

java_get_field (*object*, *field_name*, *out_value* **);**

The *object* parameter is the logical name of the object whose field is retrieved, or an object returned from a previous **java_get_field** function or any other function described in this chapter. The *field_name* parameter indicates the name of the field to retrieve. The *out_value* parameter is an output variable that holds the value from the retrieved field.

For example, you can use the **java_get_field** function to retrieve the value of the "x" field of a Java point object:

```
java_get_field(point_object, "x", ret_val);
```

Using the java_set_field Function

You can use the **java_set_field** function to set the specified value of an object's field.

The **java_set_field** function has the following syntax:

java_set_field (*object*, *field_name*, *value* **);**

The *object* parameter is the logical name of the object or the value returned from a previous **java_set_field** function or any other function described in this chapter. The *field_name* parameter indicates the name of the field whose value will be set. The *value* parameter holds the new value of the field.

The *value* parameter may belong to one of the following Java data types: boolean, int, long, float, double, or String, or it may be any other value returned from a previous **java_set_field** function or any other function described in this chapter. For more information about using returned objects in your script, see "Working with Return Values (Advanced)" on page 26.

For example, you can use the **java_set_field** function to set the value of the "x" field to 5:

```
java_set_field(point_object, "x", 5);
```

Using the java_get_static Function

You can use the **java_get_static** function to retrieve the current value of a static field.

The **java_get_static** function has the following syntax:

java_get_static (*class*, *field_name*, *out_value* **);**

The *class* parameter is the fully-qualified Java class name. The *field_name* parameter indicates the name of the field to retrieve. The *out_value* parameter is an output variable that holds a return value from the retrieved field.

For example, you can use the **java_get_static** function to retrieve the value of the "out" static field of the "java.lang.System" class:

```
java_get_static("java.lang.System", "out", ret_val);
```

Using the java_set_static Function

You can use the **java_set_static** function to set the specified value of a static field.

The **java_set_static** function has the following syntax:

java_set_static (*class*, *field_name*, *value* **);**

The *class* parameter is the fully-qualified Java class name. The *field_name* parameter indicates the name of the field whose value will be set. The *value* parameter holds the new value of the field.

The *value* parameter may belong to one of the following Java data types: boolean, int, long, float, double, or String, or it may be any other value returned from a previous **java_set_static** function or any other function described in this chapter. For more information about using returned objects in your script, see “Working with Return Values (Advanced)” on page 26.

For example, you can use the **java_set_static** function to set the value of the "out" static field of the "java.lang.System" class:

```
java_set_static("java.lang.System", "out", 12);
```

Working with Return Values (Advanced)

If a Java object is returned from a prior **java_activate_method** statement, you can use the returned object to invoke its methods. You can also use the returned object as an argument to another **java_activate_method** function or any of the other functions described in this chapter.

You can also use the **jco_create** function to create a new Java object within your application or applet.

The **jco_create** function has the following syntax:

jco_create (*existing_obj* **,** *new_obj* **,** *class_name* **,** *[param1* **,** *...* **,** *param8]* **);**

The *existing_obj* parameter specifies the object whose class loader will be used to find the class of the newly created object. This can be the main application or applet window, or any other Java object within the application or applet. The *new_obj* output parameter is the new object to be returned. The *class_name* parameter is the fully-qualified Java class name. *Param1...Param8* are the required parameters for that object constructor. These parameters can be of type: int, float, boolean ("true" or "false"), String, or any value returned from a previous **jco_create** function or any of the other functions described in this chapter.

You invoke the methods of a returned object just as you would any other Java object, using the **java_activate_method** syntax described above.

Note: You can use the "_jco_null" object as a parameter in order to represent a null object.

When a Java object is returned from a **java_activate_method** or **jco_create** statement, a reference to the object is held by the Java Add-in. When you have finished using the returned object in your script, you should use the **jco_free** function to release the reference to the specific object. You can also use the **jco_free_all** function to release all object references held by the Java Add-in.

These two functions have the following syntax:

jco_free (*object* **);**
jco_free_all();

Note: A returned object can only be used to invoke the methods of that object or as an argument for another **java_activate_method** or any of the other functions described in this chapter. Do not use a returned object as an argument for other functions.

Viewing Object Methods in Your Application or Applet

If you are not sure which methods are available for a given object, you can use the GUI Spy or the Java Method Wizard to view all of the methods associated with the object. You can also use the GUI Spy or the Java Method Wizard to generate the appropriate **java_activate_method** function for a selected method.

Using the GUI Spy

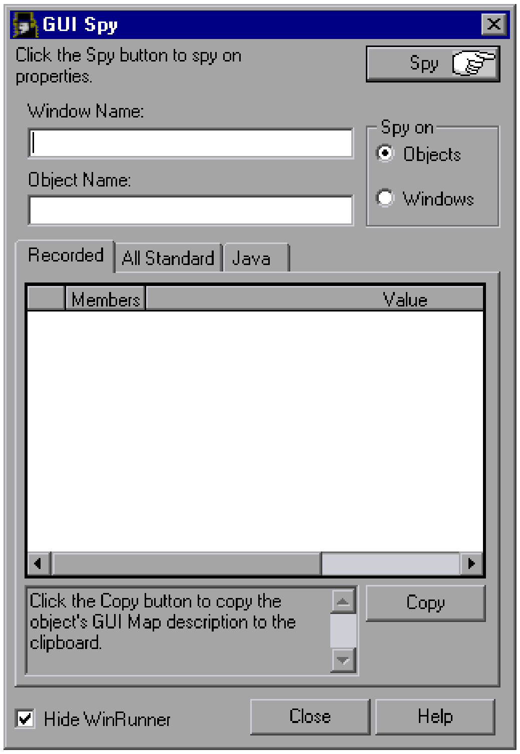
You can view all methods associated with GUI Java objects in your application or applet and generate the appropriate **java_activate_method** function for a selected method using the Java tab of the GUI Spy.

Note: As with any other GUI object, you can view all properties or just the recorded properties of a Java object in the All Standard or Recorded tabs of the GUI Spy. For more information on these elements of the GUI Spy, refer to the *WinRunner User's Guide*.

To view object methods in your application or applet using the GUI Spy:

1 Open the Java application or applet that contains the object for which you want to view the methods.

2 Choose **Tools** > **GUI Spy**. The GUI Spy opens.



Click the Spy button to spy on properties.

Window Name:

Object Name:

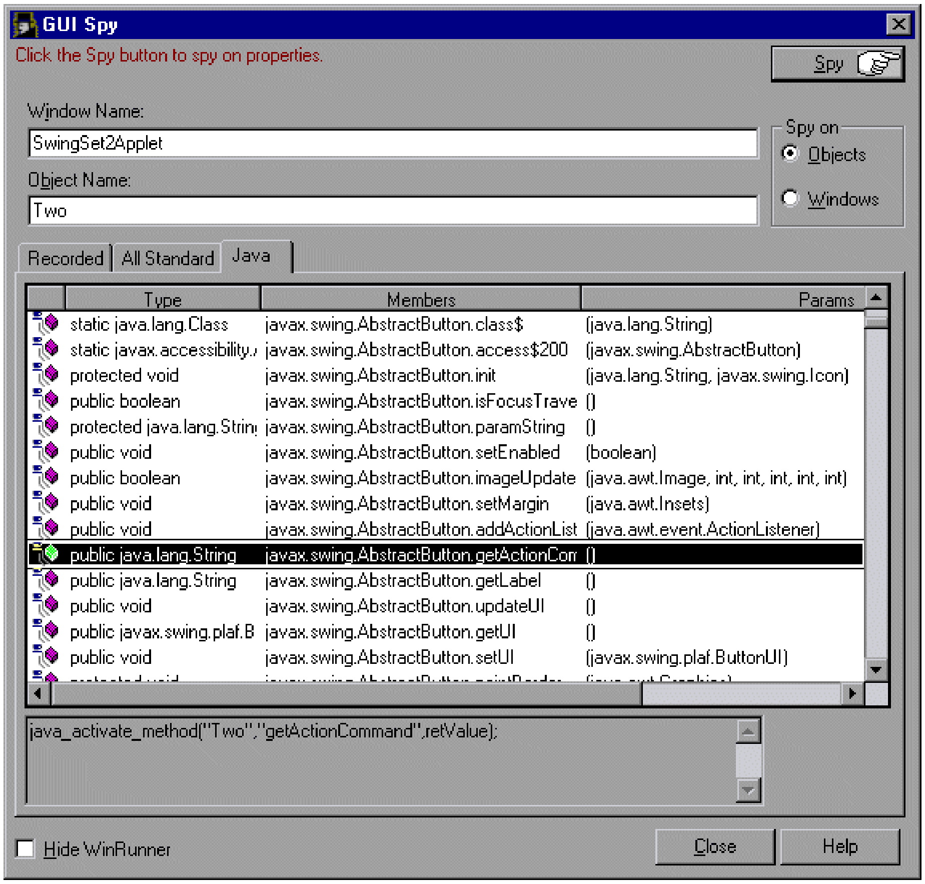
[✓ Objects] Windows

Click the Copy button to copy the object's GUI Map description to the clipboard.

[✓ Hide WinRunner]

3 Click the **Java** tab.

4 Click **Spy** and point to an object on the screen. The object is highlighted and the active window name, object name, and all of the object's Java methods appear in the appropriate fields. The object's methods are listed first, followed by a listing of methods inherited from the object's superclasses.



Click the Spy button to spy on properties.

Window Name: SwingSet2Applet

Object Name: Two

Spy on: [✓ Objects] Windows

Type	Members	Params
static java.lang.Class	javax.swing.AbstractButton.class$	(java.lang.String)
static javax.accessibility.,	javax.swing.AbstractButton.access$200	(javax.swing.AbstractButton)
protected void	javax.swing.AbstractButton.init	(java.lang.String, javax.swing.Icon)
public boolean	javax.swing.AbstractButton.isFocusTrave	()
protected java.lang.Strin	javax.swing.AbstractButton.paramString	()
public void	javax.swing.AbstractButton.setEnabled	(boolean)
public boolean	javax.swing.AbstractButton.imageUpdate	(java.awt.Image, int, int, int, int, int)
public void	javax.swing.AbstractButton.setMargin	(java.awt.Insets)
public void	javax.swing.AbstractButton.addActionList	(java.awt.event.ActionListener)
public java.lang.String	javax.swing.AbstractButton.getActionCon	()
public java.lang.String	javax.swing.AbstractButton.getLabel	()
public void	javax.swing.AbstractButton.updateUI	()
public javax.swing.plaf.B	javax.swing.AbstractButton.getUI	()
public void	javax.swing.AbstractButton.setUI	(javax.swing.plaf.ButtonUI)

```
java_activate_method("Two","getActionCommand",retValue);
```

Hide WinRunner

5 To capture the object methods in the GUI Spy dialog box, point to the desired object and press the STOP softkey. (The default softkey combination is Ctrl Left + F3.)

To generate the TSL statement for invoking a Java method:

1. Activate the GUI Spy as described on page 28.
2. Select the method that you want to invoke from the list of methods. The appropriate **java_activate_method** is displayed in the TSL statement box.

Note: If you run a Java application on a virtual machine earlier than JDK version 1.2, the **java_activate_method** function cannot invoke Protected, Default (i.e., package), or Private method types.

---

3. Copy the statement displayed in the box and paste it into your script.
4. Input parameters are identified as Param1, Param2, and so forth. Replace the input parameters in the statement with the parameter values you want to send to the method.

   The Java method parameters may belong to one of the following Java data types: boolean, int, long, float, double, or string, or they may be any other Java object returned from a previous **java_activate_method** function or any other function described in this chapter. For more information, see "Using the java_activate_method Function" on page 22.

   For example, if you want to change the text on the button labeled "One" to "Yes", highlight the **setText** method and copy the statement in the box:

   ```
   rc = java_activate_method("One","setText",retValue,param1);
   ```

   and replace Param1 with "Yes" as shown below:

   ```
   rc = java_activate_method("One","setText",retValue,"Yes");
   ```

Using the Java Method Wizard

You can use the Java Method Wizard to view the methods associated with Java objects and to generate the appropriate **java_activate_method** statement for one of the displayed methods.

To view the methods for an object in your application or applet:

1 Open the Java application or applet that contains the object for which you want to view the methods.

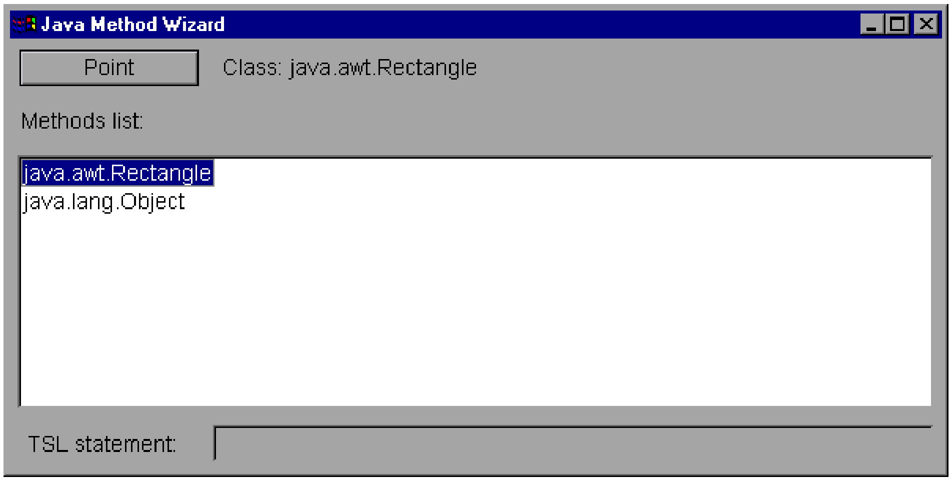
2 Enter a **method_wizard** statement to activate the Java Method Wizard using the syntax:

method_wizard (*object* **);**

where *object* is the logical name of the object for which you want to view the methods, or an object returned from a previous **java_activate_method** function, or any of the other functions described in this chapter.

3 Select **Debug** run mode in the toolbar.

4 Choose **Debug** > **Step**, or click the **Step** button to run the statement. The Java Method Wizard opens and displays a list with the object's class and all of its superclasses.



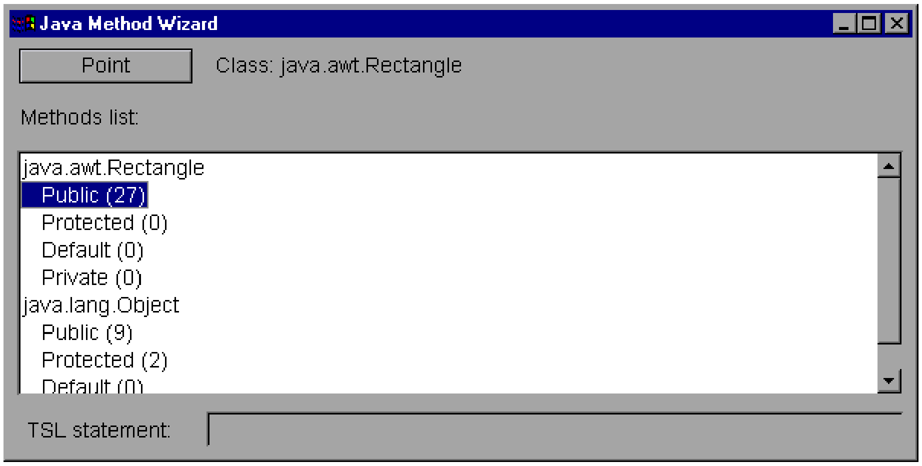
Class: java.awt.Rectangle

Methods list:

java.awt.Rectangle
java.lang.Object

Note: After the Java Method Wizard opens, the focus returns to the main WinRunner window. You may need to select the Java Method Wizard icon on your Windows taskbar to display the wizard.

5 Double-click a class element to view a summary of available methods by type.


Java Method Wizard

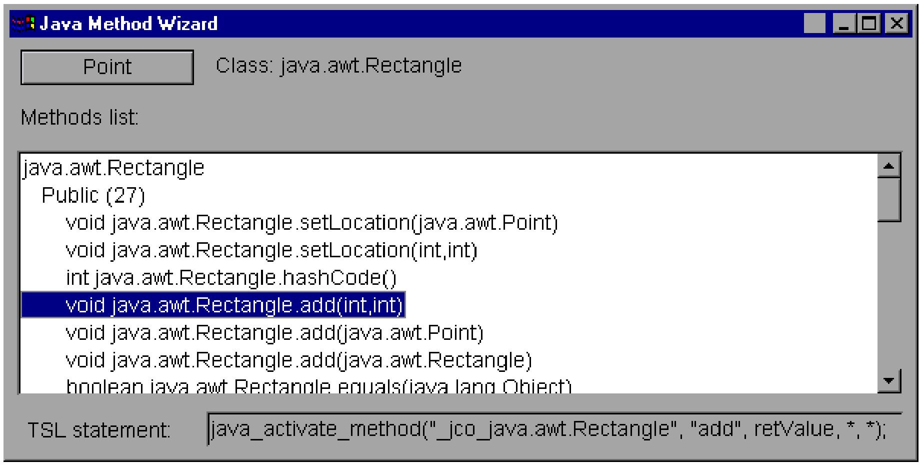
Point   Class: java.awt.Rectangle

Methods list:

```
java.awt.Rectangle
   Public (27)
   Protected (0)
   Default (0)
   Private (0)
java.lang.Object
   Public (9)
   Protected (2)
   Default (0)
```

TSL statement:

6 Double-click a method type to view the related methods.


Java Method Wizard

Point   Class: java.awt.Rectangle

Methods list:

```
java.awt.Rectangle
   Public (27)
      void java.awt.Rectangle.setLocation(java.awt.Point)
      void java.awt.Rectangle.setLocation(int,int)
      int java.awt.Rectangle.hashCode()
      void java.awt.Rectangle.add(int,int)
      void java.awt.Rectangle.add(java.awt.Point)
      void java.awt.Rectangle.add(java.awt.Rectangle)
      boolean java.awt.Rectangle.equals(java.lang.Object)
```

TSL statement:
```
java_activate_method("_jco_java.awt.Rectangle", "add", retValue, *, *);
```

To generate the TSL statement for invoking a Java method:

1. Activate the Java Method Wizard as described on page 32.
2. Select the method that you want to invoke from the list of methods under the appropriate object class. A TSL statement is displayed in the TSL statement box.

Note: If you run a Java application on a virtual machine earlier than JDK version 1.2, the **java_activate_method** function cannot invoke Protected, Default (i.e., package), or Private method types.

---

3. Copy the statement displayed in the **TSL statement** box and paste it into your script.
4. Replace the * symbols in the statement with the parameter values you want to send to the method.

   For example, if you created a Rectangle object, and you want to enlarge it by one pixel in each direction, copy the TSL statement displayed in the TSL statement box:

   ```
   rc = java_activate_method(newRectangle, "add", retValue, *, *);
   ```

   and replace each * symbol with 1 as shown below:

   ```
   rc = java_activate_method(newRectangle, "add", retValue, 1, 1);
   ```

Firing Java Events

You can simulate an event on a Java object during a test run with the **java_fire_event** function. This function has the following syntax:

java_fire_event (*object* **,** *class* **[,** *constructor_param*$_1$**,...,** *contructor_param*$_n$ **]);**

The *object* parameter is the logical name of the Java object. The *class* parameter is the name of the Java class representing the event to be activated. The *constructor_param*$_n$ parameters are the required parameters for the object constructor (excluding the object source, which is specified in the *object* parameter).

Note: The constructor's Event ID argument may be entered as the ID number or the final field string that represents the Event ID.

For example, you can use the **java_fire_event** function to fire a MOUSE_CLICKED event using the following script:

```
set_window("mybuttonapplet.htm", 2);
java_fire_event ("MyButton", "java.awt.event.MouseEvent",
"MOUSE_CLICKED", get_time(), "BUTTON1_MASK", 4, 4, 1, "false");
```

In the example above, the constructor has the following parameters: int id, long when, int modifiers, int x, int y, int clickCount, boolean popupTrigger, where id = "MOUSE_CLICKED" , when = get_time() , modifiers = "BUTTON1_MASK", x = 4, y = 4, clickCount = 1, popupTrigger = "false".

4

Configuring Custom Java Objects

This chapter explains how to add Java objects to the GUI map and to configure custom Java objects as standard GUI objects.

This chapter describes:

- About Configuring Custom Java Objects
- Adding Custom Java Objects to the GUI Map
- Configuring Custom Java Objects with the Java Custom Object Wizard

About Configuring Custom Java Objects

With the Java Add-in you can use WinRunner to record test scripts on most Java applications and applets, just like you would in any other Windows application. If you record an action on a custom or unsupported Java object, however, WinRunner maps the object to the general object class in the WinRunner GUI map. When this occurs, you can use the Java Custom Object Wizard to configure the GUI map to recognize these Java objects as a push button, check button, static text, or text field. This makes the test script easier to read and makes it easier for you to perform checks on relevant object properties.

After using the wizard to configure a custom object, you can add it to the GUI map, record actions, and run it as you would any other WinRunner test.

Adding Custom Java Objects to the GUI Map

Once the Java Add-in is loaded, you can add custom Java objects to the GUI map by recording an action or by using the GUI Map Editor to learn the objects. By default, however, these objects will each be mapped to the general object class, and activities performed on those objects will generally result in generic **obj_mouse_click** or **win_mouse_click** statements. The objects will usually be identified in the GUI map by their label property, or if WinRunner does not recognize the label, by a numbered class_index property.

For example, suppose you wish to record a test on a sophisticated subway routing Java application. This application lets you select your starting location and destination, and then suggests the best subway route to take. The application allows you to select which train line(s) you prefer to use for your travels.

Since WinRunner cannot recognize the custom Java check boxes in the subway application as GUI objects, when you check one of the options, the GUI map defines the objects as:

```
{
class: object,
label: "M (Nassau St Express)"
}
```

If you were to record a test in which you selected the "M", "A", and "Six" lines as your preferred lines, WinRunner would create a test script similar to the following:

```
set_window("Line Selection", 1);
obj_mouse_click("M (Nassau St Express)", 6, 32, LEFT);
obj_mouse_click("A (Far Rockaway) (Eighth Av...", 10, 30, LEFT);
obj_mouse_click("Six (Lexington Ave Local)", 5, 27, LEFT);
```

The test script above is difficult to understand. If, instead, you use the Java Custom Object Wizard in order to associate the custom objects with the check button class, WinRunner records a script similar to the following:

```
set_window("Line Selection", 8);
button_set("M (Nassau St Express)", ON);
button_set("A (Far Rockaway) (Eighth Av...", ON);
button_set("Six (Lexington Ave Local)", ON);
```

Now it is easy to see that the objects in the script are check buttons and that the user selected (turned ON) the three check buttons.

Configuring Custom Java Objects with the Java Custom Object Wizard

You configure a custom Java object in WinRunner using the Java Custom Object Wizard to assign the object to a standard GUI class and to object properties that uniquely identify the object.

Note: The GUI Map Configuration tool does not support configuring Java objects. The Java Custom Object Wizard serves a similar purpose for Java objects to that which the regular GUI Map Configuration tool serves for Windows objects. Because Java objects do not have a handle or window (and therefore no MSW class), the regular GUI Map Configuration tool is unable to perform a **set_class_map** type mapping. Thus, when you want to map a custom Java object to a standard class, always use the Java Custom Object Wizard option. For more information about the GUI Map Configuration tool, refer to the *WinRunner User's Guide*.

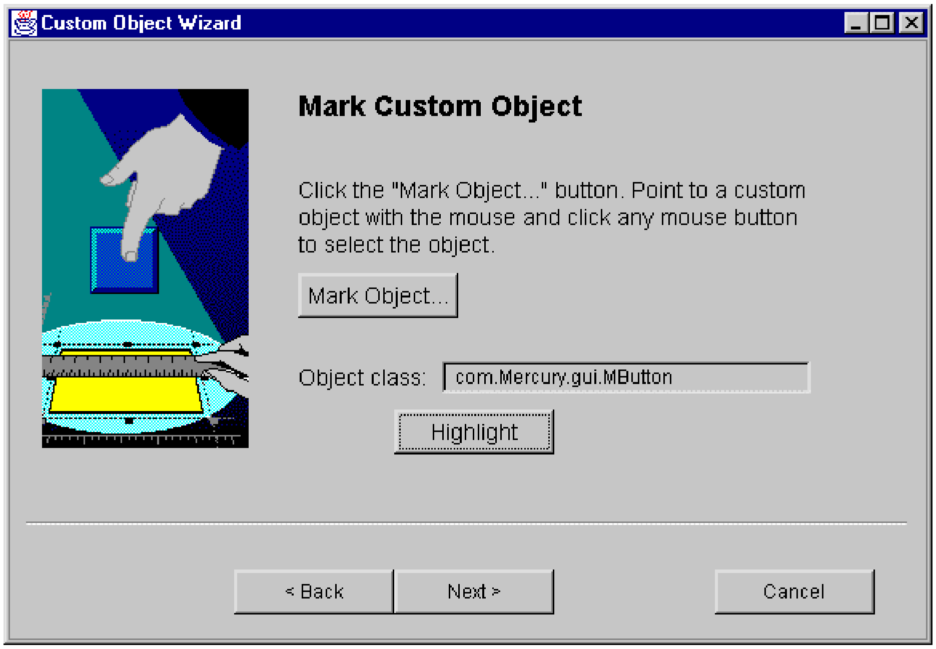
To configure a Java object using the Java Custom Object Wizard:

1 Open your Java application containing custom Java objects.

2 Open a new test in WinRunner.

3 Choose **Tools** > **Java Custom Object WIzard**. The Custom Object Wizard Welcome screen opens. Click **Next**.

4 Click the **Mark Object** button. Point to an object in the Java application. The object is highlighted. Click any mouse button to select the object. A default name appears in the **Object class** field.



Mark Custom Object

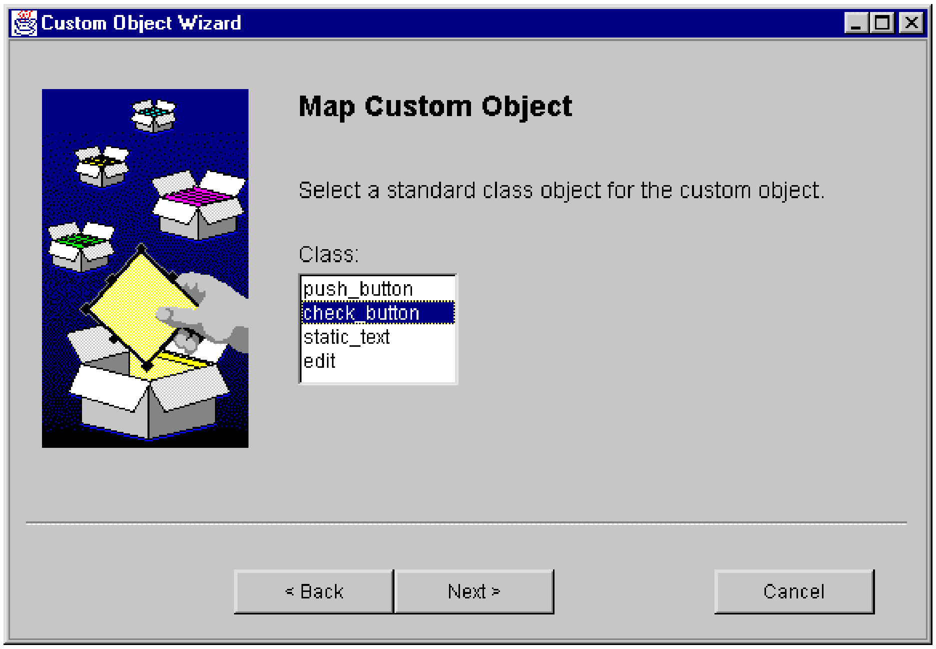
Click the "Mark Object..." button. Point to a custom object with the mouse and click any mouse button to select the object.

Object class: com.Mercury.gui.MButton

5 Click the **Highlight** button if you want to confirm that the correct option was selected. The object you selected is highlighted.

6 If you want to select a different object, repeat steps 4 and 5. When you are satisfied with your selection, click **Next**.

7 Select a standard class object for the object you selected. Click **Next**.

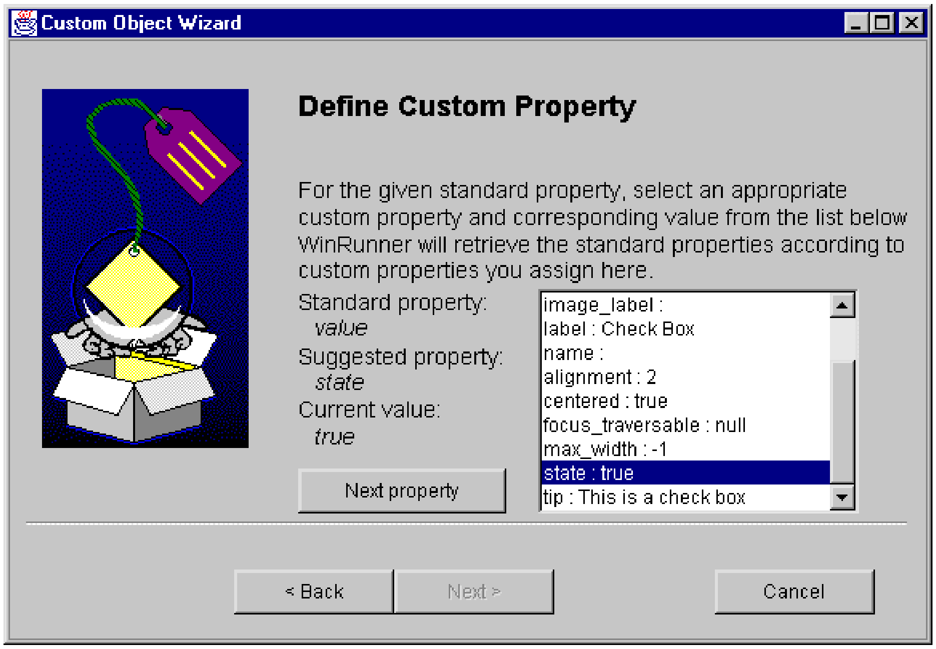

Custom Object Wizard

Map Custom Object

Select a standard class object for the custom object.

Class:
push_button
check_button
static_text
edit

8 Select an appropriate custom property and corresponding property value from the property list on the right to uniquely identify the object, or accept the suggested property and value.



For the given standard property, select an appropriate custom property and corresponding value from the list below WinRunner will retrieve the standard properties according to custom properties you assign here.

Standard property:
value
Suggested property:
state
Current value:
true

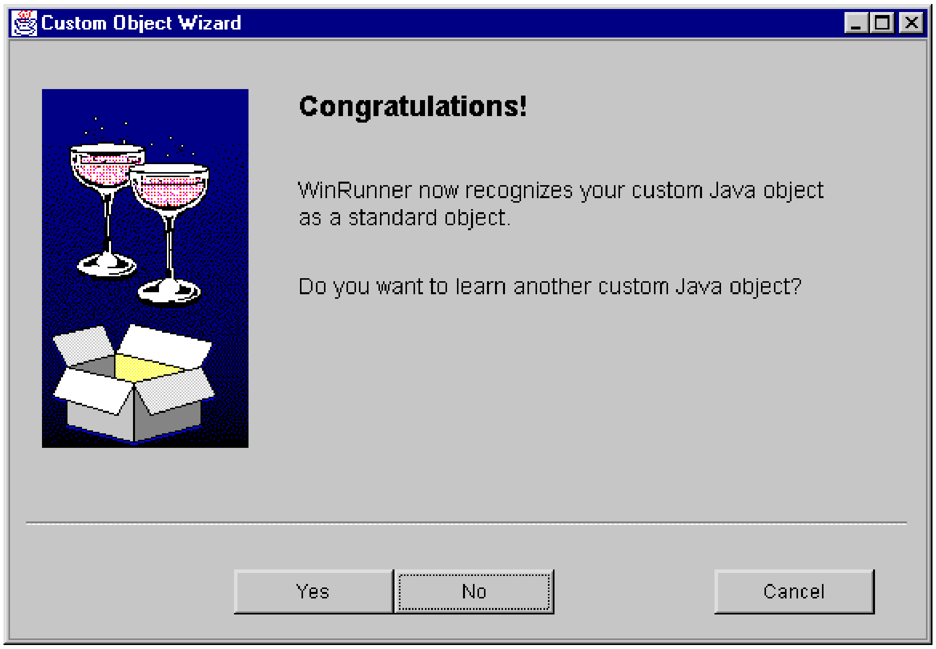
```
image_label :
label : Check Box
name :
alignment : 2
centered : true
focus_traversable : null
max_width : -1
state : true
tip : This is a check box
```

If you selected check_button as the standard object, two custom properties are necessary. After selecting the first property, click **Next Property** to select the second property for the object.

Click **Next.**

9 The Congratulations screen opens. If you want WinRunner to learn another custom Java object, click **Yes**. The wizard returns to the Mark Custom Object screen.

10 Repeat steps 4-8 for each custom object you want to configure. If you are finished configuring custom Java options, click **No**.

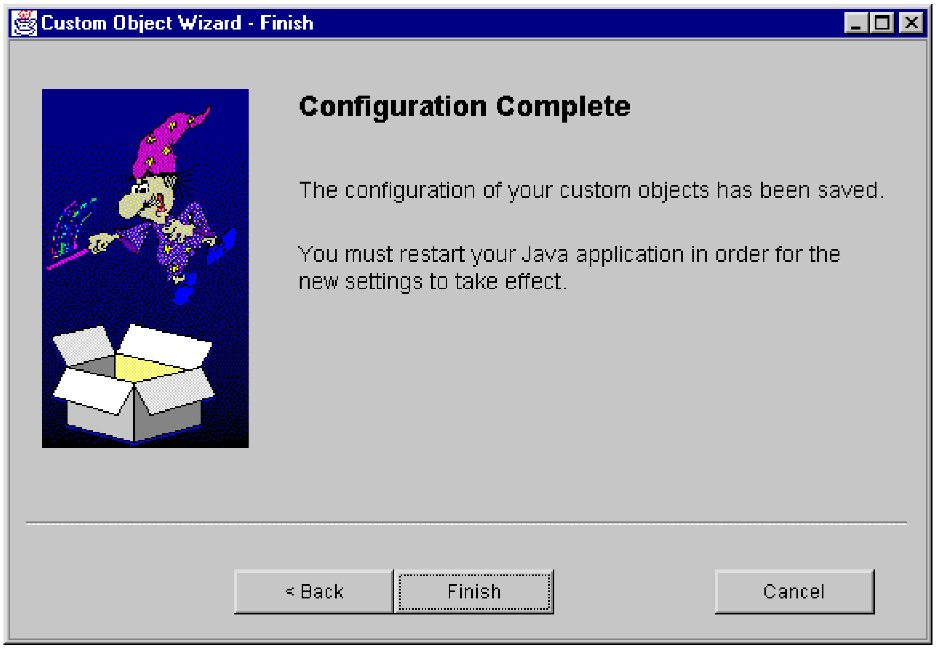


Congratulations!

WinRunner now recognizes your custom Java object as a standard object.

Do you want to learn another custom Java object?

11 The Finish screen opens. Click **Finish** to close the Custom Object Wizard.



Configuration Complete

The configuration of your custom objects has been saved.

You must restart your Java application in order for the new settings to take effect.

12 Close and reopen your Java application or applet to activate the new configuration for the object(s).

Note: The new object configuration settings will not take effect until the Java application or applet is restarted.

Once you have configured a custom Java option using the Java Custom Object Wizard, you can add the objects to the GUI map or record a test as you would in any Windows application. For more information on the GUI map and recording scripts, refer to the *WinRunner User's Guide*.

Note: When you configure custom Java objects in WinRunner, the **Program Files\Common Files\Mercury Interactive\SharedFiles\JavaAddin\classes\customization.properties** file is created and contains information about the custom Java objects. If you want to modify your custom Java configurations, or no longer want to use them, delete the custom Java objects in the GUI Map and delete the **customization.properties** file. Then restart your Java application or applet.

5

Troubleshooting Testing Java Applets and Applications

This chapter is intended to help pinpoint and resolve some common problems that may occur when testing Java applets and applications.

This chapter describes:

- ➤ Common Problems and Solutions
- ➤ Checking Java Environment Settings
- ➤ Locating the Java Console
- ➤ Accessing Java Add-in DLL Files
- ➤ Running an Application or Applet with the Same Settings
- ➤ Running the Java Add-in without Multi-JDK Support (Advanced)
- ➤ Disabling the Multi-JDK Support

Common Problems and Solutions

The Java Add-in provides a number of indicators that help you identify whether your add-in is properly installed and functioning. The following table describes the indicators you may see when your add-in is not functioning properly, and suggests possible solutions:

Indicator	Solution
The Java Add-in is not displayed in the Add-in Manager.	View the **install.log** file located in the **<WinRunner Installation folder>\dat** folder for information about the add-in installation that you performed.
The Java Support Activation Tool is not visible in the taskbar tray.	Invoke the Java Support Activation Tool: Click **Programs** > **WinRunner** > **Java Add-in** > **Java Add-in Switching Tool** in the Start menu. or Invoke **JavaSupportSwitch.exe** in **Program Files\Common Files\Mercury Interactive\SharedFiles\JavaAddin\bin**.
The Java Support Activation Tool is disabled.	Click the Java Support Activation Tool in the taskbar tray to enable it. For more information, refer to the *WinRunner Java Add-in Installation Guide*. Note that the Java Support Activation Tool only affects Java applications activated after you disable or enable the support. If a Java application is already running without Mercury Java support, you must close and restart it.
The Java console does not display a line containing the text "Loading Mercury Interactive Support."	Check that the settings in your environment correspond to the environment settings defined in this chapter, or check for a batch file that may override the settings. For more information, see: • "Checking Java Environment Settings" on page 50 • "Locating the Java Console" on page 52

Indicator	Solution
The Java console contains messages about .dll files. (This message is usually followed by UnsatisfiedLinkError messages.)	Check that you have write permission for the **jre\bin** folder, or place the Java Add-in basic .dll files in the **jre\bin** folder. For more information see: • "Accessing Java Add-in DLL Files" on page 54 • "Locating the Java Console" on page 52
A different applet or application works with WinRunner but the application you want to test does not work.	First check whether you can record and run tests if you invoke the other Java applet or application using exactly the same settings. Check that the settings in your environment correspond to the environment settings defined in this chapter, or check for a batch file that may override the settings. For more information, see: • "Running an Application or Applet with the Same Settings" on page 55 • "Checking Java Environment Settings" on page 50
The add-in does not function properly with applications that run with the –Xincgc option.	Either remove the option or run without the multi-JDK support. For more information, see: • "Running the Java Add-in without Multi-JDK Support (Advanced)" on page 55 • "Disabling the Multi-JDK Support" on page 58
Your Java console contains the line Could not find –Xrun library: micsupp.dll.	Check that you have **micsupp.dll** in your system folder (**WINNT\system32 or windows\system**).

Note for Netscape 4.x users: If you experience any unusual behavior in the Java support (for example, if the message "Loading Mercury Interactive Support" does not appear in the Java Console), try disabling the JIT (Just-In-Time) compiler. To do this, locate and rename the **jit3240.dll** file in **Communicator\Program\java\bin** and then restart Netscape.

If, after reviewing the above indicators and solutions, you are still unable to record and run tests on your Java applet or application, contact Mercury Interactive Customer Support.

Checking Java Environment Settings

This section describes the environment settings you need for loading your Java application with WinRunner Java Add-in support. For all the environments, you need to set one or more environment variables to the short path name of the Java Add-in support classes folder.

Note: The short path (also known as the 8.3 DOS name) of the Java Add-in classes folder (**Common Files\Mercury Interactive\Shared files\JavaAddin\classes**) can usually be obtained by examining the value of the mic_classes environment variable. This may be useful when defining the environment settings.

Sun Plug-in 1.4.1 or IBM Java 2 (version 1.2 or higher)

➤ Set the _JAVA_OPTIONS environment variable (Sun) or the IBM_JAVA_OPTIONS environment variable (IBM) as follows:

-Dawt.toolkit=mercury.awt.awtSW -Xrunmicsupp
-Xbootclasspath/a:<**common_files**>\MERCUR~1\SHARED~1\JAVAAD~1\
classes;<**common_files**>\MERCUR~1\SHARED~1\JAVAAD~1\classes\mic.jar

The above settings should appear on one line (no new line separators).

Note that **common_files** denotes the short path of the Common Files folder located in the Program Files folder. For example, if the Common Files folder is in **C:\Program Files\Common Files**, then the value for –Xbootclasspath is as follows:

```
-Xbootclasspath/a:C:\PROGRA~1\COMMON~1\MERCUR~1\SHARED~1\
JAVAAD~1\classes;C:\PROGRA~1\COMMON~1\MERCUR~1\SHARED~1\
JAVAAD~1\classes\mic.jar
```

Java 1.1.x

➤ The classpath environment variable should contain:

<**common_files**>\MERCUR~1\SHARED~1\JAVAAD~1\classes;
<**common_files**>\MERCUR~1\SHARED~1\JAVAAD~1\classes\mic.jar

Note that **common_files** denotes the short path of the Common Files folder located in the Program Files folder. For example, if the Common Files folder is in **C:\Program Files\Common Files** then the value for classpath is as follows:

```
C:\PROGRA~1\COMMON~1\MERCUR~1\SHARED~1\JAVAAD~1\classes;C:\
PROGRA~1\COMMON~1\MERCUR~1\SHARED~1\JAVAAD~1\classes\mic.jar
```

➤ The _classload_hook environment variable should be set to micsupp.

Microsoft Java Virtual Machine (JVM) - Internet Explorer/Jview

➤ The classpath environment variable should contain:

<**common_files**>\MERCUR~1\SHARED~1\JAVAAD~1\classes;
<**common_files**>\MERCUR~1\SHARED~1\JAVAAD~1\classes\mic.jar

Note that **common_files** denotes the short path of the Common Files folder located in the Program Files folder. For example, if the Common Files folder is in **C:\Program Files\Common Files** then the value for classpath is as follows:

```
C:\PROGRA~1\COMMON~1\MERCUR~1\SHARED~1\JAVAAD~1\classes;C:\
PROGRA~1\COMMON~1\MERCUR~1\SHARED~1\JAVAAD~1\classes\mic.jar
```

➤ The MSJAVA_ENABLE_MONITORS variable should be set to 1. This is relevant when the user who installed WinRunner with Java Add-in support is not the user running the application.

,**Netscape 4.x**

➤ The classpath environment variable should contain:

<**common_files**>\MERCUR~1\SHARED~1\JAVAAD~1\classes;
<**common_files**>\MERCUR~1\SHARED~1\JAVAAD~1\classes\mic.jar

Note that **common_files** denotes the short path of the Common Files folder located in the Program Files folder. For example, if the Common Files folder is in **C:\Program Files\Common Files** then the value for classpath is as follows:

C:\PROGRA~1\COMMON~1\MERCUR~1\SHARED~1\JAVAAD~1\classes;C:\
PROGRA~1\COMMON~1\MERCUR~1\SHARED~1\JAVAAD~1\classes\mic.jar

Netscape 6.x

Netscape 6.x uses the Java 2 Virtual Machine. For more information, see "Sun Plug-in 1.4.1 or IBM Java 2 (version 1.2 or higher)" on page 50.

Locating the Java Console

The Java console is the window in which your Java application displays messages. The location of the Java console changes according to your application setup, as follows:

If your Java application is a standalone application:

➤ Open the batch file or shortcut that invokes the application and look for the command that launched Java (**java.exe**, **javaw.exe**, **jre.exe**, or **jrew.exe**).

➤ If the application was run with **java.exe** or **jre.exe**, it will load with a console (Command prompt window).

- If the application was run with **javaw.exe** or **jrew.exe**, the console is not available. To check for Java Add-in support, invoke the application with **java.exe** or **jre.exe**. Do this by altering your batch file or the shortcut invoking your application. Note that except for how they launch a console window, **java.exe** and **javaw.exe** are identical and **jre.exe** and **jrew.exe** are identical.

If your application runs in an AppletViewer:

- Look in the DOS command prompt window that invoked the AppletViewer.
- If there is no DOS command prompt window, your AppletViewer may be run by a batch file just like a standalone application. See the information about **javaw** and **jrew** in the standalone section above.

If your application runs in Internet Explorer or Netscape 6.x:

- If your application runs with the Sun Java plug-in:
  - Right-click the Java plug-in icon in your taskbar tray and click **Show Console**.
  - For JDK 1.4 only, if you do not see the Java plug-in icon in your taskbar tray, select **Settings** > **Control Panel** in the Start menu and double-click the Java plug-in icon (choose the icon for the Java version used by your application). In the Basic tab, select the **Show Java console** option and click **Apply**. Restart the browser.

Note: To find out whether your Internet Explorer works with the Sun Java plug-in, select **Tools** > **Internet Options** > **Advanced**. Under **Java (Sun)** verify that **Use Java** is selected. Note that Java plug-in version 1.3 or later automatically configures Internet Explorer to work with the Sun Java plug-in.

- If your application runs with the Internet Explorer internal Virtual Machine, in Internet Explorer select **Tools** > **Internet Options**. In the Advanced tab, look for **Microsoft VM**. Select **Java console enabled (requires restart)** and click **OK**. Restart the browser and invoke your application. Select **View** > **Java Console**.

If your application runs in Netscape 4.x:

➤ Select **Start** > **Programs** > **Communicator** > **Tools** > **Java Console**.

Accessing Java Add-in DLL Files

For the Java Add-in to work properly, two .dll files must be accessible to the Java Virtual Machine (JVM): **mic_if2c.dll** and **mic_if2c_aqt.dll**. In most cases, the Java Add-in installs these files in the **jre\bin** folder of your Java environment when your Java application starts.

However, if you do not have write permission in the **jre\bin** folder, the Java Add-in fails to copy the .dll files. In this situation, messages similar to the following appear in the Java console:

```
Error: The file S:\JAVA\JDK1.4.0\jre\bin\mic_if2c.dll is missing.
Error: The file S:\JAVA\JDK1.4.0\jre\bin\mic_if2c_aqt.dll is missing.
```

To fix this problem, either make sure that you have write permission to the **jre\bin** folder, or manually copy the .dll files from the **Common Files\Mercury Interactive\SharedFiles\JavaAddin\bin** folder.

A variant of this problem is if you have the wrong version of the .dll files in the folder (for example, if you installed an earlier version of WinRunner or QuickTest Professional, the folder may contain .dll files from a previous version of the Java Add-in). If the files and the folder are accessible when you install the latest version of WinRunner with Java support, the Java Add-in replaces the files automatically.

If the files or the **jre\bin** folder are write protected or if another process is using the .dll files, messages similar to the following appear in the Java console:

```
Warning: The file S:\JAVA\JDK1.4.0\jre\bin\mic_if2c.dll does not match your
current Java Add-in installation version.
Warning: The file S:\JAVA\JDK1.4.0\jre\bin\mic_if2c_aqt.dll does not match your
current Java Add-in installation version.
```

To fix the problem, either make sure that the files are not write-protected, or manually copy the correct version of the files to the **jre\bin** folder from the **Common Files\Mercury Interactive\SharedFiles\JavaAddin\bin** folder.

Running an Application or Applet with the Same Settings

In some cases, running another Java application or applet with the exact same settings helps determine whether you are encountering a general problem with the Java Add-in or an application-specific problem.

To run an application or applet with the same settings:

1. Determine whether the application is a standalone application or an applet.
2. If the application is an applet, check the browser type.
3. If the applet is executed from a shortcut, execute the applet with the same command.
4. If the applet is executed from a batch file, copy the batch file and only change the class file to invoke it. Note that if the classpath must also be changed, add only the new items needed. Do not remove any of the items from the original application or applet classpath.

Running the Java Add-in without Multi-JDK Support (Advanced)

The Java Add-in uses a mechanism for supporting multiple JDK versions without configuration changes (multi-JDK support). This mechanism uses the profiler interface of the Java Virtual Machine (JVM) to adjust the Java Add-in support classes according to the JDK version used. If, for some reason, this mechanism does not work, you can still use the Java Add-in if you manually configure the Java environment.

Java 2

The multi-JDK support mechanism is invoked by the –Xrunmicsupp option supplied to the JVM. If you want to disable the multi-JDK support, remove the –Xrunmicsupp option from the JDK settings (by default, it is located in the _JAVA_OPTIONS or IBM_JAVA_OPTIONS environment variable).

Next, you should change the –Xbootclasspath setting to list the correct patches folder according to the JDK version you are using. If you do not know which version you are using, execute the java –fullversion command to retrieve the exact version (including the minor version number).

To determine the folder that you need to add to the Xbootclasspath, go to **Common Files\Mercury Interactive\SharedFiles\JavaAddin\Patches**. Select the patches folder appropriate for your JDK version (the latest version number that is not later than your version). For example, if you are using Sun’s JDK 1.3.0_05, select **jdk\1.3.0_02** as your patches folder.

In addition, you need to add the **default** patches folder under **Common Files\Mercury Interactive\SharedFiles\JavaAddin\Patches**.

To set your Java environment to load from the abovementioned folder, you need to set the –Xbootclasspath option. You can set it in the _JAVA_OPTIONSor IBM_JAVA_OPTIONS environment variable, or supply it as a parameter in the Java invocation command. The –Xbootclasspath should be set to the following value:

-Xbootclasspath/p:<**patches folder**>;<**default patches folder**>;
<**common_files**>
\MERCUR~1\SHARED~1\JAVAAD~1\classes;<**common_files**>
\MERCUR~1\SHARED~1\JAVAAD~1\classes\mic.jar

Note that instead of the usual setting of -Xbootclasspath/a:..., you should use /p (to prepend the path rather than append).

For example, if you are using JDK1.2.2_007, and **common_files** is **C:\Program Files\Common Files**, the value is:

```
-Xbootclasspath/p: C:\PROGRA~1\
COMMON~1\MERCUR~1\SHARED~1\JAVAAD~1\Patches\jdk\1.2.2_005;
C:\PROGRA~1\
COMMON~1\MERCUR~1\SHARED~1\JAVAAD~1\Patches\default;
C:\PROGRA~1\ COMMON~1\MERCUR~1\SHARED~1\JAVAAD~1\classes;
C:\PROGRA~1\
COMMON~1\MERCUR~1\SHARED~1\JAVAAD~1\classes\mic.jar
```

In addition, you need to copy **mic_if2c.dll** and **mic_if2c_aqt.dll** from the **Common Files\Mercury Interactive\SharedFiles\JavaAddin\bin** folder to the **jre\bin** folder of the JDK or JRE you are using.

Java 1

The multi-JDK support mechanism is invoked by the _classload_hook environment variable. If you need to remove the multi-JDK support, remove the _classload_hook from the JDK settings by deleting the environment variable.

Next, you should change the classpath setting to list the correct patches folder according to the JDK version you are using. If you do not know which version you are using, execute the java –fullversion command to retrieve the exact version (including the minor version number).

To determine the patches folder you need to add to the classpath, go to **Common Files\Mercury Interactive\SharedFiles\JavaAddin\Patches**. Select the patches folder appropriate for your JDK version (the latest version number that is not later than your version). For example, if you are using Sun's JDK 1.1.8_ 005, select **jdk\1.1.8** as your patches folder.

In addition, you need to add the **default** patches folder under **Common Files\Mercury Interactive\SharedFiles\JavaAddin\Patches**.

To set your Java environment to load from the abovementioned folder, you need to add a few entries in the beginning of the classpath option. You can set it in the classpath environment variable, or supply it as a parameter in the Java invocation command. The classpath should be set to the following value:

-classpath <**patches folder**>;<**default patches folder**>;<**common_files**>
\MERCUR~1\SHARED~1\JAVAAD~1\classes;<**common_files**>
\MERCUR~1\SHARED~1\JAVAAD~1\classes\mic.jar

Note that usually the classpath should include the Java core classes archive (**classes.zip** or **rt.jar**) as well as other entries needed for running your Java application. Make sure these entries are kept.

For example, if you are using JDK1.1.8, and the Common Files folder is **C:\Program Files\Common Files**, the value is:

```
-classpath C:\PROGRA~1\
COMMON~1\MERCUR~1\SHARED~1\JAVAAD~1\Patches\jdk\1.1.8;
C:\PROGRA~1\
COMMON~1\MERCUR~1\SHARED~1\JAVAAD~1\Patches\default;
C:\PROGRA~1\ COMMON~1\MERCUR~1\SHARED~1\JAVAAD~1\classes;
C:\PROGRA~1\
COMMON~1\MERCUR~1\SHARED~1\JAVAAD~1\classes\mic.jar
```

You also need to copy **mic_if2c.dll** and **mic_if2c_aqt.dll** from the **Common Files\Mercury Interactive\SharedFiles\JavaAddin\bin** folder to the **jre\bin** folder of the JDK or JRE you are using.

Disabling the Multi-JDK Support

The multi-JDK does not work when using the incremental garbage collector (-Xincgc option). If the –Xincgc option is absolutely required, follow the instructions in "Running the Java Add-in without Multi-JDK Support (Advanced)" on page 55, to enable you to use the Java Add-in.

Index

MERCURY
INTERACTIVE

Mercury Interactive Corporation
1325 Borregas Avenue
Sunnyvale, CA 94089 USA

Main Telephone: (408) 822-5200
Sales & Information: (800) TEST-911, (866) TOPAZ-4U
Customer Support: (877) TEST-HLP
Fax: (408) 822-5300

Home Page: www.mercuryinteractive.com
Customer Support: support.mercuryinteractive.com

WRJAVAUG7.6/01