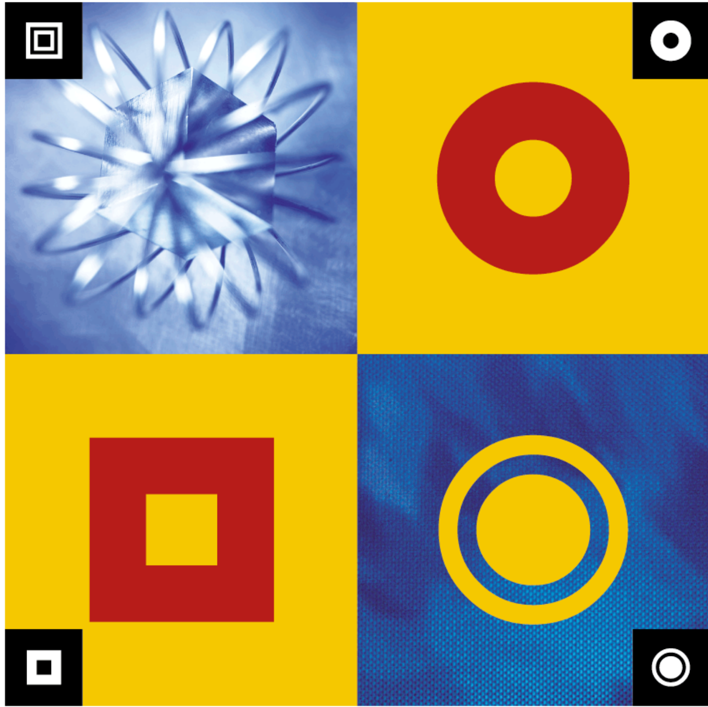
MERCURY INTERACTIVE

WinRunner

ORACLE® ADD-IN



USER'S GUIDE

WinRunner Oracle® Add-in

User's Guide

Version 7.6

MERCURY INTERACTIVE

WinRunner Oracle Add-in User's Guide, Version 7.6

This manual, and the accompanying software and other documentation, is protected by U.S. and international copyright laws, and may be used only in accordance with the accompanying license agreement. Features of the software, and of other products and services of Mercury Interactive Corporation, may be covered by one or more of the following patents: U.S. Patent Nos. 5,701,139; 5,657,438; 5,511,185; 5,870,559; 5,958,008; 5,974,572; 6,138,157; 6,144,962; 6,205,122; 6,237,006; 6,341,310; 6,360,332, 6,449,739; 6,470,383; 6,477,483; 6,549,944; 6,560,564; and 6,564,342. Other patents pending. All rights reserved.

ActiveTest, ActiveTune, Astra, FastTrack, Global SiteReliance, LoadRunner, Mercury, Mercury Interactive, the Mercury Interactive logo, Open Test Architecture, Optane, POPs on Demand, ProTune, QuickTest, RapidTest, SiteReliance, SiteRunner, SiteScope, SiteSeer, TestCenter, TestDirector, TestSuite, Topaz, Topaz AIMS, Topaz Business Process Monitor, Topaz Client Monitor, Topaz Console, Topaz Delta, Topaz Diagnostics, Topaz Global Monitor, Topaz Managed Services, Topaz Open DataSource, Topaz Real User Monitor, Topaz WeatherMap, TurboLoad, Twinlook, Visual Testing, Visual Web Display, WebTest, WebTrace, WinRunner and XRunner are trademarks or registered trademarks of Mercury Interactive Corporation or its wholly owned subsidiary Mercury Interactive (Israel) Ltd. in the United States and/or other countries.

All other company, brand and product names are registered trademarks or trademarks of their respective holders. Mercury Interactive Corporation disclaims any responsibility for specifying which marks are owned by which companies or which organizations.

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

© 2003 Mercury Interactive Corporation, All rights reserved

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

WRORAUG7.6/01

Table of Contents

Chapter 1: Introduction 1
Using the Oracle Add-in 1
How the Oracle Add-in Identifies Java Objects 2
Activating the Oracle Add-in 3

Chapter 2: Testing Standard Java Objects 5
About Testing Standard Java Objects 5
Recording Context Sensitive Tests 5
Enhancing Your Script with TSL 6

Chapter 3: Working with Java Methods and Events 17
About Working with Java Methods and Events 17
Invoking Java Methods 18
Accessing Object Fields 20
Working with Return Values (Advanced) 22
Viewing Object Methods in Your Application or Applet 24
Firing Java Events 31
Using the Name Attribute for Oracle Application GUI Objects 32

Chapter 4: Troubleshooting Testing Oracle Applications 35
Common Problems and Solutions 36
Checking Java Environment Settings 37
Locating the Java Console 39
Accessing Oracle Add-in DLL Files 40
Running the Oracle Add-in without Multi-JDK Support (Advanced) 41
Disabling the Multi-JDK Support 43

Index 45

1

Introduction

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

This chapter describes:

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

Using the Oracle Add-in

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

The Oracle Add-in enables you to record and run tests on cross-platform Oracle Applications. You can record and run user actions on Java-based and Web-based Oracle Applications 11i, as well as applications designed with Oracle Forms, running Oracle JInitiator or Java Plug-in.

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

With the Oracle Add-in you can:

- Record operations on Java objects within Oracle Forms Applications just as you would any other Windows object with WinRunner.
- 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 Oracle 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. For more information on GUI maps, refer to the "Configuring the GUI Map" chapter in the *WinRunner User's Guide*.

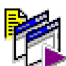
You can view the contents of your GUI map files in the GUI Map Editor by choosing **Tools** > **GUI Map Editor**. The GUI Map Editor displays the logical names and the physical descriptions of objects. For more information on GUI maps, refer to the "Understanding the GUI Map" section in the *WinRunner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

Activating the Oracle Add-in

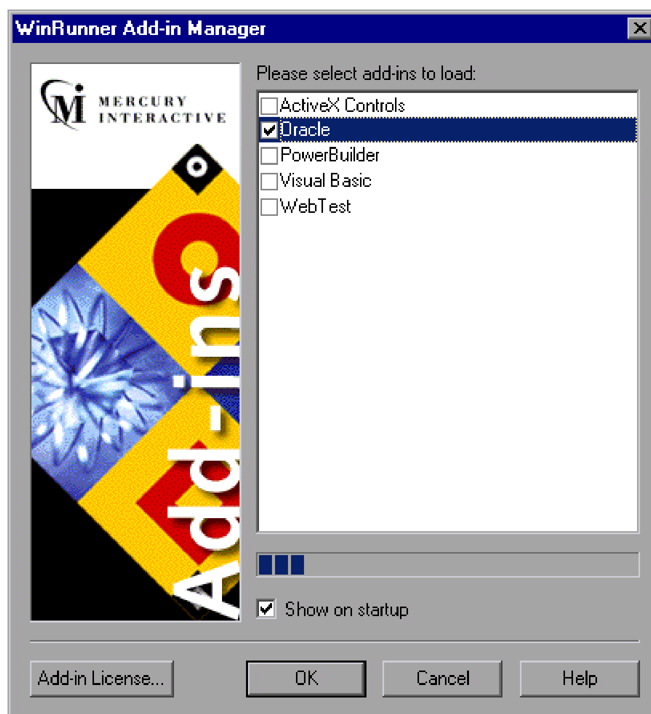
Before you begin testing your Oracle Application, make sure that you have installed all the necessary files and made any necessary configuration changes. For more information, refer to the *WinRunner Oracle Add-in Installation Guide*.

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

To activate the Oracle Add-in:



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



2 Select **Oracle**.

3 Click **OK**. WinRunner opens with the Oracle 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 Java objects within Oracle Applications and enhance scripts that test Oracle Applications.

This chapter describes:

- About Testing Standard Java Objects
- Recording Context Sensitive Tests
- Enhancing Your Script with TSL

About Testing Standard Java Objects

With the Oracle Add-in, you can record or write context sensitive scripts on all supported Oracle Forms objects. 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 Oracle Add-in loaded, support for the Oracle environments you installed will always be loaded. For more information about selecting Oracle environments, refer to the *WinRunner Oracle Add-in Installation Guide*.

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

If your Oracle Application uses Java objects from supported Oracle Forms, 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 object, WinRunner records a generic **obj_mouse_click** or **win_mouse_click** statement.

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.
- Select an item from a Java pop-up menu.
- Configure the way WinRunner learns object descriptions and runs tests on Oracle Applications.

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 21.

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");
```

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 Oracle Variable Settings

You can configure how WinRunner learns descriptions of objects, records and runs tests on Oracle Applications, 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:

COLUMN_NUMBER	In Oracle Applications, forms that appear to contain tables are actually implemented as a set of text fields and not as a table object. The Oracle Add-in can record and run tests on these forms as table objects and not as text fields. This variable specifies the minimum number of text field columns for a form with text fields to be considered a table object. Otherwise, the edit fields are treated as separate objects. This variable affects the way table objects are recorded as follows: if a test was recorded on a form as a table, it will be run as such regardless of the current value of the **COLUMN_NUMBER** variable. However, if a test was recorded on a form as separate text fields, then the **COLUMN_NUMBER** variable must reflect this during the test run, otherwise the test will fail.

	Note: If the **name** attribute is enabled (see “Using the Name Attribute for Oracle Application GUI Objects” on page 32), it is recommended that you record and run tests on these tables as text fields (by keeping the default value of **99**). **Default value: 99** **Note:** In earlier versions of WinRunner, the default value for the **COLUMN_NUMBER** variable was **2**. Tests recorded using the default value of **2** still run correctly, even when the new default value is **99**.
MAX_COLUMN_GAP	The maximum number of pixels between text fields in a form to be considered a column. This variable is used in conjunction with the **COLUMN_NUMBER** variable and is required only in rare cases. **Default value: 12**
MAX_LINE_DEVIATION	The maximum number of pixels between text fields in a form to be considered a on single line. This variable is used in conjunction with the **COLUMN_NUMBER** variable and is required only in rare cases. **Default value: 8**
MAX_ROW_GAP	The maximum number of pixels between text fields in a form to be considered one table row. This varable is used in conjunction with the **COLUMN_NUMBER** variable and is required only in rare cases. **Default value: 12**

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. "E"—generates a FocusGained event at the beginning of function execution. (AWT toolkit only) **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**"
SOFTKEYS_REC	Controls whether WinRunner records Oracle Application softkeys. By default, WinRunner does not record special function and action keys. "On" (or any non-zero numeric value)—enables recording of Oracle Application softkeys.
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 Oracle 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.

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
- Invoking Java Methods
- Accessing Object Fields
- Working with Return Values (Advanced)
- Viewing Object Methods in Your Application or Applet
- Firing Java Events
- Using the Name Attribute for Oracle Application GUI Objects

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 22. 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 22.

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 22.

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 22.

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 22.

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.

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 window, or any other Java object within the application. 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 Oracle 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 Oracle 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

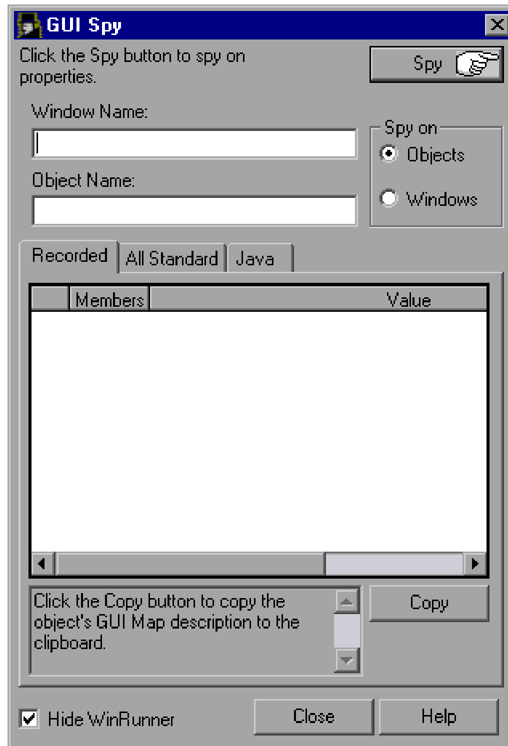
You can view all methods associated with GUI Java objects in your application and generate the appropriate **java_activate_method** function for a selected method using the Java tab of the GUI Spy.

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 using the GUI Spy:

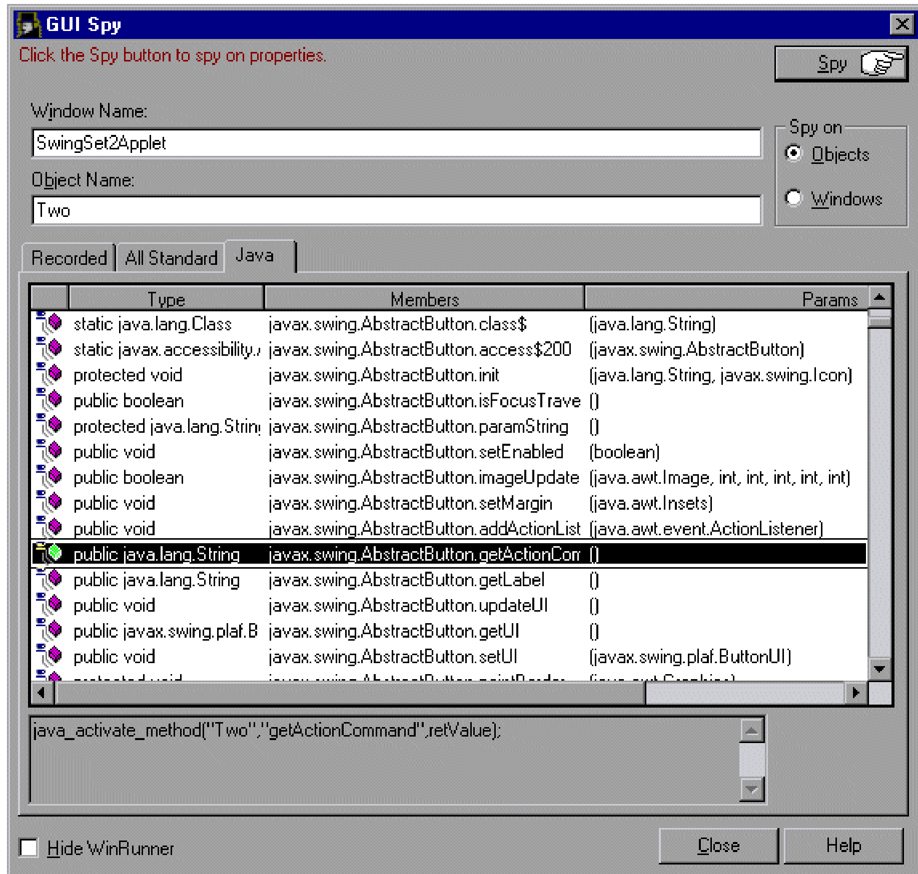
1 Open the Oracle Application that contains the object for which you want to view the methods.

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



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.



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 24.
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 an Oracle Application using Oracle JInitiator 1.1.x, 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 18.

   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.

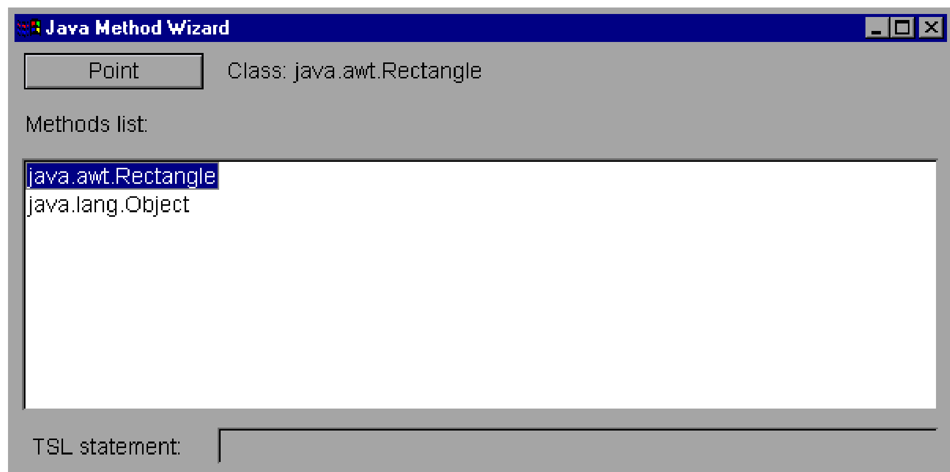
To view the methods for an object in your application:

1. Open the Oracle Application 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.

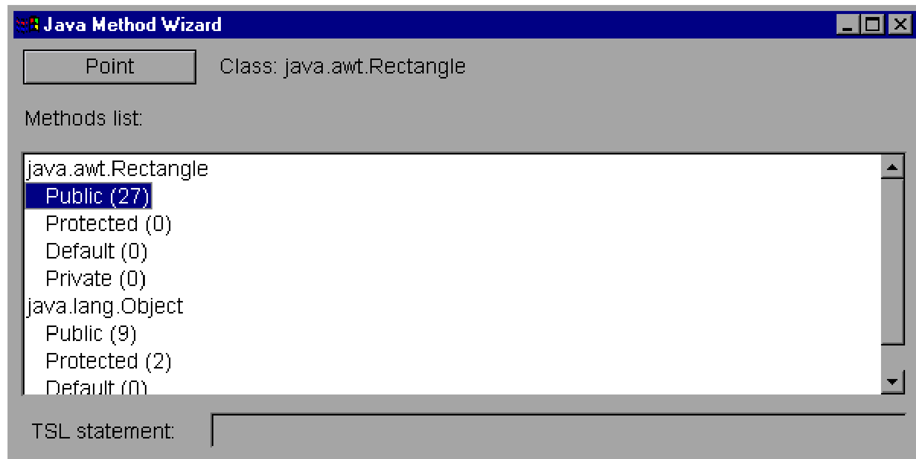
3. Select **Debug** run mode in the toolbar.
4. Choose **Debug** > **Step**, or click the **Step** button to run the statement. The Java Method Wizard opens and displays a list with the object’s class and all of its superclasses.



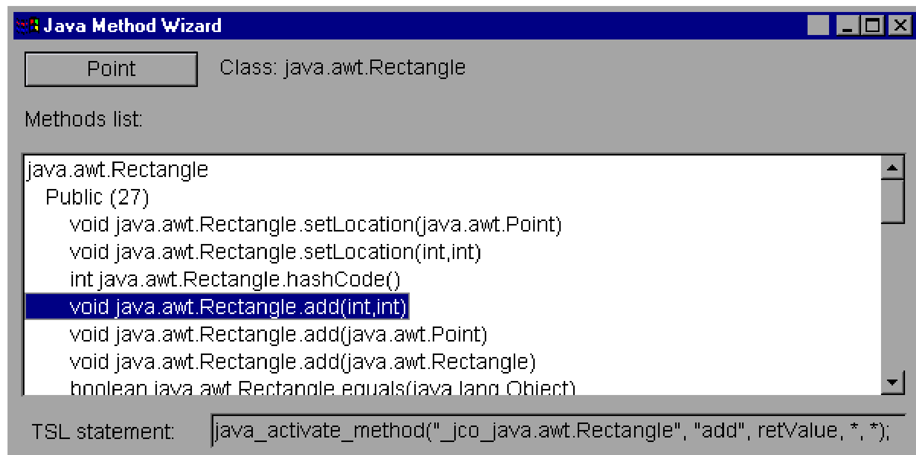
Note: After the Java Method Wizard opens, the focus returns to the main WinRunner window. You may need to select the Java Method Wizard icon on your Windows taskbar to display the wizard.

---

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



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



To generate the TSL statement for invoking a Java method:

1. Activate the Java Method Wizard as described on page 28.
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 an Oracle Application using Oracle JInitiator 1.1.x, 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".

Using the Name Attribute for Oracle Application GUI Objects

The Oracle Applications server can provide a unique **name** attribute for many GUI objects in the application. Using this attribute in the object description improves the reliability of the description and usually prevents the need for learning the **class_index** attribute that may change for a given object between the time you record and run a test.

Enabling the Name Attribute

To use the **name** attribute in your object descriptions, you must first enable the attribute supplied by the Oracle Applications server.

To enable the developer name when accessing the application directly:

1. Add record=names to the URL parameters.

   For example:
   http://oracleapps.mydomain.com:8002/dev60cgi/f60cgi?record=names

To enable the name attribute when using HTML to launch the Oracle Application:

1. In the startup HTML file that is used to launch the application, locate the line: <PARAM name="serverArgs .... fndnam= APPS">
2. Add the Oracle key: record=names. For example:

   <PARAM name="serverArgs" value="module=f:\FNDSCSGN userid=XYZ fndnam=apps?record=names">

To enable the name attribute when using the Personal Home Page to launch your Forms 6 Application:

Set up the following system profile option at your user level in order to enable the name attribute:

1. Sign on to your Oracle Application and select System Administrator responsibility.
2. Select **Nav** > **Profile** > **System**.

3 In the Find System Profile Values form:

- ➤ Confirm that **Display: Site and Users** contains your user logon.
- ➤ Enter %ICX%Launch% in the **Profile** box.
- ➤ Click the **Find** button.

4 Copy the value from the **Site** box of the **ICX: Forms Launcher** profile and paste it in the **User** box. Add &play=&record=names to the end of the URL in the **User** box.

5 Save your transaction.

6 Sign on again using your user name.

Note: If the **ICX: Forms Launcher** profile option is not UPDATABLE at the USER level, access **Application Developer** and select the **Updateable** check box for the **ICX_FORMS_LAUNCHER** profile.

Verifying that the Oracle Applications Server Provides a Unique Name Attribute

Before configuring WinRunner's GUI map to learn the name attribute, confirm that the Oracle Applications server supplies unique names. To check this, use the GUI Spy and point to some edit fields inside the Oracle application. Click the **All Standard** tab and view the name attribute for each. If the name is in all capital letters in the format FORM:BLOCK:FIELD or FORM_BLOCK_FIELD, then the **name** attribute is supplied correctly. If the name value is in the format ClassNameXXX, then the Oracle Applications server does not supply a unique name and you cannot use this attribute in the description.

Adding the Name Attribute to the GUI Map Configuration

By default, the **name** attribute is defined as an optional learned property in the record configuration of the following WinRunner classes: **push_button**, **check_button**, **radio_button**, **list**, **edit**. After you have configured the Oracle Applications server to supply a unique **name** attribute, you can add this property to the default set of properties learned by WinRunner as you record or learn Oracle objects.

For example, you could use the following statement to add the **name** attribute to an object's GUI map configuration:

```
set_record_attr("spin", "class name attached_text", "class_index", "index");
```

You should add the **name** attribute to other types of objects used in your application with additional **set_record_attr** statements.

Note that you can edit the **set_record_attr** calls made in the **java_supp init** script located in <**WinRunner Installation Folder**>**/lib** to load these settings by default with the Oracle Add-in.

4

Troubleshooting Testing Oracle Applications

This chapter is intended to help pinpoint and resolve some common problems that may occur when testing Oracle Applications.

This chapter describes:

- Common Problems and Solutions
- Checking Java Environment Settings
- Locating the Java Console
- Accessing Oracle Add-in DLL Files
- Running the Oracle Add-in without Multi-JDK Support (Advanced)
- Disabling the Multi-JDK Support

Common Problems and Solutions

The Oracle 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 Oracle 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** > **Oracle Add-in** > **Oracle 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 Oracle Add-in Installation Guide*. Note that the Java Support Activation Tool only affects Oracle Applications activated after you disable or enable the support. If an Oracle 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 37 • “Locating the Java Console” on page 39

<table>
<tr><th>Indicator</th><th>Solution</th></tr>
<tr><td>The Java console contains messages about .dll files.

(This message is usually followed by UnsatisfiedLinkError messages.)</td><td>Check that you have write permission for the jre\bin folder, or place the Oracle Add-in basic .dll files in the jre\bin folder.

For more information see:
• “Accessing Oracle Add-in DLL Files” on page 40
• “Locating the Java Console” on page 39</td></tr>
<tr><td>Your Java console contains the line Could not find –Xrun library: micsupp.dll.</td><td>Check that you have micsupp.dll in your system folder (WINNT\system32 or windows\system).</td></tr>
</table>

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 Oracle Application, contact Mercury Interactive Customer Support.

Checking Java Environment Settings

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

Note: The short path (also known as the 8.3 DOS name) of the Oracle 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

➤ Set the _JAVA_OPTIONS environment variable (Sun) 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

Oracle JInitiator 1.1.x

➤ The _classload_hook environment variable should be set to micsupp.

Oracle JInitiator 1.3.1.x

➤ Set the _JAVA_OPTIONS environment variable 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** is 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:

```
-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
```

Locating the Java Console

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

If your application runs in Oracle JInitiator 1.3 or higher:

➤ Right-click the JInitiator icon in the taskbar tray and click **Show Console**.

If your application runs in Oracle JInitiator 1.1.x:

➤ If you do not see the JInitiator icon in the taskbar tray, click **Programs** > **JInitiator Control Panel** in the Start menu. In the Basic tab, select **Show Java console** and click **Apply**. Restart your JInitiator application.

If your application runs in JDK 1.4 Plug-in:

➤ Right-click the Java Plug-in icon in the taskbar tray and click **Open Console**.

➤ If you do not see the Java Plug-in icon in the taskbar tray, click **Settings** > **Control Panel** in the Start menu. Double-click the **Java Plug-in** icon. In the Basic tab, select **Show Java in System Tray** and click **Apply**. Restart the browser.

Accessing Oracle Add-in DLL Files

For the Oracle 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 Oracle Add-in installs these files in the **jre\bin** folder of your Java environment when your Oracle Application starts.

However, if you do not have write permission in the **jre\bin** folder, the Oracle 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 Oracle Add-in). If the files and the folder are accessible when you install the latest version of WinRunner with Java support, the Oracle 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 Oracle Add-in installation version.

Warning: The file S:\JAVA\JDK1.4.0\jre\bin\mic_if2c_aqt.dll does not match your current Oracle 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 the Oracle Add-in without Multi-JDK Support (Advanced)

The Oracle 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 Oracle Add-in support classes according to the JInitiator or JDK version used. If, for some reason, this mechanism does not work, you can still use the Oracle Add-in if you manually configure the Java environment.

Java 2 (JInitiator 1.3.1.x, Sun Plug-in 1.4.1)

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 environment variable).

Next, you should change the –Xbootclasspath setting to list the correct patches folder according to the JVM version you are using. If you do not know which version you are using (but it is Java 2 VM), the exact JVM version number should be displayed in your Java Console. For information on displaying the Java Console, see “Locating the Java Console” on page 39.

If you are using JInitiator version 1.3.1.11 or earlier, add the **ora_1.3.1** folder to the Xbootclasspath. If you are using JInitiator version 1.3.1.12 or later, add the **1.3.1_06** folder to the Xbootclasspath. If you are using JDK, add the **1.4.1** folder to the Xbootclasspath.

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_OPTIONS environment variable. 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 JInitiator 1.3.1.14, 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\1.3.1_06;
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 **bin** folder of the Plug-in or JInitiator you are using.

Java 1 (JInitiator 1.1.x)

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 manually copy the classes from the correct patches folder to the JInitiator 1.1.x classes folder. For JInitiator versions 1.1.7.x, copy the classes from the **ora_1.1.7** folder. For JInitiator versions 1.1.8.x, copy the classes from the **ora_1.1.8** folder.

Note: Typically, the JInitiator version number is part of the installation path. If you do not know the version number, and do not know the installation path, check the Java Console for version information. For information on locating the Java Console, see "Locating the Java Console" on page 39.

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

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 **bin** folder of the JInitiator 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 Oracle Add-in without Multi-JDK Support (Advanced)" on page 41, to enable you to use the Oracle Add-in.

Index

A

accessing an object field 20, 21, 22
Add-in Manager 3

C

configuring the way WinRunner learns 10

E

edit objects, activating 8
edit_activate function 8

F

firing Java events 10, 31

G

GUI Map Editor 2
GUI Spy 24, 27

I

invoking
- Java method 18
- Java method from a returned object 22
- static Java method 19

J

Java
- console 39

Java bean properties, setting the value of 7
Java events, simulating 10, 31
Java method
- invoking 18
- invoking from a returned object 22
- invoking static 19

Java Method wizard 24, 28
Java objects
- working with 22

Java pop-up menu, selecting an item from 9
java_activate_method function 18
- invoking a Java method 27, 30
- viewing the methods for an object 28

java_activate_static function 19
java_fire_event function 10, 31
java_get_field function 20
java_get_static function 21
java_set_field function 7, 21
java_set_static function 22
jco_create function 22
jco_free function 23
jco_free_all function 23

M

method_wizard statement 28

O

obj_mouse_click statement 6
obj_set_info function 7
object field, accessing 20, 21, 22
object methods
- viewing 24

Oracle Add-in
- accessing .dll files 40
- checking environment settings 37
- disabling multi-JDK support 43

Oracle Add-in (*continued*)
running without multi-JDK support 41
starting 3

P

popup_select_item function 9

S

set_aut_var function 10
COLUMN_NUMBER variable 10
EDIT_REPLAY_MODE variable 12
EXCLUDE_CONTROL_CHARS variable 12
MAX_COLUMN_GAP variable 11
MAX_LINE_DEVIATION variable 11
MAX_ROW_GAP variable 11
MAX_TEXT_DISTANCE variable 13
RECORD_BY_NUM variable 13
RECORD_WIN_OPS variable 13
SKIP_ON_LEARN variable 14
SOFTKEYS_REC variable 14
TREEVIEW_PATH_SEPARATOR variable 14
USE_LOW_LEVEL_EVENTS variable 14
setting the value of a Java bean property 7
simulating Java events 10, 31
static Java method
invoking 19

T

troubleshooting, testing Java objects 35
TSL functions
for standard Java objects 5

V

variables, for set_aut_var 12

W

win_mouse_click statement 6

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