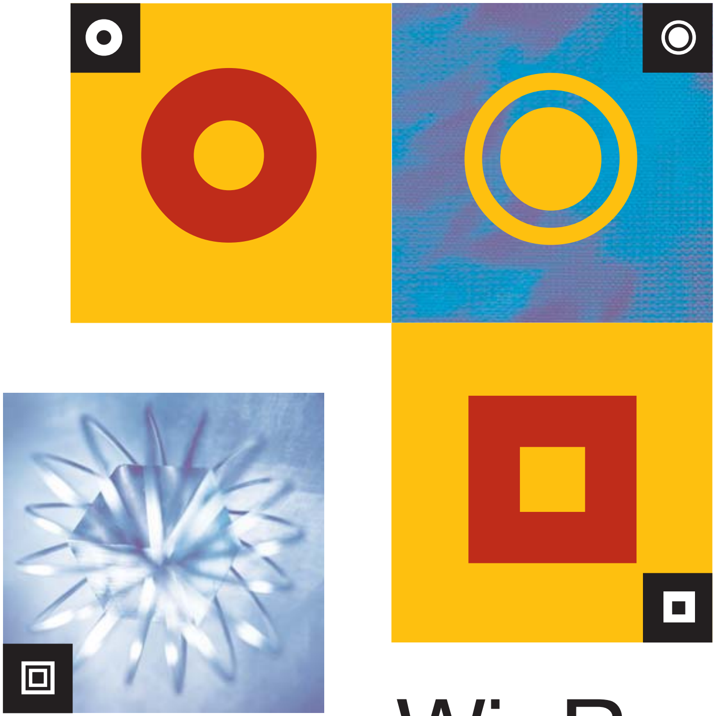
Online Guide



WinRunner® 7.5

Testing Java Applications and Applets

Table of Contents

Books Online
Find
Find Again
Help
Top of Chapter
Back

Introduction

Welcome to WinRunner with add-in support for Java. 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

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 test cross-platform Java applets and applications.

To create a test for a Java applet or application, use WinRunner to record the operations you perform on the applet or applications. As you click on 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.
- 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.

Books Online
Find
Find Again
Help
Top of Chapter
Back

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 GUI Map Configuration**. A custom java object can be configured as a push button, check button, static text, text field, etc. 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*.

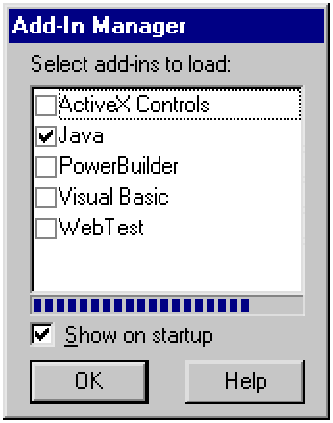
Activating the Java Add-in

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 *Java Add-in Installation Guide*.

Note: The RapidTest Script wizard option is not available when the Java Add-in is loaded. For more information about the RapidTest Script wizard, refer to the *WinRunner User's Guide*.

To activate the Java Add-in:

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


Add-In Manager

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.

Books Online
Find
Find Again
Help
Top of Chapter
Back

Note:

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

1. Start WinRunner.
2. In **Settings** > **General Options** > **Environment tab**, check **Show Add-in Manager dialog for ___ seconds** and fill in a comfortable amount of time in seconds. (The default value is 10 seconds.)
3. Click **OK**.
4. Close WinRunner. A WinRunner Message dialog opens asking “Would you like to save changes made in the configuration?” Click **Yes**.
5. Repeat the procedure described in “To activate the Java Add-in,” on **page 8**.

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

Books Online
Find
Find Again
Help
Top of Chapter
Back

Testing Standard Java Objects

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

This chapter describes:

Books Online
Find
Find Again
Help
Top of Chapter
Back

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 in Netscape, Internet Explorer, AppletViewer, or a standalone Java application.

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 *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 Java console and a Java plug-in are open simultaneously, the Java add-in support will not function properly as this scenario results in two virtual machines and WinRunner cannot distinguish between them.If this happens, close the browser and console and then re-open 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 in Netscape, Internet Explorer, AppletViewer or a standalone Java application, 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, etc. by using the Java Custom Objects 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 property.
- Activate a specified Java edit field.
- Find the dimensions and coordinates of list and tree objects 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 applets and 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*.

Books Online
Find
Find Again
Help
Top of Chapter
Back

Setting the Value of a Java Bean Property

You can set the value of a Java bean property with the **obj_set_info** function. This function works on all properties that have a set method. The 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 property name. Therefore a Java bean property called *MyProp* becomes *my_prop* in the TSL statement.

For example, for a property called MyProp, which has 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 objects 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 selected item must be the last item in a menu tree, for example: **popup_select_item ("Copy");** is not legal, while **popup_select_item ("MyEdit;Copy");** is legal.

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);
```

Books Online
Find
Find Again
Help
Top of Chapter
Back

Configuring How WinRunner Learns Object Descriptions and Runs Tests

You can configure how WinRunner learns descriptions of objects, records tests, and runs tests on a Java applet or application with the **set_aut_var** function. The function has the following syntax:

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

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.

	“E”- generates a FocusGained event at the beginning of function execution. (Oracle only, EWT toolkit) **Default value: “PTR”** Note that the default action sends a triple event to the edit field (KeyPressed-KeyTyped-KeyReleased).
EVENT_MODEL	Sets the event model that will be used to send events to the AUT objects. Use one of the following values: “NEW”- for applications written in the new event model. “OLD”- for applications written in the old event model. “DEFAULT”- Uses the OLD event model for AWT objects and NEW for all other toolkit objects. **Default value: "DEFAULT"**
MAX_TEXT_DISTANCE	Sets the maximum distance in pixels, to look for attached text. **Default value: 100**

Books Online
Find
Find Again
Help
Top of Chapter
Back

REPLAY_INTERVAL	Sets the processing time in milliseconds between the execution of two functions. **Default value: 200**
RETRY_DELAY	Sets the maximum time in milliseconds to wait before retrying to execute a command. **Default value: 1000**
SKIP_ON_LEARN	Controls how WinRunner learns a window. Mercury Interactive classes listed in the variable are ignored. May contain a list of Mercury Interactive classes, separated by spaces. By default, only non-“object" objects are learned. **Default value: "object"**
TABLE_RECORD_MODE	Sets the record mode for a table object (CS or ANALOG). “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”**

COLUMN_NUMBER	Minimum number of columns for a table to be considered a table object. Otherwise the edit fields are treated as separate objects. (Oracle only) **Default value: 2**
MAX_COLUMN_GAP	The maximum number of pixels between objects in a table to be considered a column. (Oracle only) **Default value: 12**
MAX_LINE_DEVIATION	The maximum number of pixels between objects to be considered to be on a single line. (Oracle only) **Default value: 8**
MAX_LIST_COLUMNS	The maximum number of columns in an Oracle LOV object to be considered a list. A larger number constitutes a table. (Oracle only) **Default value: 99**
MAX_ROW_GAP	The maximum number of pixels between objects to be considered one table row. (Oracle only) **Default value: 12**

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 control, tree, or a combination separated by a space. If one of these objects has been 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.)
USE_LOW_LEVEL_EVENTS	Controls whether WinRunner simulates user input by Java events or by the mouse and keyboard drivers. "All": indicates that WinRunner simulates all mouse clicks and keyboard strokes for all types of Java objects by the mouse and keyboard drivers. Class names separated by a space indicates 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.

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 REPLAY_MODE_EDIT contains "S".
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.

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 35. 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.

Books Online
Find
Find Again
Help
Top of Chapter
Back

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 35.

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_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* **);**

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.

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 35.

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, 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 *value* parameter is an output variable that holds the value from the retrieved field.

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_get_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 35.

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

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 35.

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

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, value* **);**

Books Online
Find
Find Again
Help
Top of Chapter
Back

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

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_get_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 35.

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

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);
```

Books Online
Find
Find Again
Help
Top of Chapter
Back

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 35.

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

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 classloader 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_name* **);**
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.

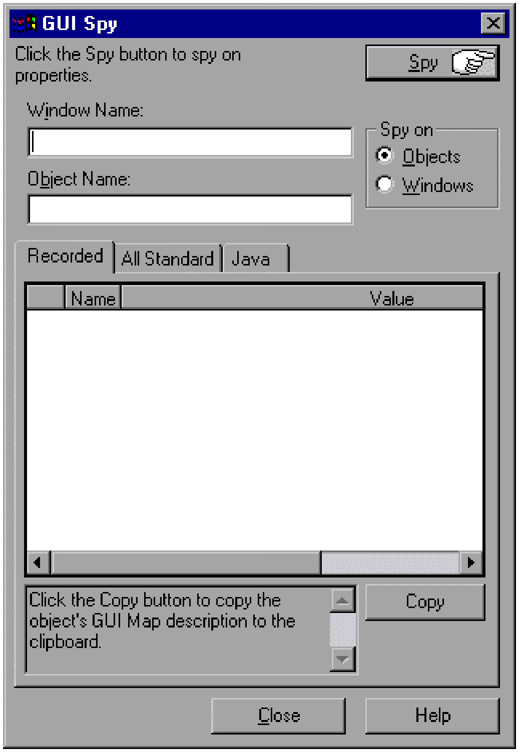
Note: As with any other GUI object, you can view all properties or just the recorded properties of a Java object in the **All Standard** or **Recorded** tabs of the GUI Spy. For more information on these elements of the GUI Spy, refer to the *WinRunner User's Guide*.

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.

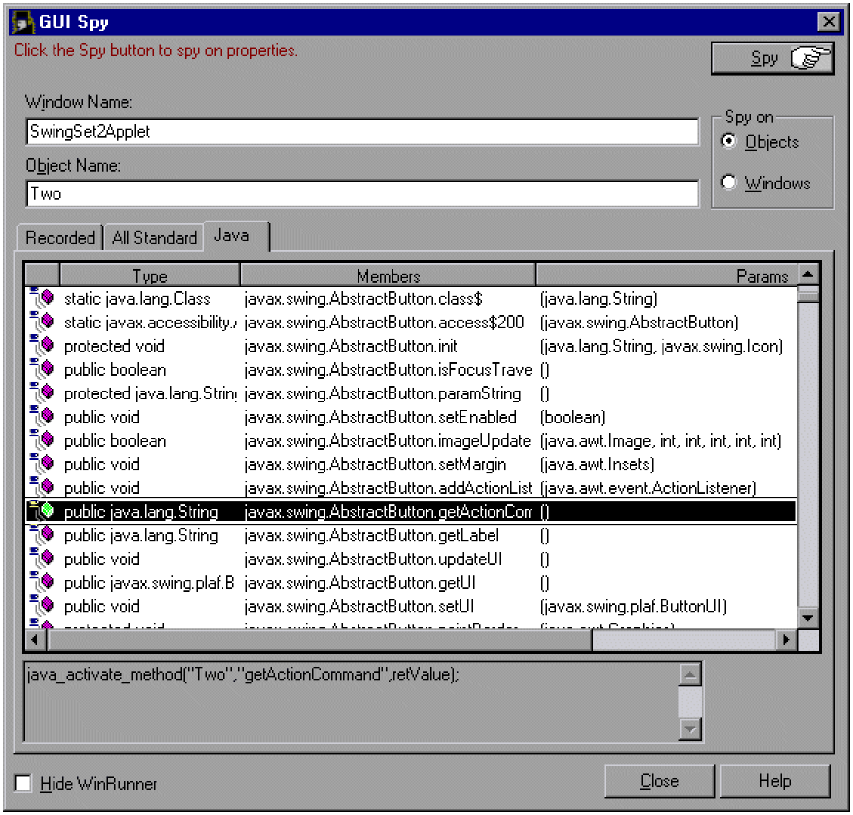
Books Online
Find
Find Again
Help
Top of Chapter
Back

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


GUI Spy
Click the Spy button to spy on properties.
Spy
Window Name:
Object Name:
Spy on
Objects
Windows
Recorded
All Standard
Java
Name
Value
Click the Copy button to copy the object's GUI Map description to the clipboard.
Copy
Close
Help

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.getActionCom	()
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 37**.

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,* 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`, etc. Replace the input parameters in the statement with the parameter values you want to send to the method.

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:

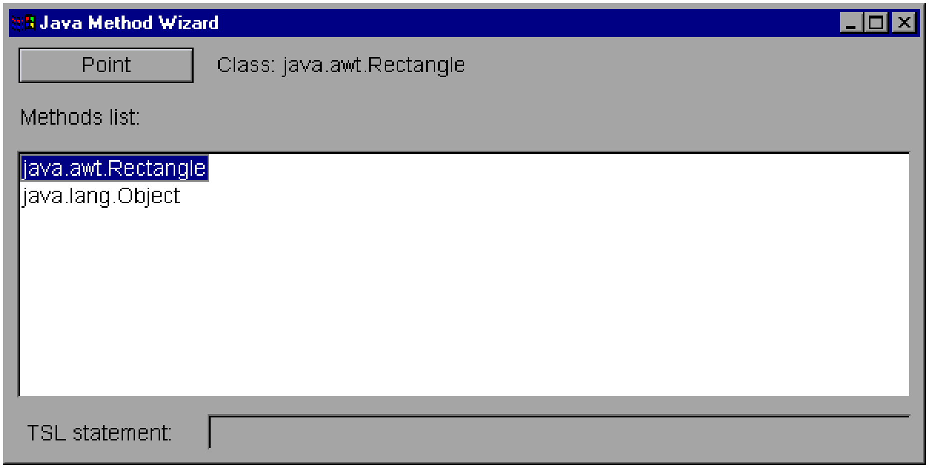
1 Open the Java application or applet that contains the object for which you want to view the methods.

2 Enter a method_wizard statement to activate the 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 Choose **Run > 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.

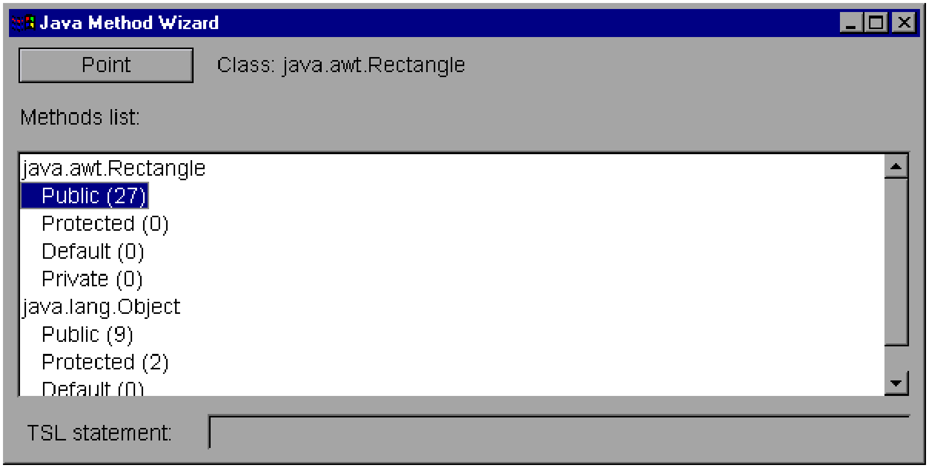


Class: java.awt.Rectangle

Methods list:

java.awt.Rectangle
java.lang.Object

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



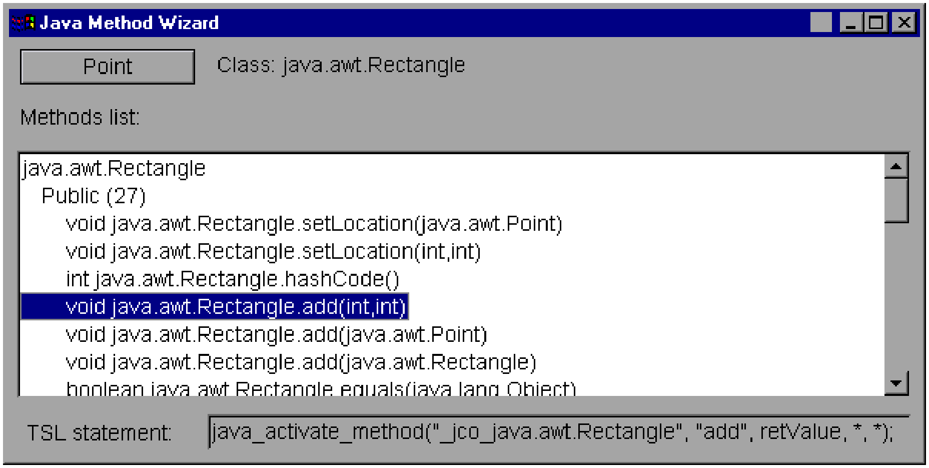
Class: java.awt.Rectangle

Methods list:

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

TSL statement:

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



```
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)
```

```
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 41.

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,* 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".

Books Online
Find
Find Again
Help
Top of Chapter
Back

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:

Books Online
Find
Find Again
Help
Top of Chapter
Back

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 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)"
}
```

Books Online
Find
Find Again
Help
Top of Chapter
Back

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 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 Custom Object Wizard

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

Note: The GUI Map Configuration tool does not support configuring Java objects. The Custom Object wizard (Java GUI Map Configuration option) 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 GUI Map Configuration option. For more information about the GUI Map Configuration tool, refer to the *WinRunner User's Guide*.

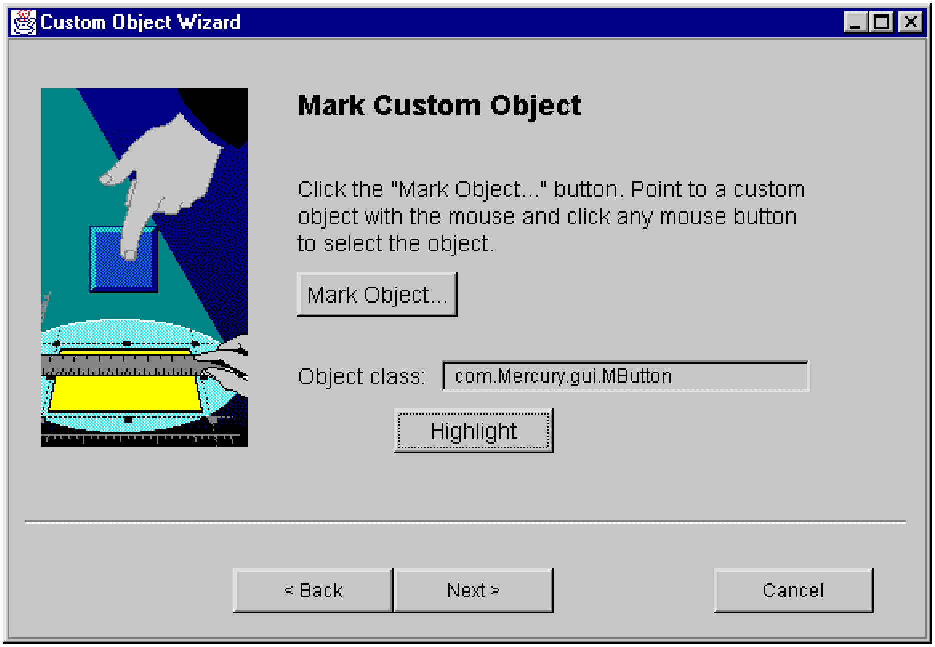
To configure a Java object using the Custom Object wizard:

1 Open your Java application containing custom Java objects.

2 Open a new test in WinRunner.

3 Choose **Tools > Java GUI Map Configuration**. 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

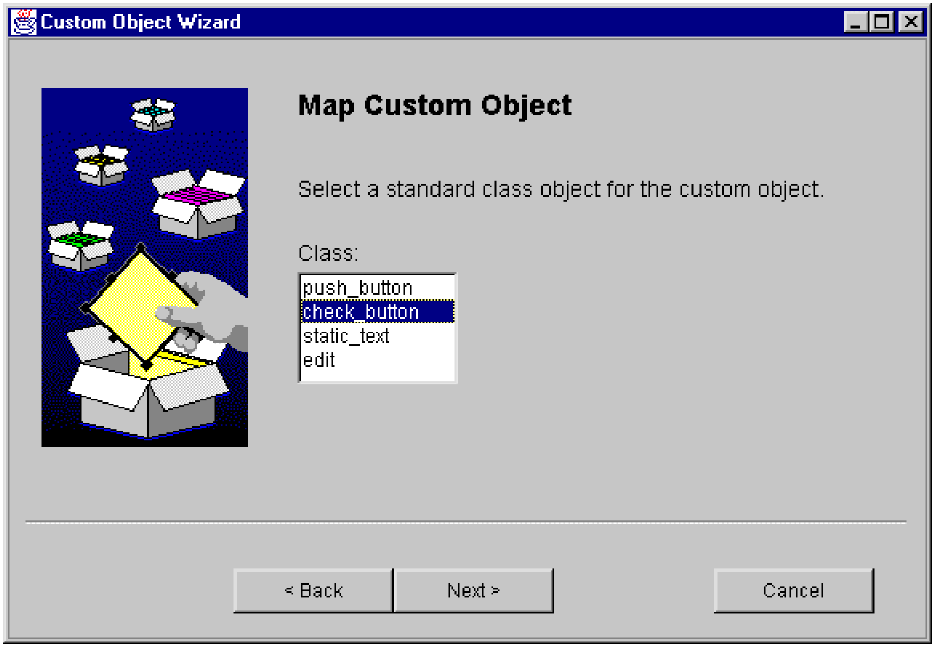
Click the "Mark Object..." button. Point to a custom object with the mouse and click any mouse button to select the object.

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**.



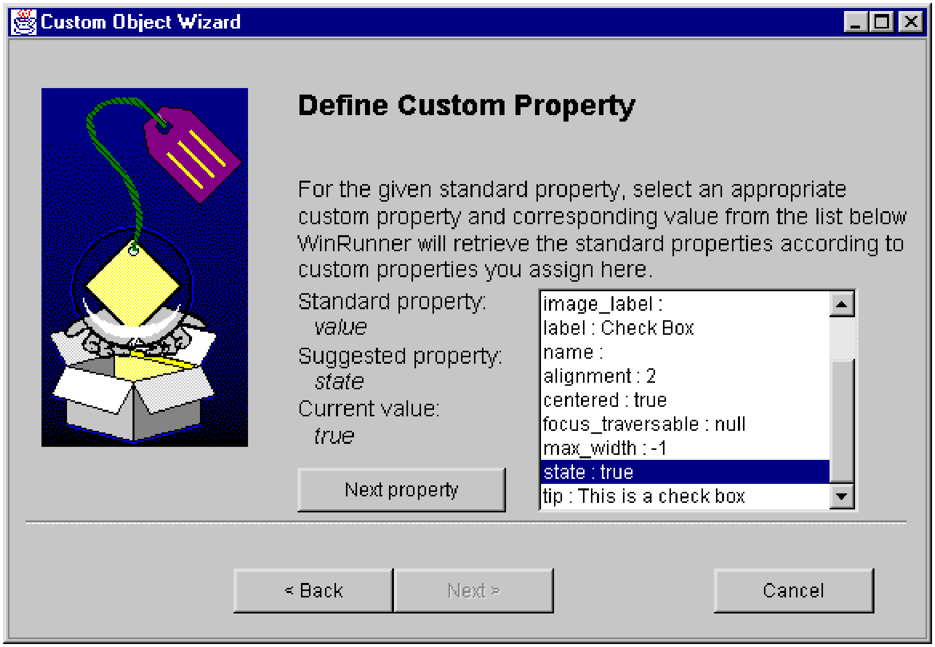
Map Custom Object

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.



Custom Object Wizard

Define Custom Property

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.

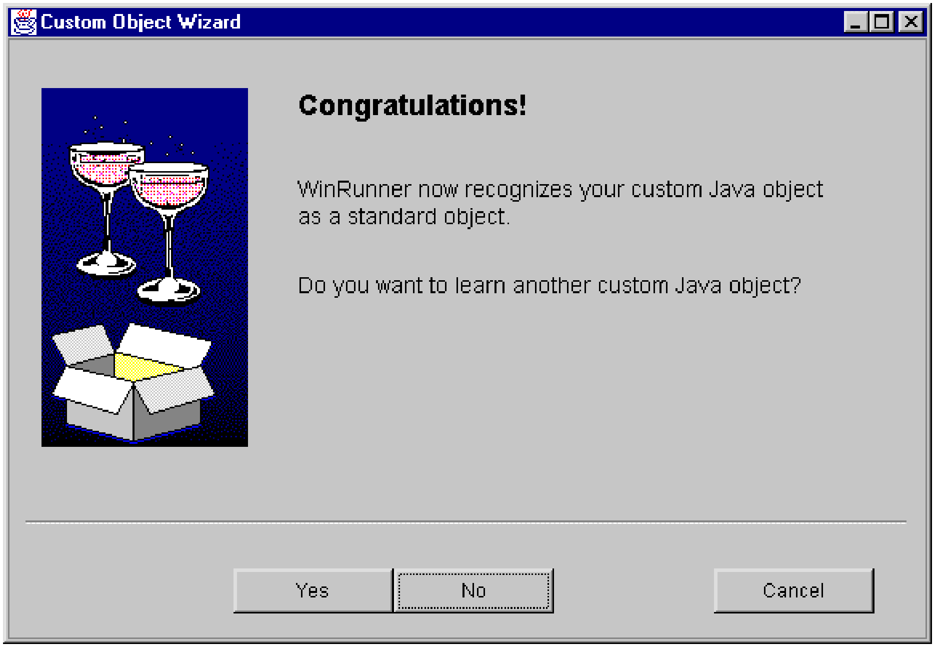
Standard property:
value
Suggested property:
state
Current value:
true

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 to learn another custom Java object, click **Yes.** The wizard returns to the Mark Custom Object screen. Repeat steps 4-8 for each custom object you want to configure. If you are finished configuring custom Java options, click **No**.

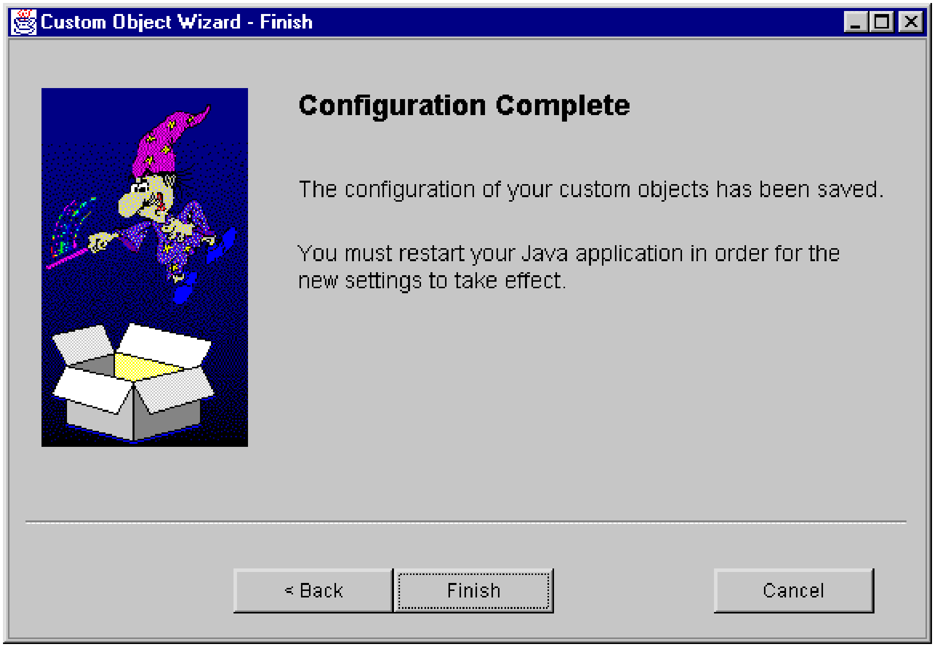


Congratulations!

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

Do you want to learn another custom Java object?

10 The Finish screen opens. Click the **Finish** button 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.

11 Close and reopen your Java application or applet in order 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 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 no longer want to use your custom Java configurations, delete the custom Java objects in the GUI Map and delete the *customization.properties* file. Then restart your Java application or applet.

Using Java Direct Call (JDC)

This chapter explains how to use the Java Direct Call (JDC) Mechanism to call Java functions from TSL scripts.

This chapter describes

Note: You can use the **java_activate_method** function to call Java functions from TSL scripts. For all new scripts, it is recommended to use the **java_activate_method** function.

About Java Direct Call Mechanism

JDC enables you to specify a Java function to execute from the TSL script. This user-defined Java function may contain any standard Java code.

Unlike the **java_activate_method** function described in Chapter 2, **Testing Standard Java Objects**, JDC functions work on Java applications that do not have any Java User Interface object bound to them. These functions can retrieve string parameters provided in TSL.

Using the JDC Mechanism

You can use standard Java code to call a Java function from a TSL script.

To enable the JDC mechanism:

1 Create a Java class listing and implementing all methods to be called from the TSL script.

All methods must follow the prototype convention:

static int jdc_< *func_name* **>(** *String* **[ ]** *params* **);**

func_name	a name of the function
params	an array of parameters passed from WinRunner

2 Register JDC class(es) with WinRunner by using the following TSL statement:

```
set_aut_var("JDC_CLASSES", "foo.bar.class1;foo.bar.class2");
```

Note: You can create as many JDC classes as required. JDC classes must be found in the CLASSPATH.

Books Online
Find
Find Again
Help
Top of Chapter
Back

3 Provide an "extern" definition for the JDC function in TSL.

For example, if you have defined a JDC function in your Java class as:

```
static int jdc_print_strings(String[] param);
```

make the following declaration in TSL:

```
extern int jdc_print_strings(in string p1, in string p2);
```

When calling Java, param[0] will contain p1 and param[1] will contain p2.

4 Call the JDC function from TSL:

```
jdc_print_strings("str1", "str2");
```

Preparing a TSL Script for Use with JDC

The **jdc_aut_connect** function must appear in your script prior to any jdc operation. This function establishes a connection between WinRunner and Java applications and must be executed at least once. Your java application or applet must be loaded before running this function. You use this function as follows:

jdc_aut_connect (*in_timeout* **);**

timeout	time (in seconds) that is added to the regular **timeout for checkpoints and CS statements** (**Settings > General Options > Run Tab**), resulting in the maximum interval before the next statement is executed.

Using JDC: An Example

The example below shows how a user prepares the Java source file with definitions for two Java functions. Then the user registers the Java functions with WinRunner so that he can call the Java functions from the TSL script.

Preparing the Java Source File

The following sample Java source file defines 2 Java functions for later use in the TSL script.

```
// Sample File of JDC calling mechanism.

public class JdcExample {

/**
 This function will print the first parameter that it
receives to the console
 */
  public static int jdc_simple_call(String[] params) {
    String first_param = params[0];
    System.out.println("jdc_simple_call called: Got parameter: " +
     first_param);
    return 0;
  }
 /**
```

```
   This function will return the upper case version of the first
     parameter string in the second parameter.
  */
  public static int jdc_out_par_call(String[] params) {
    // Convert input parameters
    String in_par = params[0];
    String out_par = params[1];

    System.out.println("jdc_out_par_call called: Got parameter: " +
     in_par);
    out_par = in_par.toUpperCase();
    // Prepare output parameters
    params[1] = out_par;
    return 0;
  }
}
```

Registering JDC with WinRunner and calling Java functions in the TSL Script

The TSL script segment below shows how to define the “extern” declaration, to load and register the JDC classes defined in the Java source code, and then to call the Java functions.

```
# define "extern" declaration
extern int jdc_simple_call(in string str);
extern int jdc_out_par_call(in string str1, out string str2<256>);

# register JDC classes

# make sure the classes are in the classpath
set_aut_var("JDC_CLASSES", "JdcExample");

# connect to the AUT
rc=jdc_aut_connect(10);
if (rc != E_OK)
    pause ("Error: Couldn’t connect to AUT\n Check that the AUT is loaded
!");
```

Books Online
Find
Find Again
Help
Top of Chapter
Back

```
# call JDC functions - this will print the parameter to the Java console
r1=jdc_simple_call("my string");
r2=jdc_simple_call(256);

# this will put the Upper Case form of the parameter in the UpperCaseParam var.
r3=jdc_out_par_call("my string", UpperCaseParam);

pause(UpperCaseParam);
```

Note: You must start WinRunner with Java Add-in support before you start your Java application or applet. Otherwise, WinRunner may not record and run your test script properly.

Troubleshooting Java Add-in Recording Problems

Once you complete the Java Add-in installation process, you should be able to successfully record from Netscape, Internet Explorer, AppletViewer, or a standalone Java application. This chapter offers some guidance if you have difficulty recording tests on Java Applets or Applications.

This chapter describes:

Note: For more troubleshooting information, as well as the latest known issues and limitations, refer to the Java Add-in *Read Me* file.

Handling General Problems Testing Applets or Applications

To analyze problems testing Java applets or applications, try the relevant solutions from the following list:

- View the Java console and confirm that the following confirmation messages appear: “Loading Mercury Support (version 7.50.700.0)”.
- Confirm that you are able to test your applet with the AppletViewer.
- View the *install.log* file located in the *<WinRunner Installation Folder>\dat* folder.

- If you encounter any problem working with:
  - JDK 1.1.x, verify that you have the following environment variable: _CLASSLOAD_HOOK=micsupp
  - JDK 1.2 or higher, verify that you have the following environment variable: _JAVA_OPTIONS=-xrunmicsupp -xbootclasspath/a:*path*;*path*\mic.jar
  - IBM JDK, verify that you have the following environment variable: IBM_JAVA_OPTIONS=-xrunmicsupp -xbootclasspath/a:*path*;*path*\mic.jar

Books Online
Find
Find Again
Help
Top of Chapter
Back

Notes: If the environment variable is missing, add it as shown above.

"*Path*" is the DOS (8.3 notation) short path for the Java Add-in folder. For example:
C:\PROGRA~1\COMMON~1\MERCUR~1\SHARED~1\JAVAAD~1\classes

The Java Add-in setup sets the environment variable "mic_classes" folder to the short path for the Java Add-in folder. You can examine the value of this variable by typing the command `echo %mic_classes%` in a command prompt window.

- If the Java Support Switching Tool is disabled, click the icon to enable Mercury Java support.
- If the Java Support Switching Tool does not appear in the task bar tray, run the Java Add-in Switching Tool (available from the WinRunner Start menu program group). For more information, refer to the *Java Add-in Read Me* file.

Note: If any of the above checks are not successful, close WinRunner and all browsers and re-install the Java Add-in. If you still have problems testing applets from Netscape or Internet Explorer, please contact Mercury Support.

Handling Java Add-in Problems

If you are having specific problems installing or recording, try the relevant solutions from the following list:

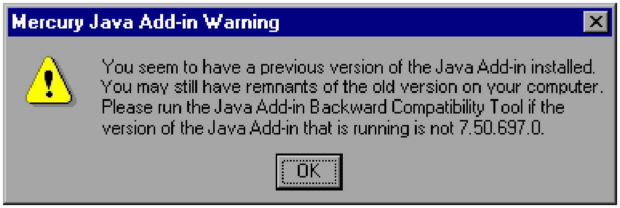
- If the Java console and a Java plug-in are open simultaneously, the Java add-in support will not function properly as this scenario results in two virtual machines and WinRunner cannot distinguish between them.

  Close the browser and Java console, then re-open the browser and try again.

- If you were using a previous version of the Java Add-in, and you added command line parameters, you may experience some problems.

  Remove the command line parameters that you added.

- If you are upgrading from a previous version of the WinRunner Java Add-in (from a version prior to 7.5), you may still have remnants of the old version on your computer. If this is the case, you will receive a warning when you try to run a supported Java application with Mercury Java support enabled.



Mercury Java Add-in Warning

You seem to have a previous version of the Java Add-in installed. You may still have remnants of the old version on your computer. Please run the Java Add-in Backward Compatibility Tool if the version of the Java Add-in that is running is not 7.50.697.0.

Run the Java Add-in Backward Compatibility Tool (available from the WinRunner Start menu program group).

Index

Books Online
Find
Find Again
Help
Top of Chapter
Back

A B C D E F G H I J K L M N O P Q R S T U V W X Y

WinRunner - Testing Java Applications and Applets, Version 7.5

Mercury Interactive Corporation
1325 Borregas Avenue
Sunnyvale, CA 94089
Tel. (408) 822-5200 (800) TEST-911
Fax. (408) 822-5300

If you have any comments or suggestions regarding this document, please send them via e-mail to documentation@mercury.co.il.

WRJAVAUG7.5/01