Technote 1162

Introduction to MRJ Scripting with AppleScript for Java

CONTENTS

With the release of MRJ 2.1, AppleScript support in Java applets and applications is now a reality. This Technote covers the technical information you will need to take full advantage of AppleScript in your Java application.

Apple Applet Runner and the MRJShellLib (inside of MRJLib) support the Macintosh Open Scripting Architecture. As a result, the Apple Applet Runner, the applets it runs, and applications created with JBindery can be scripted by AppleScript or any other OSA scripting language. The only additional work required (for scripting of java-based applications) is the inclusion of the scripting resources `'aete'` and `'scsz'`. No other specific scripting support is required from the Java applet or application. This enables MRJ users to download Java applets or applications from the Internet and use them as scriptable components on the Macintosh.

This document provides a brief overview of MRJ Scripting. It assumes that you are familiar with AppleScript and the OSA Architecture. As more information about MRJ Scripting becomes available, it will be posted on the MRJ Developer Page.

How MRJ Scripting Works

MRJ Scripting works by exporting public Java class, property, and method names as OSA scripting terminology so that they are visible to scripters. When a script runs, MRJ scripting translates the Apple Event data types and events into Java data types and method calls. A scripter can thus set and get data in property fields of Java objects, and invoke Java methods on those objects.

The OSA scripting terminology is generated dynamically by the Applet Runner or Java application. When you compile a script that targets the Apple Applet Runner, the Script Editor (or any other OSA development environment) fetches its dynamic terminology. That terminology includes AWT Component objects for each open applet. For example, the sample applet "Lightweight Gauge" has a formal applet name of `ExampleApplet`; if you open the Applet Runner dictionary while Lightweight Gauge is running, you will see that it includes suites that look like this:

```
ExampleApplet:
init: public void

ExampleApplet.init()

init reference
   Class Example Applet:
   Properties:
           <Inheritance> Applet [r/o]
                                                0Gauge:
                                                  Class Gauge:
    Properties:

<Inheritance> Component [r/o]
         total Amount integer [r/o]  -- public int Gauge.getTotalAmount( )
         preferred Size point  [r/o] -- public java.awt.Dimension
Gauge.getPreferredSize()
         minimum Size point  [r/o] -- public java.awt.Dimension
Gauge.getMinimumSize()
         current Amount  integer -- public int Gauge.getCurrentAmount()
&
                                                      -- public void
Gauge.setCurrentAmount(int)
```

Because the `ExampleApplet` is running, the Applet Runner now exposes a `Gauge` class that inherits its properties from the Component class, a read-only property `totalAmount`, and a writable property `currentAmount`.

Methods and properties of Java objects are exposed to the scripter in this manner. Occasionally some Java object, method, or property names will conflict with AppleScript reserved word namespace, or be illegal in AppleScript (for example, an item that contains periods like user.name). When this happens, the Java names will be published enclosed in AppleScript's literal quotation marks, the vertical bars (|). For example, a Java Applet parameter named "columns" can be written as `|columns|` and will not conflict with the AppleScript terminology "columns". Notice that identifiers inside vertical bars are case sensitive, whereas normal AppleScript terminology is not.

In an applet that uses AWT, the entire AWT component hierarchy is exported for scripting, so every function of the applet that can be driven by the user interface can also be driven by scripting. To script an applet, you do not have to know about its internal methods; you can script its buttons, menus, and display objects directly. This is different from normal Macintosh application scripting, where you normally script the semantic objects of the application. The principal advantage to the MRJ's scripting method is that all aspects of the applet's operation are scriptable, without any special assistance from the applet. The principal disadvantage is that internal or semantic information stored in the applet can only be accessed by pulling the "puppet strings" of the user interface by scripting the AWT components. Thus, developers should include semantic objects as properties of the top-level Component in their Java applets or applications. Then you can script Java applets or applications in the preferred way of Macintosh scripting.

Important:
Obfuscated java classes are unscriptable for all practical purposes.

Back to top

Scripting the Applet Runner or Java Application

The Applet Runner is a scriptable application that supports the Required and Standard suites of events, as well as a small number of custom events. Java applications should also have a dictionary that supports these events. The supported events are:

- **print** - prints an AWT component.
- **exists** - tests if a Java object exists
- **check... belongs to** - checks to see if a Java object belongs to a certain class
- **apply to** - calls a method by name. This can be useful if the terminology is not present and as a result, you cannot call it the normal way.
- **save object** - serialize the Java object into a .jar file
- **load object** - load an object from a .jar file
- **start tool** - the tool can be any jar file with visible bean inside. Since it is a tool with Java classes you can script it the usual manner. The developer can build up useful AppleScript tools to do things such as display an `AEList` as a tree.
- **type .. keystrokes** - a low-level event to deliver keystrokes to Java windows
- **click** - a low-level event to simulate mouse clicks on AWT components. This can also be used to do some UI scripting such as clicking on a menu item by name

See the Applet Runner dictionary for the full parameters and results of these events.

Back to top

Applet Runner Application Properties

Like most scriptable applications, the Applet Runner application or Java applications support global properties that control its operation. All system properties of the Java engine are exposed as application properties of the Applet Runner or Java application. Remember that because most system properties are illegal as AppleScript identifiers, they are enclosed in vertical bars and are case sensitive:

```
tell
application "Apple Applet Runner"
    get its |user.name|
end tell

There are two new system properties that control specific aspects of MRJ Scripting:

tell
application "Apple Applet Runner"
    set its |macos.scripting.debug| to true
    set its |macos.menu.contextual.disable| to true
end tell
```

If a method returns a Java object, it is normally returned as the opaque class of the Java object. With `|macos.scripting.debug|` set to **true**, a Java object will be translated into a text string using the Java method `toString()`. While this may not be useful from a scripting perspective, it can be very useful for debugging purposes.

MRJ Scripting provides automatic contextual menu support for Java `TextComponent` objects. You can disable this capability by setting `|macos.menu.contextual.disable|` to **true** or turning it off from the help menu.

Back to top

Java Objects as AppleScript Objects

When an applet is opened in the Applet Runner, it is available as an implicit top-level element of the application (much like desktop files are implicit top-level objects in the Finder). To experiment with this, open the "Lightweight Gauge" applet in the Applet Runner and execute the following script:

```
tell
application "Apple Applet Runner"
     restart ExampleApplet 1
end tell
```

The Java objects that make up this applet are contained by the `ExampleApplet` object.

Important:
In this version of the Applet Runner, the containment relationships of Java AWT components are exposed through the components property of the Container object. This object returns a list of Java components enclosed by that container. There is no formal "element" relationship between a container and its contents, but the Applet Runner accepts element-style references and translates them to the correct Java object references. This means that the normal AppleScript commands that act on elements (`count`, `each`, `whose`, etc.) do not operate on items contained by a container object even though you can perform some of those operations on the components property of that container.

(Note that in the applet runner, the window is a container of the Applet and is not the Applet object itself. The Applet Runner's Window class is useful only for manipulating the windows the Applet Runner displays. You can also address the applet directly.)

Discovering the correct name of a component through trial and error can be difficult. MRJ Scripting assists you in determining object specifiers (when the `|macos.scripting.debug|` property is set to true). If you turn on Balloon Help, you may point to any object, and the balloon will give you a useful object specifier for that object. The object specifier is also output to the Java console.

Following AppleScript conventions, the contents of containers are referred to using 1-based indices, even though the standard for Java is zero-based indices. So the first component in a container is component 1, not component 0.

Most properties of the Java object are available as properties of the AppleScript object, and the AppleScript `get` and `set`commands can be used to examine and change the values. For example, using the Lightweight Gauge applet as an example, the value for the second bar can be accessed with:

```
get current Amount of Gauge 2 of PrettyPanel 1 of Double Buffer Panel 1 of Example Applet 1
```

It can be set accordingly with the AppleScript`set` command if the property is not marked as [r/o] (read only) in the dictionary.

Back to top

Java Methods as AppleScript Commands

Most method calls on a Java object are available as AppleScript commands. In the dictionary displayed by the Script Editor, each Java class is listed in its own suite with an AppleScript class representing the Java class and a list of commands in that suite that correspond to the methods.

Important:
Note that AppleScript is lax about associating commands with objects. A script that sends a command to an object that does not support that method **will** compile, although it will get a runtime error.

The target or direct object of a command must be the Java object that supports the corresponding method. You can use the syntax of any of the following AppleScript examples to send a command to a target object:

```
restart Example
Applet 1
restart of Example Applet 1
tell Example Applet 1 to restart
tell Example Applet 1
    restart
end tell
```

Because AppleScript supports named parameters but not ordered parameters, and Java supports ordered parameters but typically not named parameter information via Java reflection, the parameters are usually passed in the parameter-named parameters. This is an ordered list containing the values that would appear between the parenthesis in the Java method invocation. Remember, the parameters must be supplied in the order they are expected by the Java method; for your convenience, this is listed in the comment line of the command's dictionary entry. For example, an invocation of the `replaceRange` command would look like:

```
replace Range Text Area 1 of MRJ Test 1 parameters {"testing",0,20}
```

(Because parameter lists are passed directly to Java methods without interpretation, index parameters are zero-based, not 1-based).

Java classes that implement the BeanInfo interface provides additional information for scripting. MRJ scripting can then look for Java Parameter Descripter objects to provide named parameters in the dictionary, and you can use these named parameters directly in AppleScript. As more JavaBeans are created, this will become more common, but currently very little Java code provides this information.

Most methods take scalar parameters (discrete types such as `ints`, `longs`, `Booleans`, etc.) and return scalar results. However, the Java objects `String`, `Rectangle`, `Point`, `Dimension`, and `Color` are considered to be scalar for this purpose; they are automatically translated between Java and AppleScript formats.

Back to top

Scripting Java Applets

For applets, the applet tag is provided to AppleScript as a property of the Applet object, even though it is not really a field of the Java Applet class. Furthermore, you can get and set the Applet tag to change runtime parameters. Setting the Applet tag will result in a relaunch of the Applet with the new Applet tag. Here is an example of changing the `BarChart` applet to the vertical orientation:

```
set applet tag of Applet 1 to {c1:"10", c1_color:"blue", c1_label:"Q1",
c1_style:"striped", c2:"20", ¬ c2_color:"green",
c2_label:"Q2", c2_style:"solid", c3:"5", c3_color:"magenta",
c3_label:"Q3", ¬ c3_style:"striped", c4:"30",
c4_color:"yellow", c4_label:"Q4", c4_style:"solid",
|columns|:"4", ¬ orientation:"vertical",
scale:"5", |title|:"Performance"}
```

Back to top

Creating a Scriptable Java Application

To script a Java application, the Java application file needs to have `'aete'` and `'scsz'` resources in its resource fork. A sample `'aete'` resource, "MRJ Scripting aete," is provided in the MRJ 2.1 SDK in the folder "MRJ Scripting." A similar `'aete'` resource is already included with the Applet Runner. The `'aete'` resource includes a built-in suite and a first cut at the AWT terminologies. Terminology for other classes is generated dynamically using Java's reflection feature.

Automatically generated terminologies may not be very user friendly. Their comments are simply the Java object's name, and they expose most properties and methods to the scripter. You may want to use the automatically generated terminology as a starting point, then change the comments and delete the properties and methods that are not useful to the scripter.

Since the `'aete'` can be fairly large, it is a good idea to increase the memory partition of Script Editor, HyperCard, or whatever OSA development environment you are using. You may also need to increase the memory partition of the Java application or Applet Runner.

Back to top

Generating an 'aete' for a Java application

Scriptable applications must provide a **terminology resource** (stored in resource `'aete'` 0 in the resource fork) in order to be scriptable with AppleScript and other OSA scripting languages. This terminology resource associates the four-character event and class codes used in the application's source code with the English terminology used by the scripter. Normally, a developer compiles the terminology resource from a .r file with the Rez tool or edits it using a resource editor with an `'aete'` template (such as Resorcerer or ResEdit). For MRJ scripting, the four-character codes used for scripting are generated automatically and may not be modified by the developer, but you may want to change the terminology to improve readability.

The automatically generated terminology is a combination of a static basic terminology and dynamic terminology created by your scriptable Java application. The file "MRJ Scripting aete" includes the basic terminologies and the AWT component terminologies. MRJ scripting automatically generates additional terminologies for all components in open windows of Java applications; in this way MRJ generates all basic terminology needed to make an MRJ application scriptable. To include the basic resources in a Java application, use JBindery or ResEdit.

Once the basic `'aete'` resource is in your Java application, you must run it and touch every part of the application you want to be scriptable in order for it to generate the dynamic terminology. Open all the windows with components that you are interested in, so that AppleScript classes for those components will be included in the terminology.

If it is not practical to open all the windows manually, or if there are scriptable classes that are not AWT components, you can use scripting itself to ensure terminologies will be generated for those classes. In the basic dictionary there is a suite called the "Developer Suite," which contains commands and classes that designed to assist the developer, not the end user. In this suite, the `add terminologies for class` event lets you specify that a particular Java class should be included in the dynamic terminology.

Once you have touched (manually or by scripting) all the classes you wish to expose, the next step is to retrieve the dynamic terminologies. You do this with the same `'gdte'` event that the Script Editor uses to fetch the dynamic terminology from an application. Normally this event is hidden from the scripter so there is no terminology for it; in the Developer Suite we give it the event name `get terminologies` so you can write a script that sends it to your application.

In our terminology we also enhance the `get terminologies` event with an optional boolean parameter `object parameters`. Normally, the terminology generated excludes all events that have Java objects as parameters; if you want to include such events, you may add "with object parameters" when you call the `get terminologies` event.

The result of the `get terminologies` event is a large data object containing both the static and dynamic terminologies. If you want to edit this dynamic terminology to make it more usable, you need to save it to a file. There are a number of shareware scripting additions that will do the job. In the following example we use the `add resource` command from the GTQ Scripting Library (see http://www.scriptweb.com/osaxen/gtq_scripting_library.html):

```
tell
application "test.app"
    add terminology for class "com.acme.test.SpecialButton"
    add terminology for class "com.acme.test.SpecialClass"
    set aeteRes to get terminologies --set aeteRes to get
                   terminologies with object parameters
-- NOTE: There is currently an issue with using "get terminologies
-- with object parameters"
-- Use "get terminologies with <<class objt>>"
-- instead

    add resource aeteRes to file "HD:TestRes" of
type "aete" id 0 ¬
        with replacing allowed
end tell
```

Important:
In the above AppleScript, << and >> must be replaced with their single-character equivalents - the Option-\ and Option-Shift-\ characters in order for the script to function correctly.

When installing the "Add Resource" scripting addition, increase the partition size of the Script Editor to about 1500K, and enter and run this script. Then you can find the resource in the resulting "HD:TestRes" file and edit it with your favorite `'aete'` editor: either derez it and edit the .r file, or use Resorcerer or ResEdit with the `'aete'` template.

When cleaning up the terminology for the scripter, you should delete all the events and properties that are not relevant. You may want to delete non-relevant classes well; however, if dynamic terminology is on, then the terminologies of classes you have removed will show up again dynamically. Currently, to get around the problem, you can delete all events and properties of those classes but leave the classes around. Since the dynamic terminology generation does not try to override the classes already in the `'aete'`, these classes would remain empty.

You should never change the four-character codes in the aete. However, you are encouraged to change the names of events, parameters, classes, and properties if they are generated from programming names and not the display names. You should also add comments to inform the scripter what the individual classes are and how they are used.

After you have edited the `'aete'` resource into its final form, you should probably delete the Developer Suite since it is not intended for the scripter.

Dynamic terminology is controlled by the `'scsz'` resource. You may edit the `'scsz'` resource to disable dynamic terminology and thus freeze the terminology to be what you declare in the modified `'aete'`. (If you do this, you don't need to worry about automatic generation of unwanted terminology.) But this too has drawbacks. If your Java application is extensible (for example, through the `start tool` event), new classes can be introduced at run time; the only way to make these classes scriptable is to enable dynamic terminology in the `'scsz'` resource.

You are strongly encouraged to edit and trim down your `'aete'` resource, otherwise the scripter will likely be overwhelmed by the number of entries in the dictionary. You should also give the user some suggestions on where they should direct their scripting efforts. Then the scripter will have a better chance of fully utilizing your application. It is recommended that you ship sample scripts with your application to help the scripter with terminology idiosyncrasies and to demonstrate how to make use of scripting support effectively.

Back to top

Changes in MRJ 2.2

MRJ 2.2 has made some significant changes to the scripting model used by the MRJ. These changes lie in two main areas: changes to the terminology, and improved support for alternative containment hierarchies. These changes are documented in *About MRJ Scripting* which is part of the MRJ SDK 2.2.

Terminology Changes

There are some changes in the terminology to avoid conflicts with built-in AppleScript key words:

- "file" has been changed to "Java file"
- "get" has been changed to "get dictionary item"

Java array support has been improved. In previous MRJ versions, it was possible to get scripting errors when referring to the index of an array item. In MRJ 2.2, these type of references are now possible:

```
set testobject of frame 1 to {1,2,3}
set item 2 of testobject of frame 1 to 4
get testobject of frame 1
```

This operation is now possible and will return the value {1, 4, 3} where previously, it would have caused an error.

Additionally, it is now possible to manipulate Java Vector objects just like an AppleScript Array. For example, you can now do the following:

```
get item 2 of testvector of frame 1
set item 2 of testvector of frame 1 to "gamma"
get item 2 of testvector of frame 1
```

This operation will return the result "gamma". In previous versions of the MRJ, the only way to manipulate Vector objects was via the class methods.

The "make" event of the core suite is now supported. It is analogous to calling the constructor of a class directly. For example:

```
make new class "java.awt.Frame"
```

Corresponds to calling "new Frame( )" from Java. You can even do more complex operations such as:

```
make new class "java.awt.Frame" with properties {title: "hello",
  |visible|:true}
make new class "java.awt.Button" with data "Click me" at window 1
```

This is equivalent to the following:

```
Frame f = new Frame();
f.setTitle("hello");
f.setVisible(true);
f.add(new Button("Click me"));
```

Finally, MRJ 2.2 adds support for object references. For example, if you want to store an object reference for later use, you can store it in the "object collection" property of the Java application. For example:

```
make new class "xyz"
set alpha of its object collection to its result object
```

This saves a reference to the new xyz object under the variable name "alpha", If you need to retrieve that reference later, you may refer to it as:

```
alpha of its object collection
```

Note that if AppleScript is holding a reference to a particular object, the garbage collector cannot free that memory even if the object is no longer in use. You must explicitly remove the reference from the object collection as follows:

```
remove its object collection parameter "alpha"
```

Support for Alternative Containment Hierarchies

In the MRJ, typically objects must be referenced from AppleScript in terms of the visual containment hierarchy. Thus, if a button is in a panel in a window, you must access the button as follows:

```
get button 1 of panel 1 of frame 1
```

This can be a nuisance if you have a deep containment hierarchy such as in the case of swing. In MRJ 2.2, a developer may specify an alternate hierarchy by implementing the com.apple.scripting.AlternateHierarchy interface. This bypasses the traditional visual containment hierarchy.

For example:

```
public class test extends Frame implements AlternateHierarchy
{
    public Button getButton( int i )
    {
        ... // get the ith button in the frame
    }
    ...
}
```

This allows the developer to write the following script:

```
get button 2 of window 1
```

regardless of how many layers deep button 2 is. However, it is the responsibility of the developer to keep track of items in the custom hierarchy and writing the retrieval method.

Back to top

Summary

Incorporating AppleScript into your Java application allows increased automation support. Adding basic scriptability is very simple. You need only to add a pre-generated `'aete'` and a `'scsz'` resource; however, more advanced scripting may require careful editing of these resource types for finer control of which objects and methods are exposed in the dictionary.

Further References

- AppleScript Language Guide: English Dialect
- MRJ SDK 2.2 (About MRJ Scripting)
- AppleScript Home Page

Back to top

Downloadables

Acrobat version of this Note (49K).

To contact us, please use the Contact Us page.
Updated: 02-February-2000

Technotes | Contents
Previous Technote | Next Technote