Version 1.0 Beta 1

Converting Applets

This page steps you through the process of converting 1.0 Alpha 3 applets to use the new API:

Running the Upgrade Script

The first step in converting any applet is to run the **upgrade** script on it. The **upgrade** script uses the **sed** utility to perform some simple substitutions. The **sed** utility is available on all UNIX systems, but not on most PCs.

If your system doesn't have the **sed** utility, see the Upgrading by Hand document for instructions.

The **upgrade** script expects a 1.0 Alpha 3 applet at its standard input, and produces the updated applet at its standard output. Here's an example of using the **upgrade** script on the applet NervousText.java:

```
upgrade < alpha/NervousText.java > beta/NervousText.java
```

Once you've run the **upgrade** script, try to compile your applet. Maybe you'll be lucky –– maybe it will compile cleanly and you won't have to do anything else to make it work.

Here's an example of compiling NervousText.java:

```
cd beta
javac NervousText.java
```

The compiler output shows that the applet was successfully converted, except for two errors:

```
NervousText.java:27: Variable font in class java.awt.Component not accessible from class NervousTe
        font = f;
        ^
NervousText.java:64: Return required at end of boolean mouseDown(java.awt.Event, int, int).
public boolean mouseDown(java.awt.Event evt, int x, int y) {
               ^
2 errors
```

Making Changes by Hand

Chances are that the changes that the **upgrade** script makes won't be sufficient –– that you'll have to make more changes to your applet. The reason is that we've made some significant changes to the API and the Java language. The **upgrade** script can handle straightforward API and language changes, which are the majority of many applets' conversion needs, but you must make the less straightforward changes yourself.

For example, in the new API, event–handling methods such as `mouseDown()` must now return a boolean value –– `true` if the method completely handled the event, and `false` if it didn't. Usually, you return `true`. Making this change to NervousText.java (as shown below) fixes one compile–time error.

```
public boolean mouseDown(java.awt.Event evt, int x, int y) {
    . . .
    return true; //ADDED BY HAND
}
```

To take away the other compile–time error, replace the reference to Applet's `font` instance variable (`font = f;`) with a method call:

```
setFont(f); //ADDED BY HAND
```

Here's a list of the API and language changes that are most likely to affect applets. Many items are linked to more complete discussions in Changes Since the Alpha3 Release.

- The Applet class is now a subclass of java.awt.Panel, which means that applets now use the same event–handling and drawing interfaces as the AWT.
- The Applet class no longer has any public instance variables.
- The AWT package has changed significantly.
- Images are now processed asynchronously. This affects applets that care about the size of an image, since you can't rely on getWidth() (for example) returning a valid value. See Converting Applets that Use Images for details.
- The exception hierarchy has changed.
- Methods must declare the exceptions they can throw.

Putting an Applet into an HTML Document

The first step in running applets with the Applet Viewer is to create an HTML page that includes the applet (using the new APPLET tag). One way to do this is to copy an old page that includes the applet, and make the following changes:

OLD	NEW
<APP	<APPLET
SRC="path"	CODEBASE="path/classes"
[If there's no SRC]	CODEBASE="classes"
	--or move the applet up one directory
CLASS="foo"	CODE="foo.class"
AppletAttribute=...	*See below*
>	></APPLET>
[WIDTH & HEIGHT optional]	[WIDTH & HEIGHT required]

Here's an example of the old APP tag for the NervousText applet:

```
<app class="NervousText" width=200 height=50>
```

And here's the new APPLET tag:

```
<applet codebase="classes" code="NervousText.class"
        width=200 height=50></applet>
```

The new APPLET tag differs most significantly from the APP tag in two ways:

- You specify applet attributes (now called parameters) using a new PARAM tag.
- You can specify text to be displayed by browsers that don't understand the APPLET tag.

The PARAM tag must appear between <APPLET> and </APPLET> tags, as follows:

```
<APPLET ...>
<PARAM NAME=attribute1Name VALUE=attribute1Value>
<PARAM NAME=attribute2Name VALUE=attribute2Value>
</APPLET>
```

See the README document for more examples of using the PARAM tag.

Any text you include between <APPLET> and </APPLET> is ignored by Java–enabled browsers, but displayed by other browsers. For example, when the Applet Viewer (or a browser that understands the APPLET tag) processes the HTML code shown below, it loads the NervousText applet (from the same directory as the HTML file) and gives it an initial height of 200 pixels and width of 50. If you try to view the HTML code with a browser that doesn't understand the APPLET tag, you see the paragraph "You're viewing this page...." See for yourself what happens with your browser.

```
<applet code="NervousText.class" width=200 height=50>
<blockquote>
<hr>
<em>
You're viewing this page with a browser
that doesn't understand the APPLET tag.
If you were using a Java-enabled browser
that understood the APPLET tag,
you'd see jittery text here.
</em>
<hr>
</blockquote>
</applet>
```

See the README document for details of using the APPLET tag.

Running an Applet with the Applet Viewer

Once you've created an HTML file that includes your applet, just invoke the **appletviewer** command, giving the HTML file's name or URL as the argument:

```
appletviewer index.html
```

See the Applet Viewer documentation for more information.

Debugging an Applet with JDB

Note: The Java Debugger (JDB) is an early prototype designed to test the debugging API. We've included it in this release because it might be of some help to you.

Before debugging an applet, compile it with the **–g** option, if possible. Otherwise, you won't be able to see local variables in the debugger.

To start debugging an applet, use the Applet Viewer's **–debug** option. For best results, invoke the Applet Viewer from the directory that contains the HTML file.

Here's an example of compiling an applet and starting up the debugger:

```
javac -g NervousText.java
appletviewer -debug index.html
Loading jdb ...
0xee300f50:class(sun.applet.AppletViewer)
>
```

You're now in the Java Debugger. The Applet Viewer won't come up until you start its execution. This is your chance to set breakpoints in your applet's start() method. (Do NOT set a breakpoint in **init()** –– due to a race condition with the drawing thread, that

usually results in an exception.) For example:

```
> stop in NervousText.start
Breakpoint set at NervousText.start
> run
run sun.applet.AppletViewer index.html
running ...
main[1]
```

Note: Do NOT enter **run** more than once in an applet debugging session. Because of class loader interactions, the second **run** will fail.

After a few seconds, you'll see the Applet Viewer come up. A little while later, you'll see the following:

```
Breakpoint hit: NervousText.start (NervousText.java:35)
Thread-6[1]
```

This tells you that you have hit the breakpoint and that the stopped thread is named "Thread–6". Now you can list the method's code, check the values of local variables, and step through the method. Use the **help** command to find out what commands you can use.

```
Thread-6[1] help
** command list **
threads [threadgroup]     -- list threads
thread <thread id>        -- set default thread
suspend [thread id(s)]    -- suspend threads (default: all)
resume [thread id(s)]     -- resume threads (default: all)
where [thread id] | all   -- dump a thread's stack
threadgroups              -- list threadgroups
threadgroup <name>        -- set current threadgroup

print <id> [id(s)]        -- print object or field
dump <id> [id(s)]         -- print all object information

locals                    -- print all local variables in current stack frame

classes                   -- list currently known classes
methods <class id>        -- list a class's methods

stop in <class id>.<method> -- set a breakpoint in a method
stop at <class id>:<line> -- set a breakpoint at a line
up [n frames]             -- move up a thread's stack
down [n frames]           -- move down a thread's stack
clear <class id>:<line>   -- clear a breakpoint
step                      -- execute current line
cont                      -- continue execution from breakpoint

catch <class id>          -- break for the specified exception
ignore <class id>         -- ignore when the specified exception
```

```
  list [line number]        -- print source code
  use [source file path]    -- display or change the source path

  memory                    -- report memory usage
  load classname            -- load Java class to be debugged
  run                 -- start execution of a loaded Java class
  !!                        -- repeat last command
  help (or ?)               -- list commands
  exit (or quit)            -- exit debugger
  Thread-6[1]
```

See the Java Debugging page for more information on using the debugger, including tutorials.
