# Technote 1168

## The Care And Feeding of Runtime.exec

CONTENTS

Runtime.exec( ) is probably the single least cross-platform-compatible part of the Java API set. It assumes the existence of a command-line interface to the OS and the ability to launch arbitrary apps that can accept arbitrary parameters. Nevertheless, there are times when you need to use it -- for instance, to open a URL in a Web browser or to spawn a new Java process. This technote describes MRJ 2.1's implementation of `Runtime.exec` and how it differs from that of the JDK.

## The Platform-Dependent Perils of `Runtime.exec`

The standard Java method `Runtime.exec` launches new processes on the machine on which it's running. You pass it a command line, either as a String of space-delimited arguments or as an array of argument Strings. According to the documentation in *The Java Class Libraries* :

> "This method executes the **platform-dependent** program specified ... The program may be specified using a **platform-dependent** absolute pathname or as a **platform-dependent** command name found using the **platform-dependent** search path. ...
>
> "`exec` returns a `Process` object, which has methods for obtaining the standard input, standard output, and standard error [streams] of the newly created process." *[Emphasis added]*

You may notice that they are subtly trying to imply that much of the behavior of this method is, well, **platform-dependent**. (And even the "standard" streams they refer to aren't at all standard on Mac OS.)

It is therefore quite easy, if you use this method, to end up with code that won't run or runs incorrectly on some operating systems. The Mac OS is at a disadvantage here because, unlike Unix or Windows, it is not a thin GUI veneer over a command-line-based OS. There is no such thing as a command line in the Mac OS itself, so it's been rather difficult for MRJ to support the kinds of uses to which imaginative developers put `Runtime.exec`.

Nevertheless, we do try, and the good news is that MRJ 2.1 supports a lot more of the typical uses of `Runtime.exec` than its predecessors did. This technote explains in detail what we do and don't support.

### JConfig

If you're willing to incorporate third-party code (which includes native libraries), you might consider Samizdat Productions' freeware JConfig library.

> JConfig "is a cross-platform library which supplements the core Java API. It lets you work with files, web browsers, processes, file types, and other system-level items in a much more advanced manner than that provided by the standard Java class libraries." (from the manual)

JConfig is a lot more powerful than `Runtime.exec` and lets you deal with a lot of nasty higher-level issues like locating the user's preferred web browser, or indeed the preferred helper app for any arbitrary URL type. It runs on Mac OS, Windows, and Unix, and a lot of developers who work with MRJ swear by it. (Apple, of course, g has no official connection with JConfig or Samizdat Productions and can't vouch for the reliability of the library. But we think it's cool.)

Back to top

## The Path To The App

The Mac OS has no search path, so you must specify an application using a real file path. For example, just referring to the application as "`netscape`" is not going to work. There are several ways to do this:

### Relative path

If the application you're launching is at a known location relative to the application running (i.e., both are part of the same installation) you can provide a path that's relative to the current application. This works reliably only if you're running an application built with JBindery, not if you run your code directly from JBindery itself.

### User preference

If the app to launch is not part of your installation (for instance, if it's something standard like SimpleText or a web browser), you'll need to provide an absolute path.

The most reasonable cross-platform way to do this is to store the path in a preference file. If the preference doesn't exist yet, or if you tried using the preferred path and it failed, you should put up a dialog box and ask the user to select the application, then store the path into the preference file. See Technote 1134, "The Preferences Problem," for more discussion about setting the correct path using a preference file.

### Lookup by creator

If you're willing to use some Mac-specific code, it's friendlier to use the Mac OS facility to locate an application automatically given its *creator code* , which is a unique four-letter code assigned to that application.

You can find the creator of any application by using ResEdit's `Get File/Folder Info` command. The creators of some common applications are:

| | |
|---|---|
| Microsoft Internet Explorer | `MWIE` |
| Netscape Navigator | `MOSS` |
| SimpleText | `ttxt` |
| Finder | `FNDR` |

The MRJToolkit library, provided as part of the MRJ SDK (including extensive documentation) includes a `findApplication` method that takes a creator code and returns a File object that points to the app. For example:

```
import com.apple.mrj.*;
...
```

```
File app = MRJFileUtils.findApplication(new MRJOSType("ttxt"));
```

### Let JConfig locate it for you

The third-party JConfig library from Samizdat Productions, described above, has features that can locate the user's preferred helper app for any type of URL, which is one of the common uses of `Runtime.exec`.

Back to top

## But I Came Here For An Argument!

The Mac OS uses a mechanism known as AppleEvents to send messages from one process to another. AppleEvents are much higher level than command-line arguments, and include a rich set of data types and tags. However, this means that MRJ can't take a set of arbitrary argument strings and send them to an application in any meaningful way. (Remember all the stuff about "platform-dependent" in the first section?)

MRJ tries to detect certain common types of command-line arguments and convert them into AppleEvents. In general, the two types of arguments MRJ knows about are file paths and URLs. Any argument containing a ":" character is interpreted as a URL, and anything else is interpreted as a file path. (These rules don't apply when you're launching another Java process, though. See the next section.)

**Note:**
Versions of MRJ prior to 2.1 do not support URL parameters and just assume everything is a file path.

(If you're AppleEvent savvy, you may wish to know that file paths are sent in a single `'odoc'` event, while URLs are sent in individual `'GURL'` events.)

There are a couple of things to keep in mind:

### No multi-launching

The Mac OS does not allow the same app to be launched multiple times, so if you call `Runtime.exec` on a running app, or call it multiple times on the same app, the same instance of the app will receive each request and open each item.

### URLs usually don't open a new window

The convention in web browsers, at least, is that they display most URLs in the active window, replacing its previous contents, rather than opening a new window. (They do open files in new windows.) That means that if you send a browser a URL argument it will effectively ignore any argument that came before it. (For instance, if you pass it two URLs, it will try to show the first one and then immediately the second one in the same window. If you pass a file and a URL, it will open a new window for the file, but then show the URL in that window.)

Back to top

## Launching Java

It turns out that one of the useful things you can do with `Runtime.exec` is to launch a new Java process. This is useful if you need to give the code you're launching complete independence of your current process. For instance, some Java tools like `javac` never reclaim memory or close files, and for practical purposes must be run in a separate process that is cleaned up by the OS when it quits.

In MRJ 2.1 we valiantly added some special cases (some say "hacks") to facilitate this. If the application/command named in the first argument does not point to an existing file, and it does include the substring "`java`" (independent of case), then we assume you are trying to launch a Java process, just as if you were invoking the JDK from a command line.

**Note:**
Versions of MRJ prior to 2.1 do not support this feature.

For example, you can invoke the Java compiler thusly:

```
String args = {"java", "sun.tools.javac.Main", ...javac argument list...};
Runtime.getRuntime().exec(args);
```

Since we know the thing on the other end is a Java process, we allow a bit more general functionality:

- You can pass any arguments you like; they don't have to be URLs or files. They get passed directly to the `main()` method of the app's main class, as Strings.
- You can use the `Process` object [see below] to access the Java process's standard input / output / error streams.

## Special flags for '`java`'

The JDK '`java`' command takes a number of flag parameters. MRJ 2.1 supports a subset of these:

| Flag | Purpose | Status |
|---|---|---|
| `-help` | Prints help message & exits | **Supported** |
| `-version` | Prints version & exits | **Error** |
| `-v` | Turns on verbose mode | **Error** |
| `-classpath` | Sets class search path* | **Supported** |
| `-D` | Sets system property | **Supported** |
| `-X` | Prints help on nonstandard options & exits | **Error** |
| `-Xdebug` | Enable remote debugging | **Error** |
| `-Xnoasyncgc` | Disable asynchronous garbage collection | **Supported** |
| `-Xnoclassgc` | Disable class GC | **Supported** |
| `-Xss` | Set max native stack size | **Ignored** |
| `-Xoss` | Set max Java stack size | **Ignored** |
| `-Xms` | Set initial Java heap size | **Ignored** |
| `-Xmx` | Set max Java heap size | **Ignored** |
| `-Xprof` | Enable method profiling | **Ignored** |
| `-Xiprof` | Enable instruction profiling | **Ignored** |
| `-Xhprof` | Enable heap profiling | **Ignored** |
| `-Xverify` | Enable bytecode verification | **Ignored** |

Future versions of MRJ may support more of these flags.

* A note on `-classpath`: As in JDK 1.2, you only need to specify *additions* to the regular system classpath -- you do not need to add `classes.zip` or `JDKClasses.zip` or `rt.jar`, and you'll get errors if you do. Classpath entries can be absolute paths, or can be relative to the current working directory, which by default is the directory containing the current application.

Back to top

## Pasteurized Processes

The `Runtime.exec` method, if it succeeds, returns a `Process` object. You can use this object to check the status of the process, force it to quit, and access its input / output / error streams.

Most of these work as advertised, but the latter usage is problematic -- the Mac OS has no notion of text streams attached to applications, so it's meaningless to try to access them. MRJ just returns null if you ask `Process` for a stream associated with a normal app.

The exception is Java processes, as described above. Since these are special processes that just run Java code, they do have console I/O streams, and the Process API allows you to access them.

Back to top

## Example: Opening A URL In A Browser

Here's some sample code by Levi Brown that demonstrates a cross-platform-savvy way to open a URL in a Web browser. It presents a `FileDialog` prompting the user to locate their preferred web browser:

```
import java.awt.Frame;
                  import java.awt.FileDialog;
                  import java.io.File;
                  import java.io.IOException;

                  public class ExecTest extends Frame
{

    public static void main(String[] args)
    {
        new ExecTest();
        System.exit(0);
    }

    public ExecTest()
    {
        String browserName;
        String url = "http://developer.apple.com/java/";

        //Set up a FileDialog for the user to locate the browser to use.
        FileDialog fileDialog = new java.awt.FileDialog(this);
        fileDialog.setMode(FileDialog.LOAD);
        fileDialog.setTitle("Choose the Web browser to use:");
        fileDialog.setVisible(true);

        //Retrieve the path information from the dialog and verify it.
        String resultPath = fileDialog.getDirectory();
        String resultFile = fileDialog.getFile();
        fileDialog.dispose();
        if(resultPath != null && resultPath.length() != 0
                   && resultFile != null && resultFile.length() != 0)
        {
            File file = new File(resultPath + resultFile);
            if(file != null)
```

```
                {
                    browserName = file.getPath();

                    try{
                        //Launch the browser and pass it the desired URL
                        Runtime.getRuntime().exec(new String[] {browserName, url});
                    }
                    catch (IOException exc)
                    {
                        exc.printStackTrace();
                    }
                }
            }
        }
    }
```

This example should be refined, if used in a real setting, to store the location of the browser (`browserName`) in a preference file, and only ask the user if the cached browser could not be located. It should also make the prompt string localizable.

Alternatively, as described above, you could locate a browser by its application signature, or use JConfig to handle the whole open-the-URL process for you.

## Further References

- Technote 1134: The Preferences Problem

Back to top

## Downloadables

Acrobat version of this Note (K).

Back to top

---

**To contact us, please use the Contact Us page.**
**Updated: 17-May-1999**

Technotes | Contents
Previous Technote | Next Technote