JAVAWORLD
SOLUTIONS FOR JAVA DEVELOPERS

This story appeared on JavaWorld at
http://www.javaworld.com/javaworld/javaqa/2003-08/01-qa-0808-property.html

Smartly load your properties

Strive for disk location-independent code nirvana

By Vladimir Roubtsov, JavaWorld.com, 08/08/03

August 8, 2003

Q: What is the best strategy for loading property and configuration files in Java?

A: When you think about how to load an external resource in Java, several options immediately come to mind: files, classpath resources, and URLs. Although all of them eventually get the job done, experience shows that classpath resources and URLs are by far the most flexible and user-friendly options.

In general, a configuration file can have an arbitrarily complex structure (e.g., an XML schema definition file). But for simplicity, I assume below that we're dealing with a flat list of name-value pairs (the familiar `.properties` format). There's no reason, however, why you can't apply the ideas shown below in other situations, as long as the resource in question is constructed from an `InputStream`.

Evil java.io.File

Using good old files (via `FileInputStream`, `FileReader`, and `RandomAccessFile`) is simple enough and certainly the obvious route to consider for anyone without a Java background. But it is the worst option in terms of ease of Java application deployment. Using absolute filenames in your code is not the way to write portable and disk position-independent code. Using relative filenames seems like a better alternative, but remember that they are resolved relative to the JVM's current directory. This directory setting depends on the details of the JVM's launch process, which can be obfuscated by startup shell scripts, etc. Determining the setting places an unfair amount of configuration burden on the eventual user (and in some cases, an unjustified amount of trust in the user's abilities). And in other contexts (such an Enterprise JavaBeans (EJB)/Web application server), neither you nor the user has much control over the JVM's current directory in the first place.

An ideal Java module is something you add to the classpath, and it's ready to go. Think EJB jars, Web applications packaged in `.war` files, and other similarly convenient deployment strategies. `java.io.File` is the least platform-independent area of Java. Unless you absolutely must use them, just say no to files.

Classpath resources

Having dispensed with the above diatribe, let's talk about a better option: loading resources through classloaders. This is much better because classloaders essentially act as a layer of abstraction between a resource name and its actual location on disk (or elsewhere).

Let's say you need to load a classpath resource that corresponds to a `some/pkg/resource.properties` file. I use *classpath resource* to mean something that's packaged in one of the application jars or added to the classpath before the application launches. You can add to the classpath via the `-classpath` JVM option each time the application starts or by placing the file in the `<jre home>\classes` directory once and for all. The key point is that *deploying a classpath resource is similar to deploying a compiled Java class*, and therein lies the convenience.

You can get at `some/pkg/resource.properties` programmatically from your Java code in several ways. First, try:

```
ClassLoader.getResourceAsStream ("some/pkg/resource.properties");
Class.getResourceAsStream ("/some/pkg/resource.properties");
ResourceBundle.getBundle ("some.pkg.resource");
```

Additionally, if the code is in a class within a `some.pkg` Java package, then the following works as well:

```
Class.getResourceAsStream ("resource.properties");
```

Note the subtle differences in parameter formatting for these methods. All `getResourceAsStream()` methods use slashes to separate package name segments, and the resource name includes the file extension. Compare that with resource bundles where the resource name looks more like a Java identifier, with dots separating package name segments (the `.properties` extension is implied here). Of course, that is because a resource bundle does not have to be backed by a `.properties` file: it can be a class, for a example.

To slightly complicate the picture, `java.lang.Class`'s `getResourceAsStream()` instance method can perform package-relative resource searches (which can be handy as well, see "Got Resources?"). To distinguish between relative and absolute resource names, `Class.getResourceAsStream()` uses leading slashes for absolute names. In general, there's no need to use this method if you are not planning to use package-relative resource naming in code.

It is easy to get mixed up in these small behavioral differences for `ClassLoader.getResourceAsStream()`, `Class.getResourceAsStream()`, and `ResourceBundle.getBundle()`. The following table summarizes the salient points to help you remember:

Behavioral differences

Method	Parameter format	Lookup failure behavior	Usage example
`ClassLoader. getResourceAsStream()`	"/"-separated names; no leading "/" (all names are absolute)	Silent (returns `null`)	`this.getClass().getClassLoader() .getResourceAsStream ("some/pkg/resource.properties")`
`Class. getResourceAsStream()`	"/"-separated names; leading "/" indicates absolute names; all other names are relative to the class's package	Silent (returns `null`)	`this.getClass() .getResourceAsStream ("resource.properties")`
`ResourceBundle. getBundle()`	"."-separated names; all names are absolute; `.properties` suffix is implied	Throws unchecked `java.util.MissingResourceException`	`ResourceBundle.getBundle ("some.pkg.resource")`

From data streams to java.util.Properties

You might have noticed that some previously mentioned methods are half measures only: they return `InputStream`s and nothing resembling a list of name-value pairs. Fortunately, loading data into such a list (which can be an instance of `java.util.Properties`) is easy enough. Because you will find yourself doing this over and over again, it makes sense to create a couple of helper methods for this purpose.

The small behavioral difference among Java's built-in methods for classpath resource loading can also be a nuisance, especially if some resource names were hardcoded but you now want to switch to another load method. It makes sense to abstract away little things like whether slashes or dots are used as name separators, etc. Without further ado, here's my `PropertyLoader` API that you might find useful (available with this article's download):

```
public abstract class PropertyLoader
{
    /**
     * Looks up a resource named 'name' in the classpath. The resource must map
     * to a file with .properties extention. The name is assumed to be absolute
     * and can use either "/" or "." for package segment separation with an
     * optional leading "/" and optional ".properties" suffix. Thus, the
     * following names refer to the same resource:
     * <pre>
     * some.pkg.Resource
     * some.pkg.Resource.properties
     * some/pkg/Resource
     * some/pkg/Resource.properties
     * /some/pkg/Resource
     * /some/pkg/Resource.properties
     * </pre>
     *
     * @param name classpath resource name [may not be null]
     * @param loader classloader through which to load the resource [null
     * is equivalent to the application loader]
     *
     * @return resource converted to java.util.Properties [may be null if the
     * resource was not found and THROW_ON_LOAD_FAILURE is false]
     * @throws IllegalArgumentException if the resource was not found and
     * THROW_ON_LOAD_FAILURE is true
     */
    public static Properties loadProperties (String name, ClassLoader loader)
    {
        if (name == null)
            throw new IllegalArgumentException ("null input: name");

        if (name.startsWith ("/"))
            name = name.substring (1);

        if (name.endsWith (SUFFIX))
            name = name.substring (0, name.length () - SUFFIX.length ());

        Properties result = null;

        InputStream in = null;
        try
        {
            if (loader == null) loader = ClassLoader.getSystemClassLoader ();

            if (LOAD_AS_RESOURCE_BUNDLE)
            {
                name = name.replace ('/', '.');
                // Throws MissingResourceException on lookup failures:
                final ResourceBundle rb = ResourceBundle.getBundle (name,
                    Locale.getDefault (), loader);

                result = new Properties ();
                for (Enumeration keys = rb.getKeys (); keys.hasMoreElements ();)
                {
                    final String key = (String) keys.nextElement ();
                    final String value = rb.getString (key);

                    result.put (key, value);
                }
            }
            else
```

```
            {
                name = name.replace ('.', '/');

                if (! name.endsWith (SUFFIX))
                    name = name.concat (SUFFIX);

                // Returns null on lookup failures:
                in = loader.getResourceAsStream (name);
                if (in != null)
                {
                    result = new Properties ();
                    result.load (in); // Can throw IOException
                }
            }
        }
        catch (Exception e)
        {
            result = null;
        }
        finally
        {
            if (in != null) try { in.close (); } catch (Throwable ignore) {}
        }

        if (THROW_ON_LOAD_FAILURE && (result == null))
        {
            throw new IllegalArgumentException ("could not load [" + name + "]"+
                " as " + (LOAD_AS_RESOURCE_BUNDLE
                ? "a resource bundle"
                : "a classloader resource"));
        }

        return result;
    }

    /**
     * A convenience overload of {@link #loadProperties(String, ClassLoader)}
     * that uses the current thread's context classloader.
     */
    public static Properties loadProperties (final String name)
    {
        return loadProperties (name,
            Thread.currentThread ().getContextClassLoader ());
    }

    private static final boolean THROW_ON_LOAD_FAILURE = true;
    private static final boolean LOAD_AS_RESOURCE_BUNDLE = false;
    private static final String SUFFIX = ".properties";
} // End of class
```

The Javadoc comment for the `loadProperties()` method shows that the method's input requirements are quite relaxed: it accepts a resource name formatted according to any of the native method's schemes (except for package-relative names possible with `Class.getResourceAsStream()`) and normalizes it internally to do the right thing.

The shorter `loadProperties()` convenience method decides which classloader to use for loading the resource. The solution shown is reasonable but not perfect; you might consider using techniques described in "Find a Way Out of the ClassLoader Maze" instead.

Note that two conditional compilation constants control `loadProperties()` behavior, and you can tune them to suit your tastes:

- `THROW_ON_LOAD_FAILURE` selects whether `loadProperties()` throws an exception or merely returns `null` when it can't find the resource
- `LOAD_AS_RESOURCE_BUNDLE` selects whether the resource is searched as a resource bundle or as a generic classpath resource

Setting `LOAD_AS_RESOURCE_BUNDLE` to `true` isn't advantageous unless you want to benefit from localization support built into `java.util.ResourceBundle`. Also, Java internally caches resource bundles, so you can avoid repeated disk file reads for the same resource name.

More things to come

I intentionally omitted an interesting classpath resource loading method, `ClassLoader.getResources()`. Despite its infrequent use, `ClassLoader.getResources()` allows for some very intriguing options in designing highly customizable and easily configurable applications.

I didn't discuss `ClassLoader.getResources()` in this article because it's worthy of a dedicated article. As it happens, this method goes hand in hand with the remaining way to acquire resources: `java.net.URL`s. You can use these as even more general-purpose resource descriptors than classpath resource name strings. Look for more details in the next **Java Q&A** installment.

About the author

Vladimir Roubtsov has programmed in a variety of languages for more than 13 years, including Java since 1995. Currently, he develops enterprise software as a senior engineer for Trilogy in Austin, Texas.