The Java™ Tutorials

Trail: Deployment
Lesson: Packaging Programs in JAR Files
Section: Using JAR-related APIs

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases.

The JarClassLoader Class

The `JarClassLoader` class extends `java.net.URLClassLoader`. As its name implies, `URLClassLoader` is designed to be used for loading classes and resources that are accessed by searching a set of URLs. The URLs can refer either to directories or to JAR files.

In addition to subclassing `URLClassLoader`, `JarClassLoader` also makes use of features in two other new JAR-related APIs, the `java.util.jar` package and the `java.net.JarURLConnection` class. In this section, we'll look in detail at the constructor and two methods of `JarClassLoader`.

The JarClassLoader Constructor

The constructor takes an instance of `java.net.URL` as an argument. The URL passed to this constructor will be used elsewhere in `JarClassLoader` to find the JAR file from which classes are to be loaded.

```
public JarClassLoader(URL url) {
    super(new URL[] { url });
    this.url = url;
}
```

The `URL` object is passed to the constructor of the superclass, `URLClassLoader`, which takes a `URL[]` array, rather than a single `URL` instance, as an argument.

The getMainClassName Method

Once a `JarClassLoader` object is constructed with the URL of a JAR-bundled application, it's going to need a way to determine which class in the JAR file is the application's entry point. That's the job of the `getMainClassName` method:

```
public String getMainClassName() throws IOException {
    URL u = new URL("jar", "", url + "!/");
    JarURLConnection uc = (JarURLConnection)u.openConnection();
    Attributes attr = uc.getMainAttributes();
    return attr != null
                   ? attr.getValue(Attributes.Name.MAIN_CLASS)
                   : null;
}
```

You may recall from a previous lesson that a JAR-bundled application's entry point is specified by the `Main-Class` header of the JAR file's manifest. To understand how `getMainClassName` accesses the `Main-Class` header value, let's look at the method in detail, paying special attention to the new JAR-handling features that it uses:

The JarURLConnection class and JAR URLs

The `getMainClassName` method uses the JAR URL format specified by the `java.net.JarURLConnection` class. The syntax for the URL of a JAR file is as in this example:

```
jar:http://www.example.com/jarfile.jar!/
```

The terminating `!/` separator indicates that the URL refers to an entire JAR file. Anything following the separator refers to specific JAR-file contents, as in this example:

```
jar:http://www.example.com/jarfile.jar!/mypackage/myclass.class
```

The first line in the `getMainClassName` method is:

```
URL u = new URL("jar", "", url + "!/");
```

This statement constructs a new `URL` object representing a JAR URL, appending the `!/` separator to the URL that was used in creating the `JarClassLoader` instance.

The java.net.JarURLConnection class

This class represents a communications link between an application and a JAR file. It has methods for accessing the JAR file's manifest. The second line of `getMainClassName` is:

```
JarURLConnection uc = (JarURLConnection)u.openConnection();
```

In this statement, `URL` instance created in the first line opens a `URLConnection`. The `URLConnection` instance is then cast to `JarURLConnection` so it can take advantage of `JarURLConnection`'s JAR-handling features.

Fetching Manifest Attributes: java.util.jar.Attributes

With a `JarURLConnection` open to a JAR file, you can access the header information in the JAR file's manifest by using the `getMainAttributes` method of `JarURLConnection`. This method returns an instance of `java.util.jar.Attributes`, a class that maps header names in JAR-file manifests with their associated string values. The third line in `getMainClassName` creates an `Attributes` object:

```
Attributes attr = uc.getMainAttributes();
```

To get the value of the manifest's `Main-Class` header, the fourth line of `getMainClassName` invokes the `Attributes.getValue` method:

```
return attr != null
               ? attr.getValue(Attributes.Name.MAIN_CLASS)
               : null;
```

The method's argument, `Attributes.Name.MAIN_CLASS`, specifies that it's the value of the `Main-Class` header that you want. (The `Attributes.Name` class also provides static fields such as `MANIFEST_VERSION`, `CLASS_PATH`, and `SEALED` for specifying other standard manifest headers.)

The invokeClass Method

We've seen how `JarURLClassLoader` can identify the main class in a JAR-bundled application. The last method to consider, `JarURLClassLoader.invokeClass`, enables that main class to be invoked to launch the JAR-bundled application:

```
public void invokeClass(String name, String[] args)
    throws ClassNotFoundException,
           NoSuchMethodException,
           InvocationTargetException
{
    Class c = loadClass(name);
    Method m = c.getMethod("main", new Class[] { args.getClass() });
    m.setAccessible(true);
    int mods = m.getModifiers();
    if (m.getReturnType() != void.class || !Modifier.isStatic(mods) ||
        !Modifier.isPublic(mods)) {
        throw new NoSuchMethodException("main");
    }
    try {
        m.invoke(null, new Object[] { args });
    } catch (IllegalAccessException e) {
        // This should not happen, as we have disabled access checks
    }
}
```

The `invokeClass` method takes two arguments: the name of the application's entry-point class and an array of string arguments to pass to the entry-point class's `main` method. First, the main class is loaded:

```
Class c = loadClass(name);
```

The `loadClass` method is inherited from `java.lang.ClassLoader`.

Once the main class is loaded, the reflection API of the `java.lang.reflect` package is used to pass the arguments to the class and launch it. You can refer to the tutorial on The Reflection API for a review of reflection.

Previous page: Using JAR-related APIs
Next page: The JarRunner Class