

Zippping Files Using Java

by Julian Robichaux, panagenda

originally published on socialbizug.org, July 2012

Sometimes it is convenient to generate a zip file programmatically. Perhaps you want to email a large text file to someone on a scheduled basis. Or you have to archive a day/week/month's worth of log files. Or you would like to combine several files together dynamically for download.

Whatever the need, Java has had built-in support for zipping and unzipping files almost since the language was born. Because the classes are part of the language itself, you can easily add zip file support to your Java agents and XPages without having to include any third-party libraries. Let's take a look at how it works.

Getting Started

The classes you'll need to use are primarily in the `java.util.zip` package (<http://docs.oracle.com/javase/6/docs/api/java/util/zip/package-summary.html>). You'll notice right away that there are Deflater/Inflater classes, GZIP classes, and Zip classes. Which one to use? For "normal" zip file reading and writing, you just need the Zip classes.

Along those lines, whenever you see any options to change the default settings of the zip operations — like changing the compression type or increasing the compression level — you should avoid the temptation to change those options. Not only are the defaults a good tradeoff between size and speed, but they are also the most compatible settings for whatever program will try to unzip your file later.

For the code we will go through in this article, you will need the following imports in your Java class:

```
import java.io.*;
import java.util.zip.*;
```

Method #1: Setting Up The Files

Here is the code for the general method that creates the zip file:

```
public int createZipFileTest() {
    // the new zip file we will create
    File myNewZipFile = new File("c:\\MyZipFile.zip");

    // files and folders we want to add to the zip file
    File folderToZip = new File("c:\\temp\\mystuff");
    File anotherFileToZip = new File("c:\\temp\\foo.txt");
```

```

int count = 0;
ZipOutputStream out = null;
String folderBase = folderToZip.getAbsolutePath();
String fileBase = anotherFileToZip.getParent();

try {
    // set up the zip stream and add the files
    out = new ZipOutputStream(new FileOutputStream(myNewZipFile));
    count += addToZip(folderToZip, out, folderBase);
    count += addToZip(anotherFileToZip, out, fileBase);
} catch (Exception e) {
    // you really should log this or something...
    e.printStackTrace();
} finally {
    // make sure the zip stream gets closed properly
    try { out.close(); } catch (Exception ignored) {}
}

// return the total number of files we added to the zip file
return count;
}

```

The method starts out by just defining the zip file we will end up with, and the files and/or folders we will be zipping up. If the zip file already exists, it will be overwritten. Normally this information would probably be a parameter for the method call, but we're just doing an example.

We are figuring out the base directories (folderBase and fileBase) because all the file references inside the zip file have to be relative path references rather than absolute path references. You will see in the next method how this works, and why we pass this in instead of computing it dynamically when we add the files.

In the try/catch block we set up the ZipOutputStream, which is just a special OutputStream that compresses file entries as they are written to it, and it computes all the extra header information used by the zip file format. The “addToZip” calls use the method we will see below that actually adds files and folders to the ZipOutputStream.

After that, we close the ZipOutputStream in a finally block to make sure the zip file is properly closed, and we return the total number of files that were added to the new zip file.

Simple, right? Next up is the “addToZip” method.

Method #2: Adding Files To The New Zip File

Here is the code for actually adding files and folders to the ZipOutputStream:

```

public int addToZip(File fileOrFolder, ZipOutputStream zip, String baseDir)
throws IOException {
    int count = 0;

    // get an array of files to add to the zip file
    File[] files = {fileOrFolder};
    if (fileOrFolder.isDirectory()) { files = fileOrFolder.listFiles(); }

    // add each file, one at a time
    for (int i = 0; i < files.length; i++) {
        File file = files[i];

        if (file.isDirectory()) {
            // call this method recursively for subfolders
            count += addToZip(file, zip, baseDir);
        } else {
            // IMPORTANT: the ZipEntry file name must use "/", not "\".
            // Also it must be a relative path, not an absolute one.
            String name =
file.getAbsolutePath().substring(baseDir.length());
            name = name.replace('\\', '/');
            while (name.startsWith("/")) {
                name = name.substring(1);
            }

            // set up the new zip entry
            ZipEntry zipEntry = new ZipEntry(name);
            zipEntry.setTime(file.lastModified());
            zip.putNextEntry(zipEntry);

            // read the file into the zip entry
            int bufferSize = 2048;
            byte[] buffer = new byte[bufferSize];
            int len = 0;
            BufferedInputStream in = new BufferedInputStream(
                new FileInputStream(file), bufferSize);
            while ((len = in.read(buffer, 0, bufferSize)) != -1) {
                zip.write(buffer, 0, len);
            }
            in.close();

            // don't forget to close the entry when you're done
            zip.closeEntry();
            count++;
        }
    }

    // this will return the total number of files we added
    return count;
}

```

We start off by creating an array of files to add to the zip file. If the “fileOrFolder” parameter is just a file, this will be an array with a single entry; if it’s a folder, this will be an array of all the files and subfolders in that folder.

As we go one-by-one through the array of files, the first thing we look at is whether or not each Java File reference is a single file or a folder — Java uses the same “File” object for both, but we need to treat them differently. If it’s a folder, we can make a recursive call to this method to zip up the files in this folder (and any subfolders will also trigger recursive calls within the recursive call... fun!).

This also explains why we passed the base directory in as a parameter instead of computing it from the file reference. If we were computing the base directory, it would get recomputed again during the recursive calls, which means we would lose the subfolder names. Since we passed it in and reuse it on the recursive calls, the relative paths of subfolders will be correct. If that doesn’t make sense don’t worry; it works.

The main block of code in this method is (not surprisingly) the part that actually adds a file to the ZipOutputStream. When we create the “name” string, we take the full file path of the file we are zipping, and then make that path relative to the baseDir we were given. For example, if the baseDir is “c:\temp” and the file is “c:\temp\foo.txt”, the relative name we use will be “foo.txt”. If the file is “c:\temp\somefolder\foo.txt” then the relative name will be “somefolder/foo.txt”.

There are two important things here. First, the relative name should NOT start with “\” or “/”. Second, all the backslash characters (“\”) used in Windows paths MUST be changed to front slashes (“/”). If you don’t take care to do both these things, you can end up with a zip file that other programs might not be able to open properly.

After that we create a new ZipEntry using the relative name we calculated, and then we set the timestamp on that ZipEntry to the lastModified time of the file and add the ZipEntry to the ZipOutputStream. From there, it’s the standard Java process of reading from an InputStream (the file we are adding to the zip) and writing to an OutputStream (the zip file itself).

Note that we don’t write to the ZipEntry and then add the entry to the ZipOutputStream. Instead, the process is: create the entry, add to the stream, write to the stream, close the entry. This might be counter-intuitive — you might think you’re supposed to write to the entry and then add the whole thing to the stream — but that’s just how it works.

After the ZipEntry has been closed, the file has successfully been added to our ZipOutputStream.

Suggested Changes and Improvements

As with most example code, there are probably a few changes you might want to make before you use it in production. For example:

- Add a flag to indicate whether or not you want to add subfolders.
- Add a flag to indicate whether or not you should overwrite the zip file if it already exists.

- Make a generic wrapper around the process so you can arbitrarily specify which files and folders to add, or which zip file to create.
- More error handling.

However, this should be enough to get you started playing with code. As always, there are plenty of examples and third party libraries on the Internet too.