Distributing your Application as an executable JAR file

A JAR (Java ARchive) is a way of packaging together all of the resources associated with a program (class files, images, sounds, etc.). Putting your program in a JAR allows it to be distributed as a single executable file, saving space and simplifying the download process. The information in this tutorial applies to Java version 1.2 or higher. For more information about JAR files, follow Sun's tutorial. To learn about signing the JAR and Java Web Start.

A simple example. Let's say we wanted to distribute the simple program Hello.java as a JAR. First, we create a text file named `Hello.mf` which contains:

```
Manifest-Version: 1.0
Main-Class: Hello
```

Then, we create the archive by typing:

```
jar cmf Hello.mf Hello.jar Hello.class Hello.java
```

and run it by typing:

```
java -jar Hello.jar
```

The file Hello.jar can now be downloaded and executed.

Creating an executable JAR file. Here is the general procedure for creating an executable JAR:

1. Compile your java code, generating all of the program's class files.

2. Create a *manifest file* containing the following 2 lines:

   ```
   Manifest-Version: 1.0
   Main-Class: name of class containing main
   ```

   The name of the file should end with the `.mf` suffix. It is important that the file ends with a blank line.

3. To create the JAR, type the following command:

   ```
   jar cmf manifest-file jar-file input-files
   ```

   The *input-files* must include any class files, images, sounds, etc. that your program uses. Optionally, you can include the program's `.java` files in the JAR. See below for adding directories ot the JAR.

4. To view the contents of the JAR, type:

   ```
   jar tf jar-file
   ```

5. Execute the application from the command line by typing:

   ```
   java -jar jar-file
   ```

   If the application is GUI-based, you can also launch it by double-clicking the JAR file.

Accessing resources in a JAR. In general, the first step in accessing a JAR resource involves creating a URL. This might require modifying your program. For example, you can no longer use the following code fragment to read in an image that is stored in a file as follows

```
Image image = Toolkit.getDefaultToolkit().getImage(filename);
```

Instead, create the `URL` object using

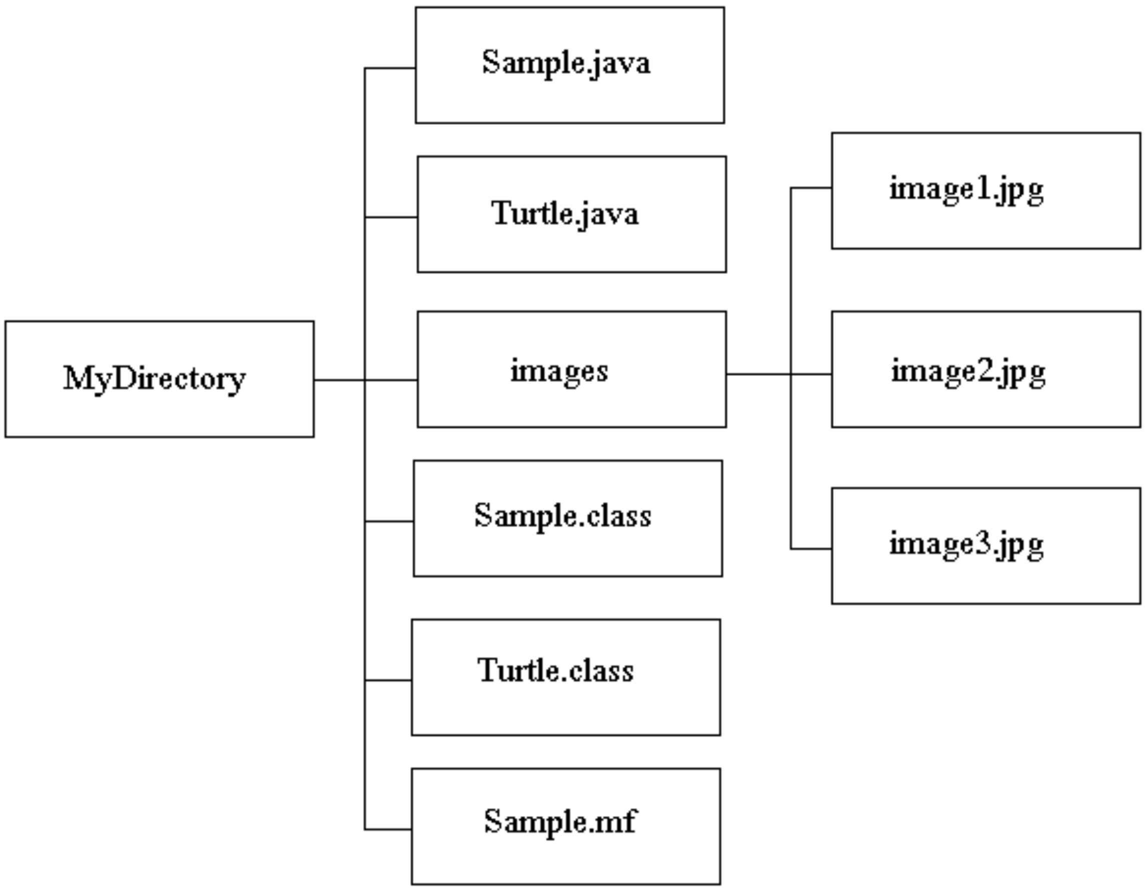
```
URL url = getClass.getResource(filename);
Image image = Toolkit.getDefaultToolkit().getImage(url);
```

Or, if the code is in a static method of class `X`, then create the URL with

```
URL url = X.class.getResource(filename);
```

Now, the resource can be accessed the same way, regardless of whether it is in a JAR or the current directory. See the method `play(filename)` in StdDraw.java for an example involving audio clips, and and the constructor `In(String s)` in In.java for an example involving text files.

JAR Subdirectories. The JAR format also support storing files in a directory structure. Consider a program `Sample.java`, which uses the Turtle Graphics interface to display a collection of pictures stored in a subdirectory called `images`. Our working directory looks like:



The Manifest should read:

```
Manifest-Version: 1.0
Main-Class: Sample
```

To create the JAR, type:

```
jar cmf Sample.mf Sample.jar Sample.class Turtle.class Sample.java Turtle.java images
```

The contents listing appears as:

```
META-INF/
META-INF/MANIFEST.MF
Sample.class
Turtle.class
Sample.java
Turtle.java
images/
images/image1.gif
images/image2.gif
images/image3.gif
```

Notice that the directory structure is still preserved (the META-INF directory is created to hold the manifest and other general information about the JAR).

Thomas P. Ventimiglia

Introduction to Computer Science Last modified: Wed Jul 30 16:25:37 EDT 2003