HMI Resources and Data

Java programs often need to read data files and resource files. There are a few reasons why this is not completely straightforward:

- programs need to run on all kind of systems, not just on the machine where it was developed. Consequently, you cannot assume that there is a completely fixed file/directory structure. We discuss our resource strategy in section 1.1
- A more challenging situations is development via webstart. You simply cannot control the locations of resources and data files on systems that start your program via webstart. We discuss special requirements for webstart resources and data in section 1.4
- We have a lot of *shared* resources. You must take some measures to read such resources. We discuss these issues in section 1.2
- Although less variable, resource do change over time, just like program code. Sometimes, you must use one *particular* version in combination with some particular program. This is the topic of section 1.3

1.1 Resource files

Data that is needed by Java programs comes in two varieties:

1. Data like icons, images and configurations files. This data tends to be fixed, and is is strongly bound to the program itself. You cannot run the program when this data is "missing". This type of data is called *resource* data, or "*resources*"
2. Data that can be read or written by the program, but that is variable, and not essential for the proper functioning of the program. Like the document you open with word processor, or the BML document that you want to play with our BML realizer. We simply call this "*data*", to distinguish it from resources.

This chapter deal primarily with resources since it is more difficult to handle this properly than "data". A few remarks:

- It is bad style to read resource data from fixed file locations like:

  ```
  C:\Javaprojects\myproject\resource\config.xml
  ```

  The reason is simple: not everyone who wants to use your program has the same `C:` disk structure, and maybe not even a `C:` disk at all, in case of a Mac or Linux system. Even on Windows systems, it is annoying that other users *must* have a "`C:\Javaprojects` directory.

- When you use Java webstart to deploy your system, then it is simply not possible to refer to the local file system except in very limited ways.

Java has a neat way of reading from (not writing to) resource files via the `getResourceAsStream` methods from the Java `Class` and `ClassLoader` classes. We use this method heavily in the Hmi packages and so we explain the "rules" of this game:

1. resource data is accessed via the Java *classpath* in use when the program is running. Basically, every directory on the classpath is, potentially, a root directory for searching resource files.

2. The simplest way to use this is to call the `getResourceAsStream` method for a `ClassLoader`. You do this like so:

   ```
   getClass().getClassLoader().getResourceAsStream("myresource-path");
   ```

   Here, "`myresource-path`" should be a simple filename or a more complex path like "`dir_00/dir_11/.../dir_n/file`. This path is appended to each of the directories on the classpath in turn, until the resource file is found.

   The alternative method is to use the same method from `Class`, not `ClassLoader`:

   ```
   getClass().getResourceAsStream("myresource-path");
   ```

   This works similar, but the rules are a little different in this case:

   - When `myresourcepath` begins with a '/' character it is appended to directories on the classpath, as described above for `ClassLoader`.
   - But when `myresourcepath` is a file or path *not* starting with '/', then the method searches for a file name of the form:

     `modified_package_name/myresourcepath,`

     where `modified_package_name` is the package name of this object with '/' substituted for '.' . In itself this is a neat way of *organizing* your resources: each resource file resides in a subdirectory that "matches" the package name of the class that reads the resource.

Sometimes, you want to read a resource file from within a *static* method. In this case, you cannot use "`getClass().getClassLoader().getResourceAsStream`", since a static method cannot use non-static methods like `getClass`. In this case an elegant way to get a Class or ClassLoader is to use idiom like: `<full-classname>.class.getClassLoader()`. For example:

```
hmi.graphics.collada.scenegraph.ColladaReader.class.getClassLoader()
  .getResourceAsStream("my-collada-resource.dae");
```

1.2 Project Resources and Shared resources

The resource reading methods from section 1.1 are very useful, but you need some organization of your resource files. In our Hmi projects, we use a project directory structure as follows:

```
   project-directory
       src
       docs
       lib
       resource
       test
           src
           lib
           resource
...
```

The important directories here are the two `resource` directories: one at the top-level, for "ordinary" resources, and a second one within the `test` directory, for resources need only for "junit testing". The `resource` directories are automatically included on the Java classpath when using our `ant`-based build system. If you use a tool like Eclipse or Netbeans, you should include these directories on the classpath as well. The convention is that all resources used by a project are within these two `resource` directories. This ensures that the `getResourceAsStream` methods can always find your resources. The only exception is that sometimes we want to *share* resource files between different projects, especially if they are very large. (If they are not that large, consider simply *copying* resource files into the `resource` directory of your own project). For sharing, you need to ensure that the correct resource directories are included on the Java classpath. For our `ant`-based build system, this works via build properties, that you should set within the project's `build.properties` file (located in the top-level directory of your project). The relevant property is called "`resource.path`". As an example consider this line inside `build.properties` file for a virtual-humans project:

```
resource.path=${shared.resources}/HmiHumanoidEmbodiments/resource;\
```

```
${shared.resources}/DefaultShaders/resource;\
${shared.resources}/Shared3DModels/resource
```

The three directories included on this `resource.path` are added to the classpath, just like the project's own `resource` directory.
(Note: the '\' characters at the end of the first two lines are necessary to split the command over several lines)
The `${shared.resources}` property is an `ant` property that refers to the `${shared.project.root}/HmiResource` directory, that is, the `HmiResource` directory within the shared project directory. The latter (i.e. `${shared.project.root}`) is (by definition) the directory where the `HmiShared` directory resides. Our "standard" `ant build.xml` file redirects to the build file in `HmiShared`:

```
<import file="../../HmiShared/ant/build.xml" />
```

This shared build file infers (from its own location) the location of the shared project directory and all other places defined relative to the shared project directory. It defines properties like `${shared.project.root}` accordingly.

1.3 Resource management

1.4 Webstart resources and data