

## HOWTO: Build a Multi-component Application

Article ID: 000022

Last Revised On: 28-Oct-2004

The information in this article applies to:

- Excelsior JET since version 3.6 (both Windows and Linux versions)

### SUMMARY

This article describes the process of using Excelsior JET to create a multi-component application, that is, an application consisting of an executable and one or more dynamic libraries, possibly shared with other multi-component applications.

Two JavaHelp 2.0\_01 sample programs are used as an example.

Control files for this article may be found [here](#).

### INTRODUCTION

#### Multi-component applications

Suppose that you need to compile several applications which share pieces of code, e.g. they use the same third-party library or an API you wrote. An example of such a relationship is shown in Figure 1:

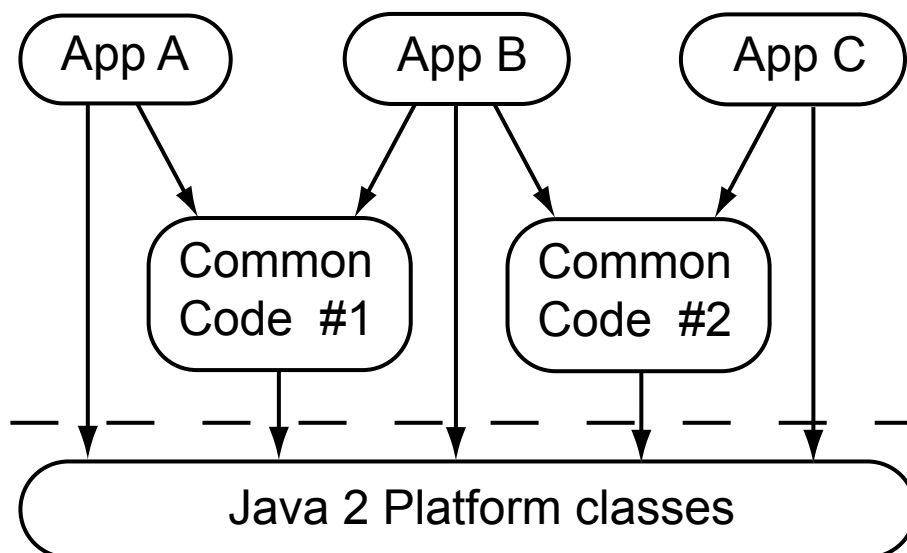


Fig. 1.

You can determine the parts common to two or more applications and put them into *dynamic libraries* (Windows DLLs or Linux shared objects). Dynamic libraries are shared between processes at the operating system level, implying the following important benefits:

- Applications that use common libraries use less memory when running at the same time, thanks to code sharing.
- Application start-up is much faster if some of its dynamic libraries are already loaded by another process.
- The total size of native code can be reduced dramatically if your applications contain large pieces of common code. This also means that the compilation process takes less time.

Thus, if you place common code into dynamic libraries as it is shown in Figure 2, each application would consist of several executable components.

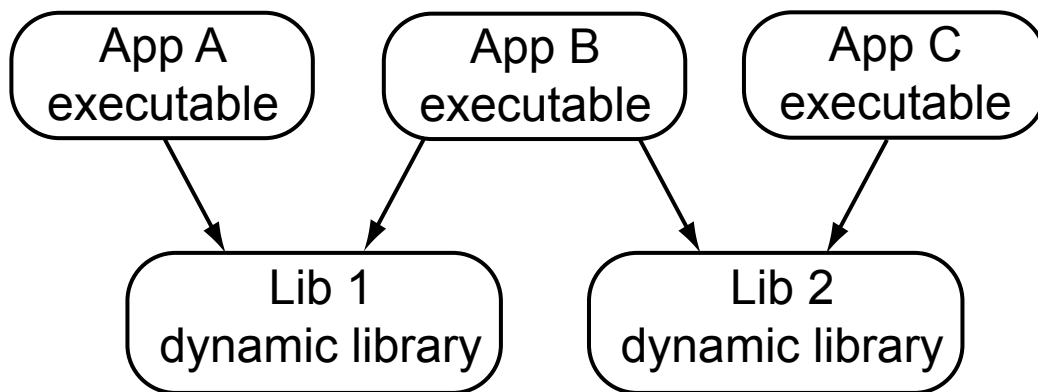


Fig. 2.

Applications that consists of multiple executable components (main executable and one or more dynamic libraries) are called *multi-component*.

Technically, any application compiled with Excelsior JET is multi-component: JET Setup compiles the Java 2 platform classes into a set of dynamic libraries when creating a profile, and then you only compile your application's classes into an executable that requires one or more of those dynamic libraries.

## Hidden problems

Usually, each component is made of one or more jars that represent a complete Java library (API).

There are several limitations you should be aware of before breaking down your applications into components:

1. Each compiled class must appear in exactly one component. Otherwise you will get components that duplicate common code, thus losing all benefits mentioned above. Moreover, the application may work incorrectly. See JET User's Guide, Chapter "Dynamic Linking", section "Multi-component applications" for instructions on how to overcome the problem with duplicated classes.
2. The breakdown of the set of applications' jars into components is **not arbitrary** as it should obey the following rules:
  - As an implication of the previous rule, a jar must not belong to more than one compilation set.

- Cyclic import between components is not allowed. For example, if classes from `A.jar` reference classes from `B.jar` and, at the same time, `B.jar` uses `A.jar`, these jars must be in the same compilation set.

So be prepared the model “one jar — one dynamic library” will not work if jars have cyclic interdependencies.

3. If you have your applications split into components and rebuild a component on which other components depend, it would be necessary to rebuild the dependent components as well.

For example, considering the breakdown shown on Figure 2, if you rebuild the dynamic library `Lib 1`, you also have to rebuild the executables `App A` and `App B`.

## INSTRUCTIONS

In this section, we will use two JavaHelp samples to illustrate the process of setting up the build environment for multi-component applications.

Currently, the JET Control Panel does not support creating multi-component applications so you have to use the command line interface to the JET compiler. However, the JET Control Panel can assist you at the first steps of creating a multi-component application.

### Analysis

Suppose you have installed the JavaHelp API 2.0\_01 package into the directory `JH-dir`, and your goal is to compile two samples: `JH-dir/demos/bin/hsvviewer.jar` and `JH-dir/demos/bin/idedemo.jar`. Let’s call these samples `HsViewer` and `IdeDemo` respectively.

First of all, you need to determine which jars are necessary for these samples. This information can be obtained by looking at the contents of the sample jars’ manifest files (`META-INF/manifest.mf`):

- `HsViewer` uses:  
`JH-dir/javahelp/lib/jh.jar`
- `IdeDemo` uses:  
`JH-dir/javahelp/lib/jh.jar`  
`JH-dir/demos/lib/classviewer.jar`  
`JH-dir/demos/hsjar/idehelp.jar`  
`JH-dir/demos/hsjar/apidoc.jar`

As can be seen, both samples use `JH-dir/javahelp/lib/jh.jar`, therefore the following decomposition looks quite natural:

1. `jh` dynamic library:  
`JH-dir/javahelp/lib/jh.jar`
2. `HsViewer` executable:  
`JH-dir/demos/bin/hsvviewer.jar`
3. `IdeDemo.exe` executable:  
`JH-dir/demos/bin/idedemo.jar`  
`JH-dir/demos/lib/classviewer.jar`  
`JH-dir/demos/hsjar/idehelp.jar`  
`JH-dir/demos/hsjar/apidoc.jar`

## Consistency check

The purpose of this check is to ensure that the following conditions are met:

- Each application does not contain duplicated classes. Otherwise, consult Excelsior JET User's Guide [1] to learn what to do in this situation.
- Each application works properly if compiled in a single-component manner.

To check the JavaHelp samples, first create a project for the HsViewer sample using the JET Control Panel. On the **Classes** page, add *JH-dir/demos/bin/hsvviewer.jar* and force all classes into the compilation set. The JET Control Panel will add *JH-dir/javahelp/lib/jh.jar* automatically by analyzing the manifest of *hsvviewer.jar*.

In this particular sample you will see no warnings about class duplication. If warnings about unresolved import dependencies are displayed, you can ignore them, provided the sample works under the HotSpot JVM flawlessly.

Finally, go through the remaining pages to set up the project, compile it and make sure that the compiled HsViewer demo works properly.

Repeat the above instructions for the IdeDemo sample.

**Note:** The created project files **cannot** be used for subsequent compilation of dynamic libraries and executables.

## Creating the project files

Create an empty directory that will be used for building the executables. In that directory, create JET projects (plain text files with the extension `.prj`, one project per component) as follows:

Insert the following text to `jh.prj` file (percent signs “%” mark comments):

```
-OUTPUTNAME=jh
-GENDLL+
% The OUTPUTNAME equation sets the name of the output executable
% file (without extension), whereas -GENDLL+ tells the compiler
% to produce a DLL (shared library). So the resulting executable
% will be named jh.dll on Windows and jh.so on Linux.

-BINDRESOURCES+
% Resources are images, audio clips, and whatever data files that
% may appear in jars in addition to Java classes. This option instructs
% the compiler to bind those resources to the resulting executable
% so that the jar file is not needed at run time.

-CLASSABSENCE=IGNORE
-IGNOREMEMBERABSENCE+
% These two options force the compiler to ignore possible compilation
% set inconsistencies, namely references to absent classes, fields
% and methods.

-LOOKUP=*.obj=./obj_$(OUTPUTNAME)
% This lookup statement tells the compiler where to place generated
% object files. $(OUTPUTNAME) expands to the value of OUTPUTNAME,
% which was set above to "jh", so object files will be placed to the
```

```
% obj_jh subdirectory of the current directory.

-LOOKUP=*.jar=$(JHDIR)/javahelp/lib
% This lookup directive informs the compiler where to look
% for .jar files. To make it work you should either set the
% environment variable JHDIR to proper directory or replace
% "$(JHDIR)" with the proper path.

!module jh.jar
% This directive specifies the jar to be compiled.
```

Now, copy and paste the following text to `IdeDemo.prj` file:

```
-OUTPUTNAME=IdeDemo
-BINDRESOURCES+
-CLASSABSENCE=IGNORE
-IGNOREMEMBERABSENCE+

-LOOKUP=*.sym=./sym_$(OUTPUTNAME)
-LOOKUP=*.bod=./bod_$(OUTPUTNAME)
-LOOKUP=*.obj=./obj_$(OUTPUTNAME)

-LOOKUP=*.jar=$(JHDIR)/demos/hsjar;$(JHDIR)/demos/bin

-MAIN=sunw/demo/idedemo/ApiDemo
% If more than one class in the compilation set has a method
%
%   public static void main (String[])
%
% the compiler needs to know which of them must be designated
% the executable's entry point. The MAIN equation specifies
% the class which main method will be called upon executable
% startup.

!module idedemo.jar
!module idehelp.jar
!module classviewer.jar
% These directives specify jars to be compiled.

!module jh.dll
% This directive tells the compiler that 'jh' dynamic library
% is required by the application. Make sure to replace 'jh.dll'
% with 'libjh.so' on Linux.
```

Finally, copy `IdeDemo.prj` to `HsViewer.prj` and edit the `-OUTPUTNAME=`, `-LOOKUP=*.jar=`, `-MAIN=` and `!module` directives as appropriate. You will find the main class name in the manifest file of `hsviewer.jar`.

If you did all right, the build directory should now contain three project files similar to those found in the [attachment](#) to this article.

## Compilation

At the command prompt, go to the build directory and compile the projects as follows:

```
jc =p jh.prj
jc =p HsViewer.prj
jc =p IdeDemo.prj
```

**Note:** The order of compilation is important, that is, `jh.prj` must be compiled before the other two projects.

Now you can run the resulting executables.

## Control files for this article

This section contains information about the contents of the attached zip-archive.

The archive includes a set of project files to compile the components described in this article and the build and cleanup scripts.

In order to build the JavaHelp samples you should do the following:

1. Unzip the attached archive.
2. Correct path to JavaHelp 2.0 in the build script.
3. Run the bulid script to initiate the compilation process.

## REFERENCES

1. Excelsior JET User's Guide, Chapter "Dynamic Linking", Sections "Multi-component applications" and "Using multiple components".  
<http://www.excelsior-usa.com/doc/jc.html>
2. JavaHelp System.  
<http://java.sun.com/products/javahelp/>