Compiling Eclipse Applications for Windows With GCJ/MinGW [1, 2]

by David Scuse

Department of Computer Science, University of Manitoba, Winnipeg, Manitoba, Canada

Last revised: Monday, July 14, 2003

Overview:

In this document, we examine the creation of native (Windows) executables for Eclipse applications (not for Eclipse itself). While we have concentrated on Windows aspects, the equivalent systems should work without any difficulty on Linux. We will eventually test the generation of native Linux executables.

The advantage of compiled Eclipse application executables is that they do not require the JRE, they are self-contained except for the one swt dll. The applications can be distributed in zip format and can be moved around on a machine without requiring changes to the registry (there is no information saved in the Windows registry).

GCC and MinGW:

To compile Java applications to run on Windows, download the MinGW build of GCC/GCJ 3.3 for win32 from http://www.thisiscool.com/gcc_mingw.htm. This build uses a slightly older version of Eclipse (Eclipse 2.0.2) but it worked without any problems on the applications that we tested. This build includes a build of the SWT libraries so that the entire Eclipse application can be compiled. (The Swing libraries have not yet been ported to GCC/GCJ.) Special thanks are due to Mohan Embar and Ranjit Mathew for their work in building this win32 version.

We tried the GCC 3.4 build of 29 June 2003 (which includes Eclipse 2.1) but ran into some difficulties that could not be easily identified so in this document, we are using GCC/GCJ 3.3 and Eclipse 2.0.2 for now.

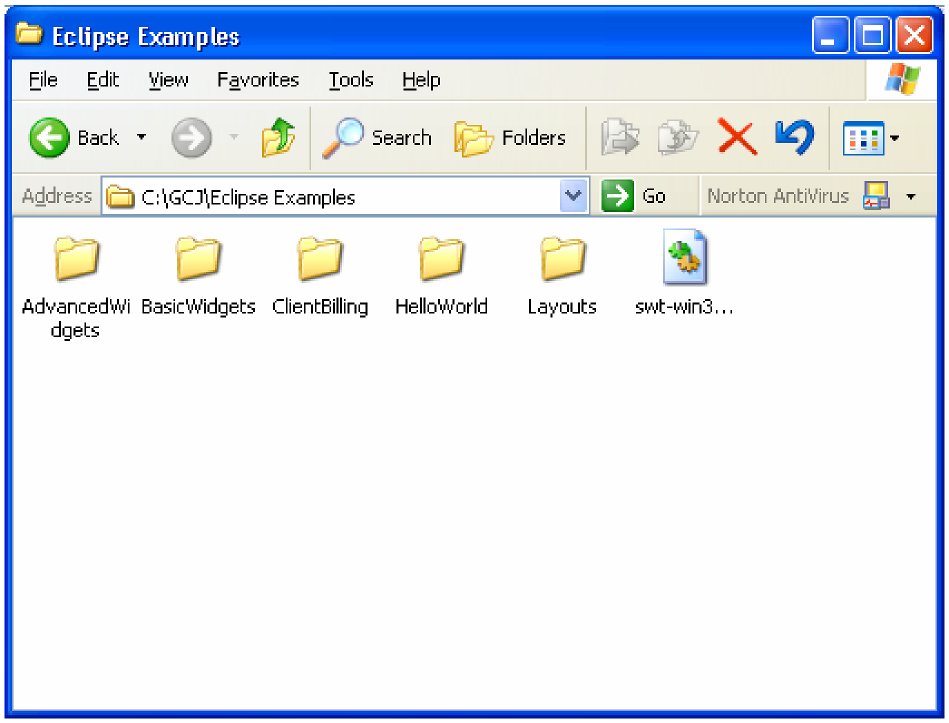
[1] This work was funded by an IBM Eclipse Innovation Grant.

[2] © David Scuse

Once the compiler system has been installed, you are ready to go. Compiles are typically defined in scripts instead of from the command prompt. In this document, we have used Windows command scripts (.bat files) since they are likely more familiar to Windows users than shell scripts. However, the same scripts could be defined using the Windows Bash port msys, available at http://www.mingw.org/. By using shell scripts instead of Windows command scripts, the same script could also be run under Linux. A complete Windows command script is included with each example.

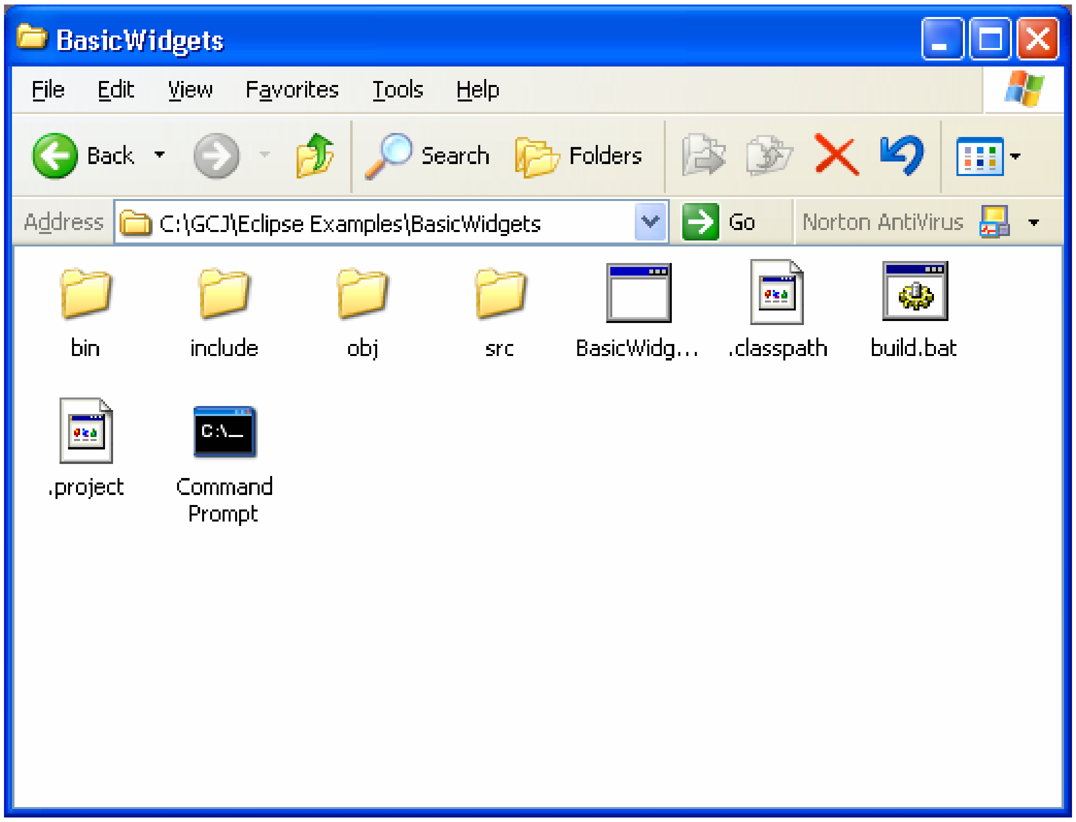
After an Eclipse application has been compiled, a standard Windows exe file is generated. This file can be executed by double-clicking on it. The only Eclipse file that must be present for the exe to run is the swt dll for the associated Eclipse build – in these examples, that is swt-win32-2052.dll. This dll must be available somewhere on the system path; putting it the System32 folder makes it available to all programs. (See the first tutorial, Installing Eclipse, for more information on the swt dll.)

All of the applications that were developed in the earlier tutorials have been included in the associated project folder. Note that the required swt dll has also been included in the folder. This folder can be unzipped to any desired location. Note that it is not necessary to have Eclipse installed on the machine for the applications to run. It is also not necessary to have installed the MinGW build of GCC/GCJ 3.3 to run the exe's but it is necessary to install MinGW if you want to compile any other applications.



Basic Widgets Project:

In this section, we examine the BasicWidgets application that was described in the Basic Widgets tutorial. The folder contains the files shown below. Note that this folder is a copy of the folder used in the Eclipse workspace; it is probably not a good idea to build the executables in the Eclipse workspace folder itself.

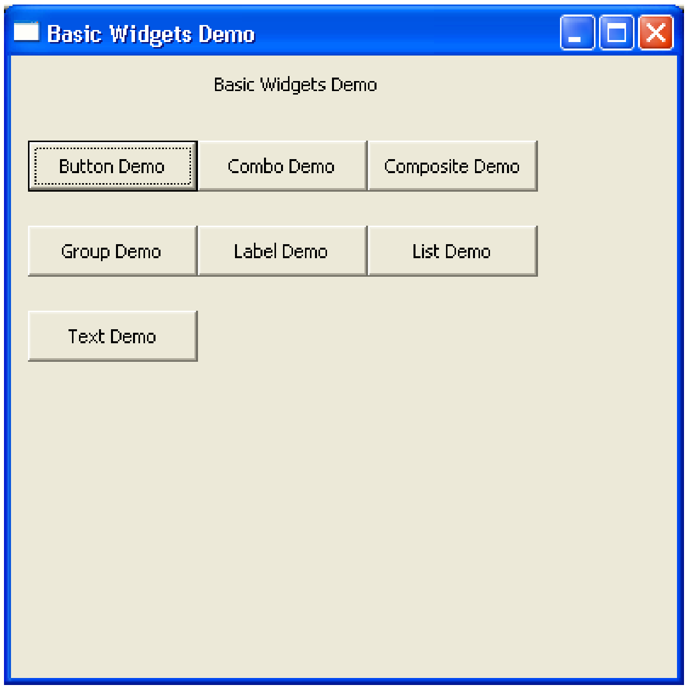


We will not describe the parameters to GCJ; for more information, see the GCC website at http://gcc.gnu.org/ or a GCC book.

The build script shown below initially defines some environment variables. These variables indicate the location of the GCC compiler and the GCC SWT libraries that are required. Each source file is then compiled into the corresponding class files which are stored in the bin folder. These files are then collected into a jar file. The source files are then compiled into object files (*.o) and are stored in the obj folder. The object files are then collected into an object library (*.a). Finally, the application is built by the last gcj command that includes the --main parameter.

```
setlocal
set prompt=?

PATH C:\gcc-3.3\bin;%PATH%
set SWT_BASE_DIR=C:\gcc-3.3\swt\win32
set SWT_GCJ_LIB_DIR=%SWT_BASE_DIR%\lib
set SWT_JAVA_LIB_DIR=%SWT_BASE_DIR%\swt-2052

set MAIN=BasicWidgetsDemo

mkdir bin
mkdir obj
mkdir include

gcj -C -fCLASSPATH=bin;src;%SWT_JAVA_LIB_DIR%\swt.jar -d bin src\BasicWidgetsDemo.java
gcj -C -fCLASSPATH=bin;src;%SWT_JAVA_LIB_DIR%\swt.jar -d bin src\ButtonDemo.java
gcj -C -fCLASSPATH=bin;src;%SWT_JAVA_LIB_DIR%\swt.jar -d bin src\ComboDemo.java
gcj -C -fCLASSPATH=bin;src;%SWT_JAVA_LIB_DIR%\swt.jar -d bin src\CompositeDemo.java
gcj -C -fCLASSPATH=bin;src;%SWT_JAVA_LIB_DIR%\swt.jar -d bin src\GroupDemo.java
gcj -C -fCLASSPATH=bin;src;%SWT_JAVA_LIB_DIR%\swt.jar -d bin src\LabelDemo.java
gcj -C -fCLASSPATH=bin;src;%SWT_JAVA_LIB_DIR%\swt.jar -d bin src\ListDemo.java
gcj -C -fCLASSPATH=bin;src;%SWT_JAVA_LIB_DIR%\swt.jar -d bin src\TextDemo.java

cd bin
jar cf %MAIN%.jar
move %MAIN%.jar ..\include
cd ..

gcj --classpath=obj;bin;%SWT_JAVA_LIB_DIR%\swt.jar –c src\BasicWidgetsDemo.java -o obj\BasicWidgetsDemo.o
gcj --classpath=obj;bin;%SWT_JAVA_LIB_DIR%\swt.jar -c src\ButtonDemo.java -o obj\ButtonDemo.o
gcj --classpath=obj;bin;%SWT_JAVA_LIB_DIR%\swt.jar -c src\ComboDemo.java -o obj\ComboDemo.o
gcj --classpath=obj;bin;%SWT_JAVA_LIB_DIR%\swt.jar -c src\CompositeDemo.java -o obj\CompositeDemo.o
gcj --classpath=obj;bin;%SWT_JAVA_LIB_DIR%\swt.jar -c src\GroupDemo.java -o obj\GroupDemo.o
gcj --classpath=obj;bin;%SWT_JAVA_LIB_DIR%\swt.jar -c src\LabelDemo.java -o obj\LabelDemo.o
gcj --classpath=obj;bin;%SWT_JAVA_LIB_DIR%\swt.jar -c src\ListDemo.java -o obj\ListDemo.o
gcj --classpath=obj;bin;%SWT_JAVA_LIB_DIR%\swt.jar -c src\TextDemo.java -o obj\TextDemo.o

cd obj
ar -crs lib%MAIN%.a *.o
move lib%MAIN%.a ..\include
cd ..

gcj --main=%MAIN% --classpath=.;%SWT_JAVA_LIB_DIR%\swt.jar -o %MAIN%.exe obj\*.o include\*.a -L%SWT_GCJ_LIB_DIR% -lswt

endlocal
```

When the exe is run, the following window is displayed. The window is identical to the window generated within Eclipse.



If you examine BasicWidgetsDemo.exe, you will find that it is 5.91 MB in size. This is significantly larger than the equivalent C/C++ exe but the exe includes the run-time tools (such as garbage collection) that are required to support the Java environment.

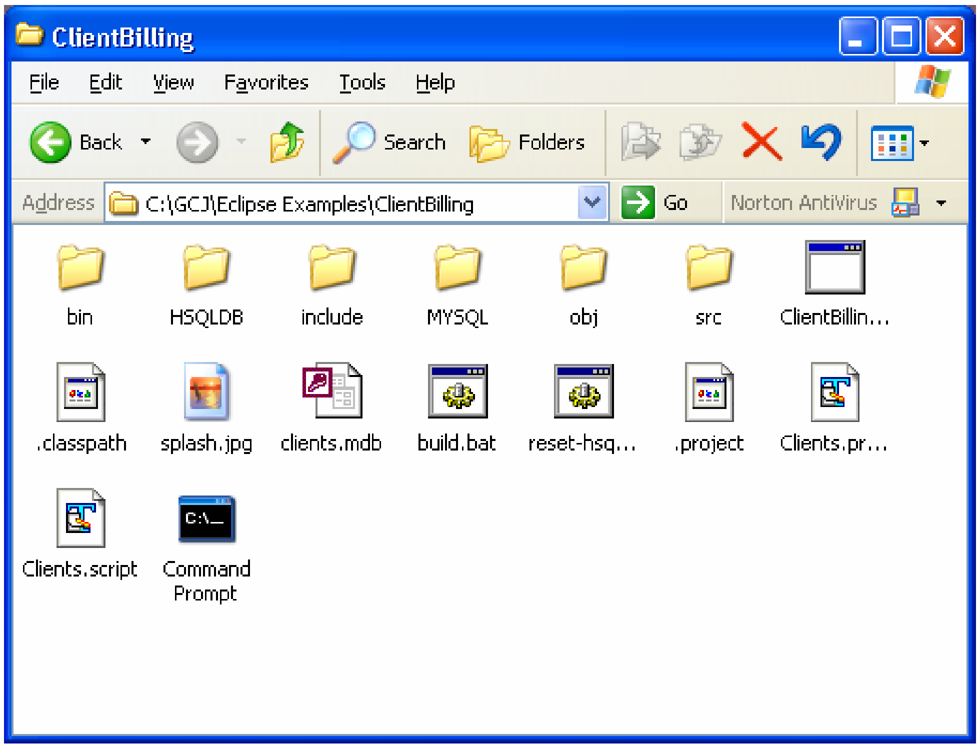
JDBC Driver Compilation:

Most of the applications in the associated zip file can be compiled without any difficulties. However, the ClientBilling application uses a database and this causes some difficulties for GCC/GCJ since JDBC drivers compiled with GCJ are not readily available.

We chose to try the mysql and hsqldb systems since they are both written in Java and are both open source.

I would like to thank Dave Orme of Advanced Systems Concepts, Inc. and Kirk Vogen of IBM for providing suggestions about the choice of JDBC drivers. I would also like to thank Erik Poupaert for modifying the mysql driver so that it could be compiled with GCJ and for providing a shell script to do the compile.

The folder for the ClientBilling application is shown below.



The include folder contains the following files that were created by compiling mysql and hsqldb with GCJ: com-mysql-jdbc.jar; libcom-mysql-jdbc.a; org-hsqldb.jar; and liborg-hsqldb.a. (The jar files contain the compiled .class files and the "a" files contain the compiled .o files.)

The jdbc driver to be used is identified in the ClientBillingDB.java file. Prior to compiling the application, the desired database management system is selected by adding or removing comments in the source code.

```
//Load the hsqldb driver
Class.forName("org.hsqldb.jdbcDriver");
url = "jdbc:hsqldb:Clients"; // stored on disk mode
//Open the connection
conn = DriverManager.getConnection(url, "SA", "");
System.out.println("Opened HSQLDB database.");

//Load the MySQL driver
//Class.forName("com.mysql.jdbc.Driver");
//url = "jdbc:mysql://localhost/clients";
//Open the connection
//conn = DriverManager.getConnection(url, "root", "");
//System.out.println("Opened MYSQL database.");
```

Because the jdbc driver is loaded at dynamically, it is not automatically included when GCJ performs the compile and link of the routines in the application. There are various methods that can be used to force GCJ to include routines that are not referenced statically but the one that we have chosen is to include an additional source file, ForceInclude.java, that references the jdbc driver and any other classes that are loaded dynamically.

The following ForceInclude.java file was used with ClientBilling. Note that the Calendar and LocaleInformation classes must also be loaded. In this example, both the mysql and the hsqldb drivers have been included; normally, only one database is used so only one driver would be required.

```
public class ForceInclude {

   private static final Class class1 = gnu.java.locale.Calendar.class;
   private static final Class class2 =
                              gnu.java.locale.LocaleInformation.class;
   private static final Class class3 = com.mysql.jdbc.Driver.class;
   private static final Class class4 = org.hsqldb.jdbcDriver.class;

   ForceInclude()  {
   }
}
```

Once the application has been built, it can be executed. If mysql is used, the mysql server must first have been installed (http://www.mysql.com/) and the database must be created. A script that creates the database is included in the MYSQL folder in the ClientBilling folder.

If hsqldb (http://hsqldb.sourceforge.net/) is used, no downloads are required since the jdbc driver has already been compiled with the application. An earlier version of hsqldb (1.7.1) did not close the database correctly after the application terminated. This problem appears to have been corrected with version 1.7.2 Alpha M (22 JAN 2003). One difference between the version of hsqldb used with Eclipse is that it was compiled with JDK 1.3; this version did not compile correctly with GCJ but changing the source to use JDK 1.4 fixed the compile problems. (HSQLDB includes a build routine “switchToJDK14.bat” that modifies the source files so that they are compatible with a specific version of the JDK.)

Although we used the jdbc-odbc driver in the original Eclipse application, we do not yet have an odbc driver compiled with GCJ for Windows.

GCJ Options

```
gcj --help

Usage: gcj [options] file...
Options:
  -pass-exit-codes         Exit with highest error code from a phase
  --help                   Display this information
  --target-help            Display target specific command line options
  -dumpspecs               Display all of the built in spec strings
  -dumpversion             Display the version of the compiler
  -dumpmachine             Display the compiler's target processor
  -print-search-dirs       Display the directories in the compiler's search
                           path
  -print-libgcc-file-name  Display the name of the compiler's companion library
  -print-file-name=<lib>   Display the full path to library <lib>
  -print-prog-name=<prog>  Display the full path to compiler component <prog>
  -print-multi-directory   Display the root directory for versions of libgcc
  -print-multi-lib         Display the mapping between command line options and
                           multiple library search directories
  -print-multi-os-directory Display the relative path to OS libraries
  -Wa,<options>            Pass comma-separated <options> on to the assembler
  -Wp,<options>            Pass comma-separated <options> on to the
                           preprocessor
  -Wl,<options>            Pass comma-separated <options> on to the linker
  -Xlinker <arg>           Pass <arg> on to the linker
  -save-temps              Do not delete intermediate files
  -pipe                    Use pipes rather than intermediate files
  -time                    Time the execution of each subprocess
  -specs=<file>            Override built-in specs with the contents of <file>
  -std=<standard>          Assume that the input sources are for <standard>
  -B <directory>           Add <directory> to the compiler's search paths
  -b <machine>             Run gcc for target <machine>, if installed
  -V <version>             Run gcc version number <version>, if installed
  -v                       Display the programs invoked by the compiler
  -###                     Like -v but options quoted and commands not executed
  -E                       Preprocess only; do not compile, assemble or link
  -S                       Compile only; do not assemble or link
  -c                       Compile and assemble, but do not link
  -o <file>                Place the output into <file>
  -x <language>            Specify the language of the following input files
                           Permissible languages include: c c++ assembler none
                           'none' means revert to the default behavior of
                           guessing the language based on the file's extension
```

Note: gcj -v –help generates additional options specific to the version being used.

Note: adding the –mwindows option to the final step that generates the exe will remove the black console window from behind the application.

```
Options starting with -g, -f, -m, -O, -W, or --param are automatically
passed on to the various sub-processes invoked by gcj.  In order to
pass other options on to these processes the -W<letter> options must be
used.
```

Acknowledgements:

Again, special thanks are due to Mohan Embar and Ranjit Mathew for their work in building this win32 version; to Dave Orme of Advanced Systems Concepts, Inc. and Kirk Vogen of IBM for providing suggestions about the choice of JDBC drivers, and to Erik Poupaert for modifying the mysql driver so that it could be compiled with GCJ and for providing a shell script to do the compile. (See also Erik's Freestyler Toolkit in the References section.)

References:

Kirk Vogen, *Create native, cross-platform GUI applications: How GCJ, Linux, and the SWT come together to solve the Java UI conundrum*, 2002.

http://www-106.ibm.com/developerworks/java/library/j-nativegui/index.html

Kirk Vogen, *Create native, cross-platform GUI applications, revisited: An updated look at GCJ and the SWT*, 2003.

http://www-106.ibm.com/developerworks/java/library/j-nativegui2/

Erik Poupaert, *The Freestyler Toolkit*, 2003. (A more elaborate and complete system for creating and compiling Java/SWT applications.)

http://www.freestyler-toolkit.org/

Additional information on compiling with GCJ can be found using Google.