make files

Make files are a technique by which disparate software tools may be marshalled to work on the construction of a single software project.

Command line

Make files work using command line utilities so that it will be necessary to learn how to compile programs using the command line rather than using and IDE such as Eclipse.

Compiling

- SYNOPSIS

  javac [options] [sourcefiles] [@argfiles]

- PARAMETERS

  Arguments may be in any order.

  options Command line options.

  sourcefiles One or more source files to be compiled (such as MyClass.java).

Dependencies in javac

Suppose I type

javac Gamma.java

and class Gamma uses or extends classes Alpha and Beta

- Search produces a class file but no source file: javac uses the class file.
- Search produces a source file but no class file: javac compiles the source file and uses the resulting class file.
- Search produces both a source file and a class file: javac determines whether the class file is out of date. If the class file is out of date, javac recompiles the source file and uses the updated class file. Otherwise, javac just uses the class file.

Dependencies not caught

- Suppose both class Alpha and class Delta were declared in file Delta.java
- In this case javac will not detect that Delta.java has to be compiled to produce file Alpha.class
- This is an example of the sort of dependency that Makefiles are useful for.

Compiler options

Options are all preceded by a – sign

-classpath dirlist

Sets the user class path, overriding the user class path in the CLASSPATH environment variable. If neither CLASSPATH or -classpath is specified, the user class path consists of the current directory.

-d directory

Sets the destination directory for class files. The destination directory must already exist; javac will not create the destination directory. If a class is part of a package, javac puts the class file in a subdirectory reflecting the package name, creating directories as needed.

Running java programs

- java [options] class [argument ...]
- java [options] -jar file.jar [argument ...]

PARAMETERS

options	Command-line options.
class	Name of the class to be invoked.
file.jar	Name of the jar file to be invoked. Used only with the -jar option.
argument	Argument passed to the main function.

program to run

By default, the first non-option argument is the name of the class to be invoked. A fully-qualified class name should be used. If the -jar option is specified, the first non-option argument is the name of a JAR archive containing class and resource files for the application, with the startup class indicated by the Main-Class manifest header.

Examples

java Alpha.class

Runs class Alpha in the current directory

java uk.ac.gla.dcs.mscit.Alpha.class

Runs file uk/ac/gla/dcs/mscit/Alpha.class relative to the current classpath

Options

- -classpath classpath
- -cp classpath

Specifies a list of directories, JAR archives, and ZIP archives to search for class files. Class path entries are separated by colons (:). Specifying -classpath or -cp overrides any setting of the CLASSPATH environment variable

Example

java -cp foo.jar:fi.jar:fo.jar:fum.jar beanstalk.Jack.class

Search for beanstalk/Jack.class in the jar files specified

More options

- -verbose:class

  Displays information about each class loaded.
- -verbose:gc

  Reports on each garbage collection event.
- -version

  Displays version information and exit.
- -showversion

  Displays version information and continues.
- -help Displays usage information and exit.

Controlling Virtual Memory space

- -Xmxn

  Specifies the maximum size, in bytes, of the memory allocation pool. This value must be a multiple of 1024 greater than 2 MB. Append the letter k or K to indicate kilobytes or the letter m or M to indicate megabytes. The default value is 64MB.

Examples:

- -Xmx83886080
- -Xmx81920k
- -Xmx80m

On Linux platforms, the upper limit is approximately 2000m minus overhead amounts. On windows 1000m - overhead.

ubuntu

using jar

- Create jar file

  jar c[v0M]f jarfile [-C dir] inputfiles [-Joption]

  jar c[v0]mf manifest jarfile [-C dir] inputfiles [-Joption]
- Extract jar file

  jar x[v]f jarfile [inputfiles] [-Joption]
-

manifest

- Pre-existing manifest file whose name: value pairs are to be included in MANIFEST.MF in the jar file. The m option and filename manifest are a pair -- if either is present, they must both appear. The letters m and f must appear in the same order that manifest and jarfile appear.

Example of a manifest file

- $ cat *.mf

Manifest-Version: 1.0

Main-Class: ilcg.Pascal.PascalCompiler

using make

to use make simply type

make

on the command line.

make then looks for a file called “makefile” which tells it what to do

dependencies

The following is the generic target entry form:

```
# comment

# (note: the <tab> in the command line is necessary
  for make to work)

target:  dependency1 dependency2 ...

      <tab> command
```

•

Example

```
#
# target entry to build program executable from program and
   mylib
# object files  using the gcc compiler
#
program: program.o mylib.o
     gcc -o program program.o mylib.o
```

Example makefile

```
# define a makefile variable for the java compiler

JCC = javac

# typing 'make' will invoke the first target entry in the
  makefile

# (the default one in this case)

default: Average.class Convert.class Volume.class

Convert.class: Convert.java

	$(JCC) $(JFLAGS) Convert.java

Volume.class: Volume.java

	$(JCC) $(JFLAGS) Volume.java

# To start over from scratch, type 'make clean'.

# Removes all .class files, so that the next make rebuilds them

clean:

	rm *.class
```

another dependency

```
# this target entry builds the Average class
# the Average.class file is dependent on the Average.java file
# and the rule associated with this entry gives the command to create it
#
Average.class: Average.java
        $(JCC) $(JFLAGS) Average.java
```

type dependencies

- There exist systematic dependencies
- for example
- .class files are always produced from .java files
- .o files are usually produced from .c files
-
- You can tell make about systematic dependencies

dependencies between file types

```
JC = javac
.SUFFIXES: .java .class
.java.class:
	$(JC) $*.java
CLASSES = \
	Foo.class \
	Blah.class \
	Library.class

Main.class: Main.java $(CLASSES)
```