How to simple make file configuration

For Click : >> Download file Here : Make_file_ch01

The mechanics of programming usually follow a fairly simple routine of editing
source files, compiling the source into an executable form, and debugging the result.
Although transforming the source into an executable is considered routine, if done
incorrectly a programmer can waste immense amounts of time tracking down the
problem. Most developers have experienced the frustration of modifying a function
and running the new code only to find that their change did not fix the bug. Later
they discover that they were never executing their modified function because of some
procedural error such as failing to recompile the source, relink the executable, or
rebuild a jar. Moreover, as the program's complexity grows these mundane tasks can
become increasingly error-prone as different versions of the program are developed,
perhaps for other platforms or other versions of support libraries, etc.
The make program is intended to automate the mundane aspects of transforming
source code into an executable. The advantages of make over scripts is that you can
specify the relationships between the elements of your program to make, and it knows
through these relationships and timestamps exactly what steps need to be redone to

produce the desired program each time. Using this information, make can also opti-
mize the build process avoiding unnecessary steps.
GNU make (and other variants of make) do precisely this. make defines a language for
describing the relationships between source code, intermediate files, and executa-
bles. It also provides features to manage alternate configurations, implement reus-
able libraries of specifications, and parameterize processes with user-defined macros.
In short, make can be considered the center of the development process by providing
a roadmap of an application’s components and how they fit together.
The specification that make uses is generally saved in a file named makefile. Here is a
makefile to build the traditional “Hello, World” program:

```
hello: hello.c
gcc hello.c -o hello
```

To build the program execute make by typing:

```
$ make
```

3

at the command prompt of your favorite shell. This will cause the make program to
read the makefile and build the first target it finds there:

```
$ make
gcc hello.c -o hello
```

If a target is included as a command-line argument, that target is updated. If no com-
mand-line targets are given, then the first target in the file is used, called the default
goal.
Typically the default goal in most makefiles is to build a program. This usually
involves many steps. Often the source code for the program is incomplete and the

source must be generated using utilities such as flex or bison. Next the source is
compiled into binary object files (.o files for C/C++, .class files for Java, etc.). Then,
for C/C++, the object files are bound together by a linker (usually invoked through
the compiler, gcc) to form an executable program.
Modifying any of the source files and reinvoking make will cause some, but usually
not all, of these commands to be repeated so the source code changes are properly
incorporated into the executable. The specification file, or makefile, describes the
relationship between the source, intermediate, and executable program files so that
make can perform the minimum amount of work necessary to update the executable.
So the principle value of make comes from its ability to perform the complex series of
commands necessary to build an application and to optimize these operations when
possible to reduce the time taken by the edit-compile-debug cycle. Furthermore, make
is flexible enough to be used anywhere one kind of file depends on another from tra-
ditional programming in C/C++ to Java, TEX, database management, and more.