The Make utility

software build tool

What Make does

- Automate building of software, even very complex source trees
- Perform tasks using rules written in a **`Makefile`**
- Only does what is needed:
  only update missing or out-of-date outputs ("targets")

Makefile: Target, Dependencies & Actions

```
game: Game.o Board.o
    g++ -o game Game.o Board.o

Game.o: Game.cpp  Board.h
    g++ -c Game.cpp

Board.o: Board.cpp Board.h
    g++ -c Board.cpp
```

A Rule

Dependencies: what target depends on

Target

Action: how to build the target

Running make

```
cmd> make game
g++ -c Game.cpp        (creates Game.o)
g++ -c Board.cpp       (creates Board.o)
g++ -o game Game.o Board.o
```

If you run make again, everything is up to date, so nothing needs to be done:

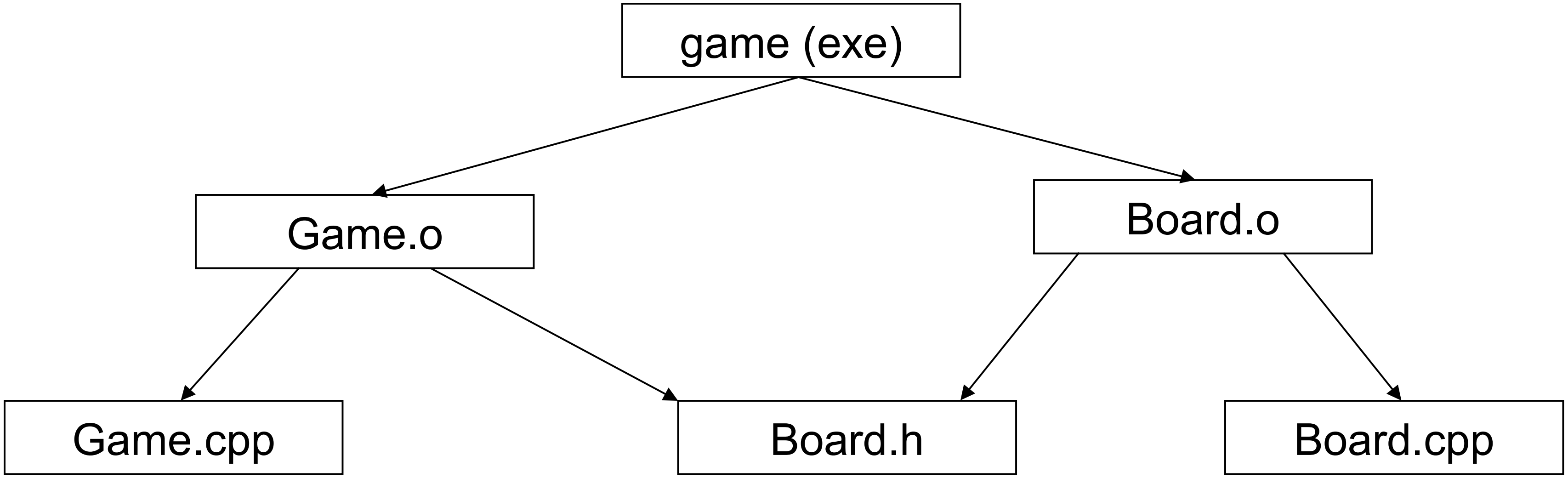
```
cmd> make game
Up to date
```

What if Board class changes?

```
cmd> touch Board.cpp
("touch" command updates modification time)
cmd> make
g++ -c Board.cpp     (recreates Board.o)
g++ -o game Game.o Board.o
```

Notice that make did not compile `Game.cpp`

The rules define a graph



Makefile rules

rules have the following form:

```
target: dependencies ...
<tab>      command
<tab>      command2
<tab>      command3
```

target - usually name of a <u>file</u> that is created by the commands; but it can be any name

```
# remove all object files
clean:
    rm -f *.o
```

Using Variables and Macros

- Simplify rules and reduce redundancy

  `$@` = name of the target

  `$^` = the dependencies

- Rule without using variables:

```
game: Game.o Board.o
    g++ -o game  Game.o Board.o
```

- Same rule using variables:

```
game: Game.o Board.o
  g++ -o $@  $^
```

Predefined variables

CC	Compiler, defaults to cc.
CFLAGS	Passed to $(CC)
LD	Loader command, defaults to ld
LDFLAGS	Passed to $(LD)
$@	Full name of the target
$?	Names of all dependencies which are out-of-date
$^	Names of all dependencies
$<	The name of the current (single) dependency

http://www.gnu.org/software/make/manual/html_node/Implicit-Variables.html

Makefile design

- Do not write rules like:

```
game: Game.cpp Board.cpp Board.h
    g++ -o game Game.cpp Board.cpp
```

requires compiling all files when anything changes

Can We Use Make with Java?

```
TicTacToe/
    Makefile
    src/
    src/tictactoe/     <--- package
         Board.java
         BoardSquare.java
         GameController.java
         GameUI.fxml
         Main.java
         TicTacToeGame.java
```

This does NOT work ('wrong package' error):

```
   cd src/tictactoe
   javac Board.java
```

Makefile for TicTacToe

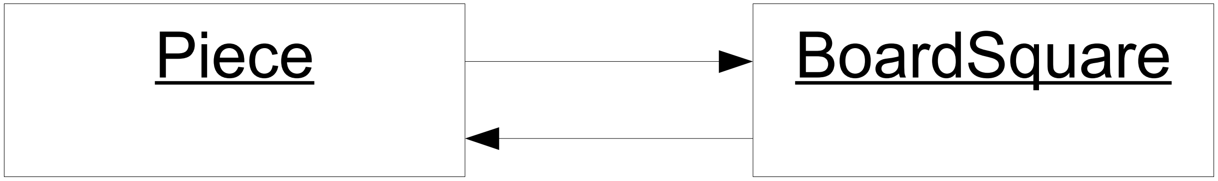
```
JAVAC = javac -sourcepath src
#VPATH = where to find files needed by rules
VPATH = src/tictactoe
CLASSES = Board.class Piece.class ... \
          TicTacToeGame.class Main.class
FXML = GameUI.fxml

tictactoe.jar:  $(CLASSES) $(FXML)
    jar cvf $@ -C src .     # no so good

# rule to make .class file from .java file
%.class: %.java
    $(JAVAC) $<
```

Demo - make tictactoe.jar

```
cmd> cd workspace/ttt
cmd> make clean
cmd> make
```

Problems with Java and Make

- Cyclic Dependencies!
  - Bad design, but it does happen



- Hard to record all deps. in Makefile
- Code is in many directories for packages
- You have to run javac from top-level src dir

Can we Use Make for Python?

Yes! many Python projects use make

Automate any repetitive task:

- running testing and continuous integration
- download or update dependencies (pip)
- create a virtualenv
- initialize a project or database
- clean up

Reference

GNU make tutorial and user guide

https://www.gnu.org/software/make/manual/