Preparations

- CMake: https://cmake.org/install/
- Gradle: https://gradle.org/install/
- Compilers for C++ / Java

Build Systems

CS350 Introduction to Software engineering

Shin Yoo

Compile vs. Build

- You **compile** a single source code file.
- You **build** a software project.
  - Manage dependencies between individual files.
  - Manage external dependencies.
  - Automatically execute test cases.
  - Automatically generate documentations (based on information in source).

Build System

- Tools/frameworks that allow you to process/execute build **programatically.**
- Typically involves a Domain Specific Language (DSL) that can describe:
  - Individual tasks
  - Dependencies between them
  - Ability to invoke external tools (compilers, etc)
  - Sometimes full-fledged language can be used (gradle scripts are written in groovy or kotlin_)

Build Scripts

- The DSL script that can be executed by the build system and actually performs the build.
- Should be part of your source code, and committed to a repository.
- Have you seen `Makefile`, `CMakefile`, or `build.xml`?

Make / Makefile

Stuart Feldman, April 1976 at Bell Labs (Unix version 1.0)

- A basic, default build tool that comes with *nix systems.
- By default, make takes Makefile (unless you specify the input using -f)
- Targets and pre-requisites are all file names; commands are any shell commands.

Makefile Structure

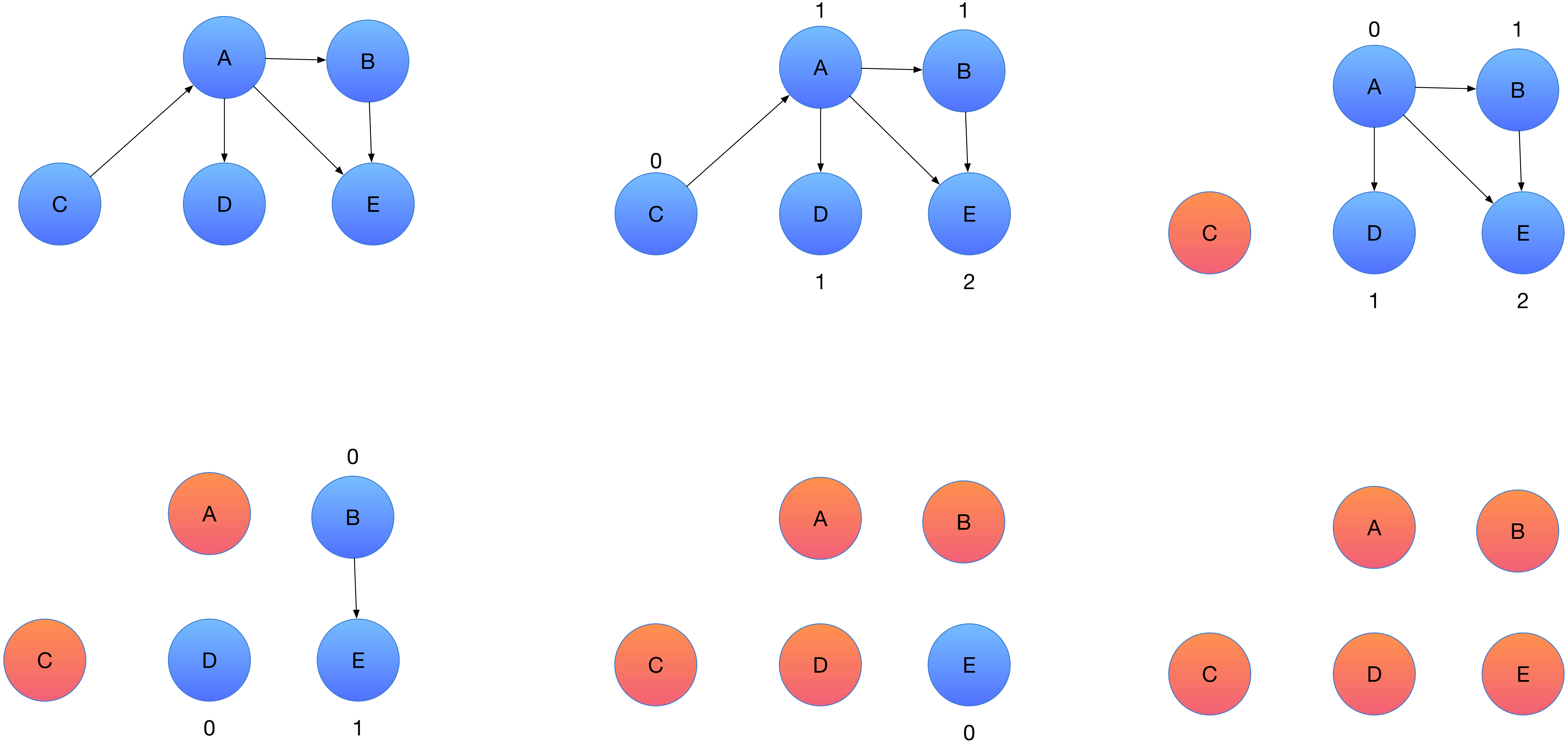
```
target [, target]: pre-requisites
    command1
    command2
    …
```

demo: echo

Prerequisites

- You can chain tasks based on their dependency.
- Given dependency, make decides the task order using topological sorting.
    - If A is a prerequisite for B, add A —> B.
    - Topological sorting based on in-degrees.

demo: hello + touch



`make -j [jobs]`

Timestamps

- Make determines whether a target file is up to date, or needs to be made again, based on the timestamps in the file system.
  - If the target is older than any of the pre-requisites, it needs to be made again.

demo: hello + touch

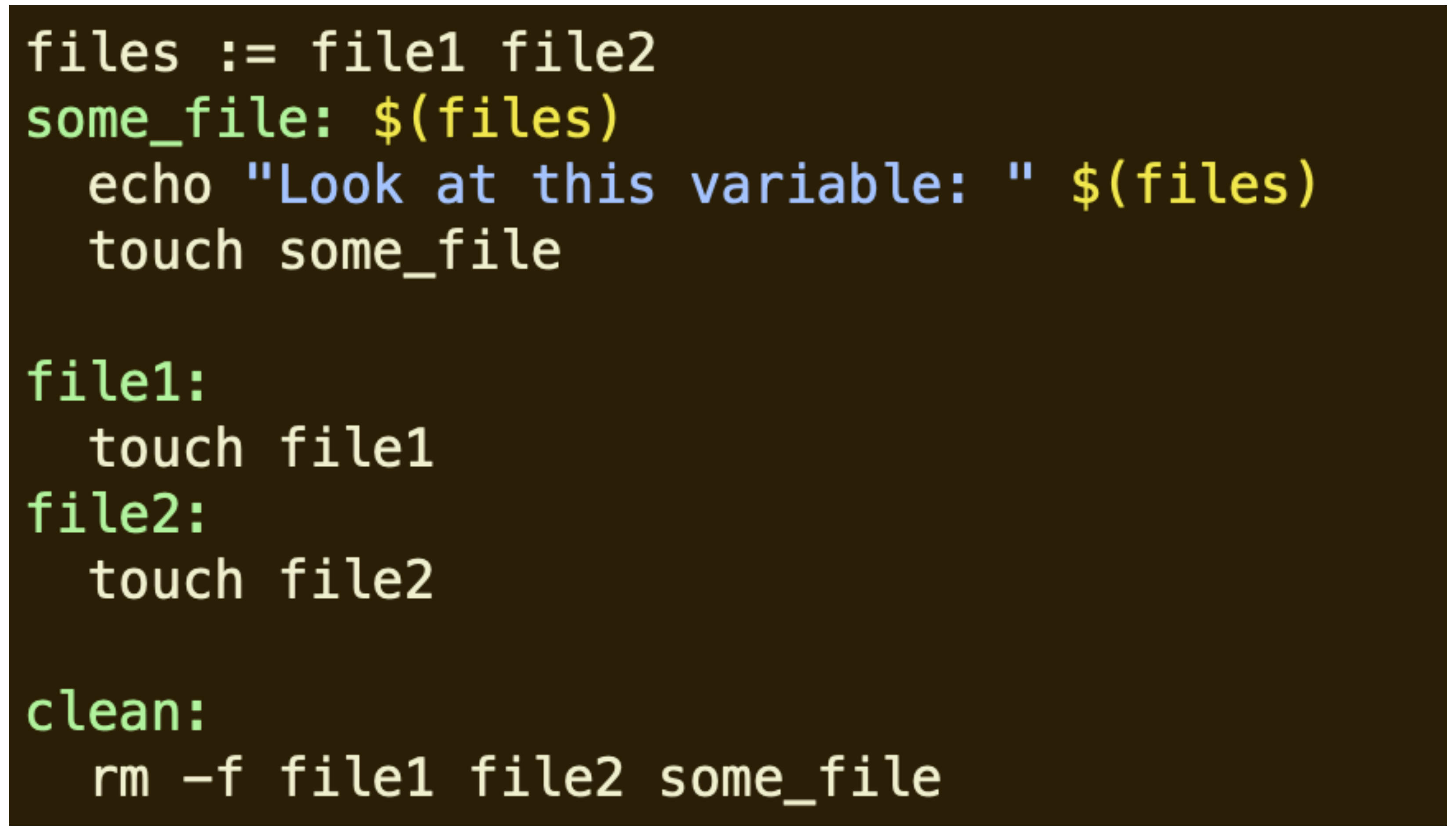
Typical make targets

- clean: remove all files that have been generated by running this Makefile
- install: builds, then copies the executables to appropriate locations so that the executables can be used (e.g., /usr/local/bin)
- all: achieve all other tasks

Other details (there are more tips and tricks)

See https://makefiletutorial.com/ for a comprehensive tutorial

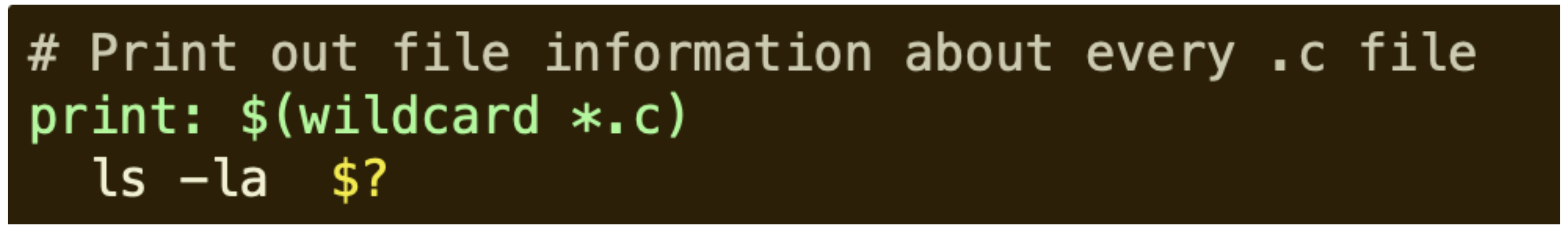
- variables
  - `files := file1 file2`, followed by `$(files)`
- wildcards
  - For example: `$(wildcard *.c)`
- Automatic variables
  - `$@:` target name
  - `$?:` all pre-requisites newer than target
  - `$^:` all pre-requisites



```
files := file1 file2
some_file: $(files)
  echo "Look at this variable: " $(files)
  touch some_file

file1:
  touch file1
file2:
  touch file2

clean:
  rm -f file1 file2 some_file
```



```
# Print out file information about every .c file
print: $(wildcard *.c)
  ls -la  $?
```

demo: hey

CMake

the meta make

- Manages the build system in a compiler-independent way: useful when you want to specify build process across multiple platforms
  - Chooses the appropriate build toolchain based on local platform and language standard
  - Can specify variable values in the build script and share those values in the source code via pre-processing

CMake

Some basic instructions

- `project(name version)`: sets the project name, and version numbers
- `add_executable(target_name dependencies)`: sets the dependency between the target and files it depends on
- `set(variable_name value)`: sets the value of a variable
- `configure_file (input output)`: copies input file to output file while preprocessing variable values

CMake Hands-on

(taken from https://cmake.org/cmake/help/latest/guide/tutorial/)

The tutorial example can be downloaded from:
https://cmake.org/cmake/help/latest/_downloads/ab37d97e635ba7864c2f68e9eb370b73/cmake-3.26.2-tutorial-source.zip

- Todo 1: set the minimum required version of CMake in `CMakeLists.txt`
- Todo 2: set the project name to Tutorial in `CMakeLists.txt`
- Todo 3: add an executable called Tutorial to the project - this is built using the `tutorial.cxx` file.
- Can you build it now?

CMake Hands-on

(taken from https://cmake.org/cmake/help/latest/guide/tutorial/)

```
$ mkdir Step1_build
$ cd Step1_build
$ cmake ../Step1 # create the actual build scripts
$ ls
$ cmake --build . # executes the created build scripts
```

CMake Hands-on

(taken from https://cmake.org/cmake/help/latest/guide/tutorial/)

```
# in tutorial.cxx file, change the following
const double inputValue = atof(argv[1]);

# into the following, which is in C++11 standard
const double inputValue = std::stod(argv[1]);
```

- Todo 4: we will introduce C++11 feature into tutorial.cxx as above.
- Todo 5: remove the line `#include <cstdlib>` from `tutorial.cxx` file.
- Can you build?
- Todo 6: add `CMAKE_CXX_STANDARD` and `CMAKE_CXX_STANDARD_REQUIRED` to `CMakeLists.txt` file.

CMake Hands-on

(taken from https://cmake.org/cmake/help/latest/guide/tutorial/)

- Todo 7: add version number 1.0 to the current project in `CMakeLists.txt` file.
- Todo 8: process `TutorialConfig.h.in` file with the version number and add to the build directory.
- Todo 9: add the build directory to the include path.
- Todo 10: define `Tutorial_VERSION_MAJOR` and `Tutorial_VERSION_MINOR` in `TutorialConfig.h.in` file.
- Todo 11: include `TutorialConfig.h` in `tutorial.cxx` file.
- Todo 12: print the version number!

ANT

**Another Neat Tool,
James Duncan Davidson, 1999**

- Java build system
- `<target>`: specifies units of build targets
- `<task>`: specifies units of build activity
- Can modularise using the `<ant>` task

```
<project name="example" default="link">

<property name="blddir" location="build" />
<property name="classes" location="${blddir}/classes" />
<property name="dist" location="${blddir}/dist" />

<target name="init">
  <mkdir dir="${blddir}" />
  <mkdir dir="${classes}" />
  <mkdir dir="${dist}" />
</target>

<target name="compile" depends="init">
  <javac destdir="${classes}"
    srcdir="maindir"
    includes="**/*.java"/>
 <ant antfile="sub/build.xml"
    target="compile"/>
</target>

<target name="link" depends="compile">
  <jar jarfile="${dist}/example.jar"
    basedir="${classes}"/>
</target>

<target name="clean">
  <delete dir="${blddir}" />
</target>
</project>
```

Maven

Apache Software Foundation, 2004

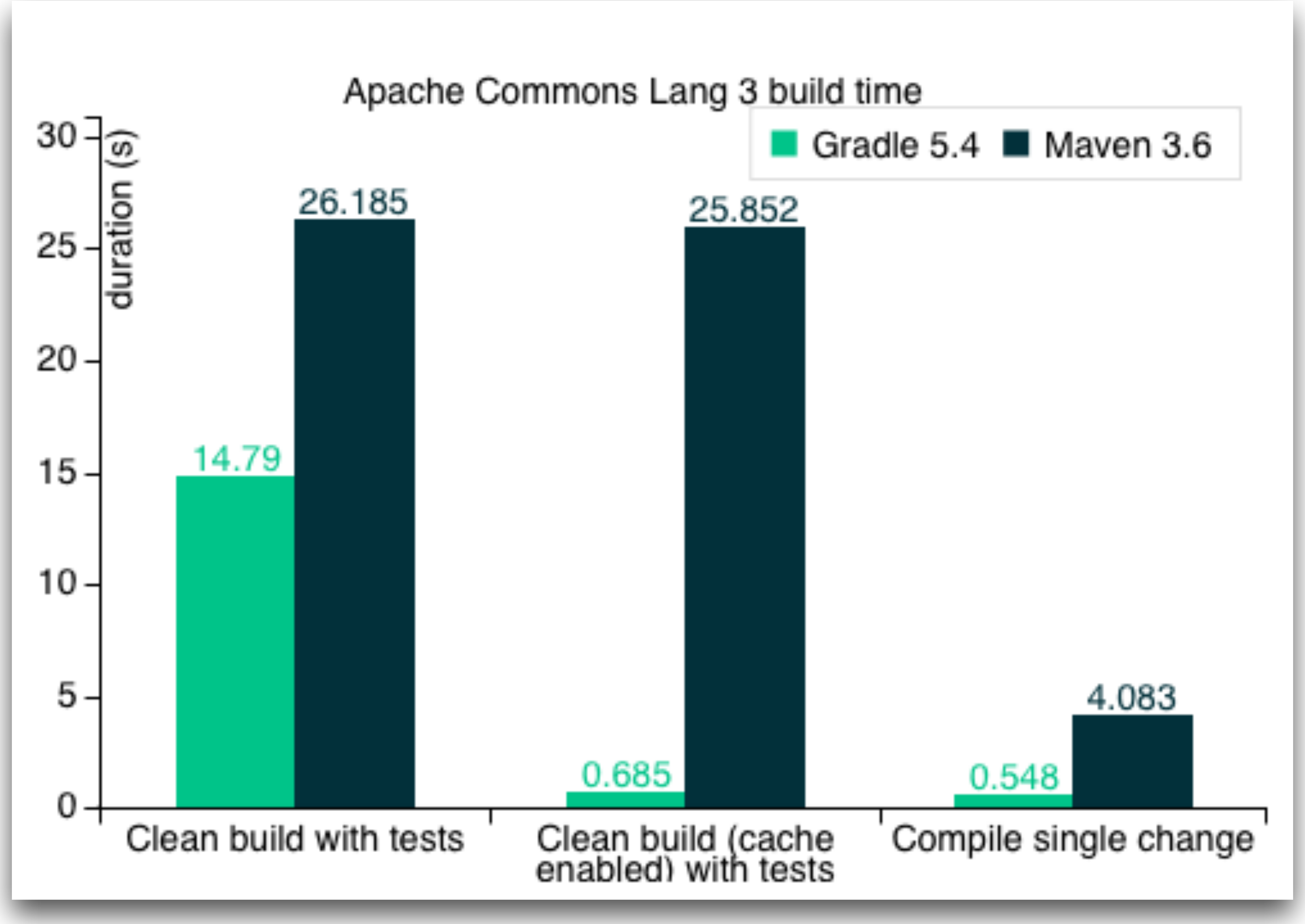
- Instead of individual build targets, organises builds into multiple stages of Build Lifecycle, such as `validate`, `compile`, `test`, `package`, `install`, …
- Provides 3rd party library repository, so that you do not have to commit specific versions of libraries your project depends on into VCS
    - https://mvnrepository.com/

    -
      ```
      <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>3.8.1</version>
        <scope>test</scope>
      </dependency>
      ```

Gradle

From 2008, Apache License 2.0

- Official build system of Android SDK
- Uses Groovy/Kotlin based Domain Specific Language (DSL) to describe builds
- More flexible compared to stages of Maven
- Better performance (build caching across networks, incremental builds, parallel compilation…)



https://gradle.org/maven-vs-gradle/

Bazel

Google, 2015

- An open source version of Google’s internal build tool, Blaze
- Uses contents-based hash to detect up-to-dateness (filesystem timestamps are problematic when…?)
- Uses Starlark DSL, which is a subset of Python
- Designed to handle multi-lingual projects from the scratch

Gradle Build Scripts

Build scripts are code (you have full power of Kotlin/Groovy)

```
# build.gradle
tasks.register('count') {
    doLast {
        4.times { print "$it " }
    }
}

$ gradle -q count
0 1 2 3
```

```
# build.gradle
4.times { counter ->
    tasks.register("task$counter") {
        doLast {
            println "I'm task number $counter"
        }
    }
}
tasks.named('task0') { dependsOn('task2', 'task3') }

$ gradle -q task0
I'm task number 2
I'm task number 3
I'm task number 0
```

Gradle Hands-on

(taken from https://spring.io/guides/gs/gradle/)

- Create a temporary project directory.
- Inside, create the following structure:

```
└── src
    └── main
        └── java
            └── hello
```

- `(mkdir -p src/main/java/hello)`

Gradle Hands-on

(taken from https://spring.io/guides/gs/gradle/)

```
src/main/java/hello/HelloWorld.java

package hello;

public class HelloWorld {
  public static void main(String[] args) {
  Greeter greeter = new Greeter();
  System.out.println(greeter.sayHello());
  }
}
```

```
src/main/java/hello/Greeter.java

package hello;

public class Greeter {
  public String sayHello() {
  return "Hello world!";
  }
}
```

Gradle Hands-on

(taken from https://spring.io/guides/gs/gradle/)

```
$ touch build.gradle
$ gradle tasks

$ vi build.gradle #add the following line

apply plugin: ‘java’

$ gradle tasks
$ gradle build
…
$ ls
```

Gradle Hands-on

(taken from https://spring.io/guides/gs/gradle/)

```
src/main/java/hello/HelloWorld.java

package hello;

import org.joda.time.LocalTime;

public class HelloWorld {
  public static void main(String[] args) {
    LocalTime currentTime = new LocalTime();
    System.out.println("The current local time is: " + currentTime);

    Greeter greeter = new Greeter();
    System.out.println(greeter.sayHello());
  }
}
```

Gradle Hands-on

(taken from https://spring.io/guides/gs/gradle/)

```
$ gradle build

$ vi build.gradle #add the following lines

repositories {
  mavenCentral()
}

sourceCompatibility = 1.8
targetCompatibility = 1.8

dependencies {
  implementation “joda-time:joda-time:2.2”
  testImplementation “junit:junit:4.12”
}
```

Gradle Hands-on

(taken from https://spring.io/guides/gs/gradle/)

```
$ vi build.gradle #add the following lines

jar {
  archiveBaseName = ‘gs-gradle’
  archiveVersion = ‘0.1.0’
}

apply plugin: ‘application’
mainClassName = ‘hello.HelloWorld’

$ gradle run
```

Hermetic Builds

- **hermetic**: adj. (of a seal or closure) complete and tight.
- A build is hermetic if it can be completed, from the project repository, in a self-contained manner.
    - The build tool itself should be part of the source code.
    - External dependencies should be taken care of.

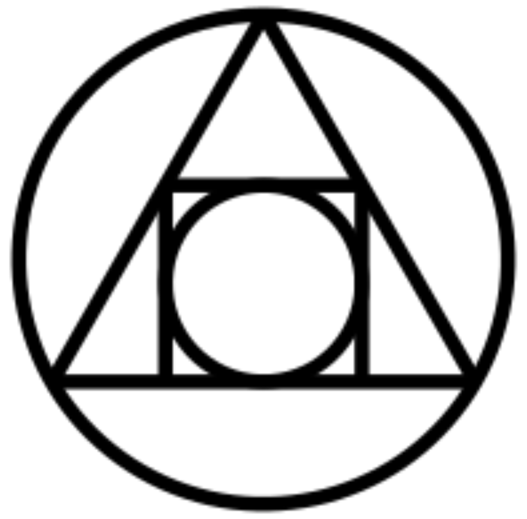
Imhotep
27th Century BCE
High priest to the sun god, Ra

deified after death

Thoth
Egyptian god of wisdom, writing, science, etc

Hermes Trismegistus
Syncretic combination of Thoth and Hermes

believed to have invented sealed glass to create…



Philosopher's Stone

made of a Hollywood movie

"The Mummy"
Original in 1932
Portrayed by Boris Karloff

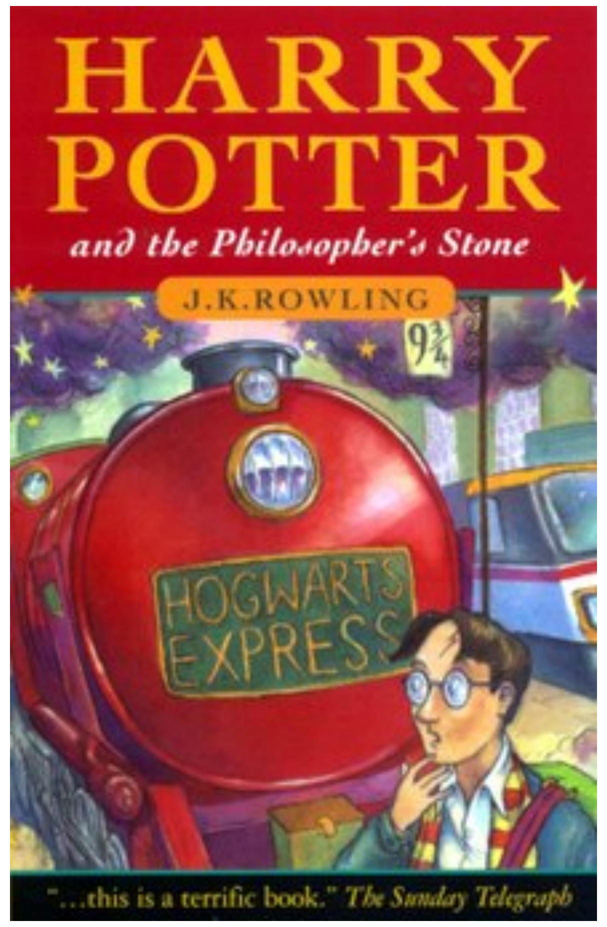
also played

"Bride of Frankenstein"
1935

remake

"The Mummy"
Universal Pictures
Remake 2001

Hermes
Greek god of travellers and thieves



HARRY POTTER
and the Philosopher's Stone
J.K.ROWLING
9¾
HOGWARTS EXPRESS
"...this is a terrific book." The Sunday Telegraph

Harry Potter
and the Philosopher's Stone
J. K. Rowling, 1997

Gradle Wrapper

- `$ gradle wrapper --gradle-version 8.0.2`
- `$ gradlew build`

Summary

- Build should be automated.
- Build should be controlled programatically.
- Build should be hermetic.