Build process & build management & docker

Jana Kostičová

Build process

Building SW = processing source code and its dependencies into a form that can be executed or used by a computer system.

- Writing / generating sources
- Static checking
- ….

} **Broader build process**

- Preprocessing (C / C++, some Fortran dialects, ..)
- Compilation (compiled languages)
- Linking (C / C++, assembly language, ..)
- Dependency resolution
- Packaging / bundling

} **Core build process**

- Automated tests
- Deployment
- ….

} **Broader build process**

It depends on various factors what activities are involved (programming language, size and type of application, target environment, ...)

Build tools

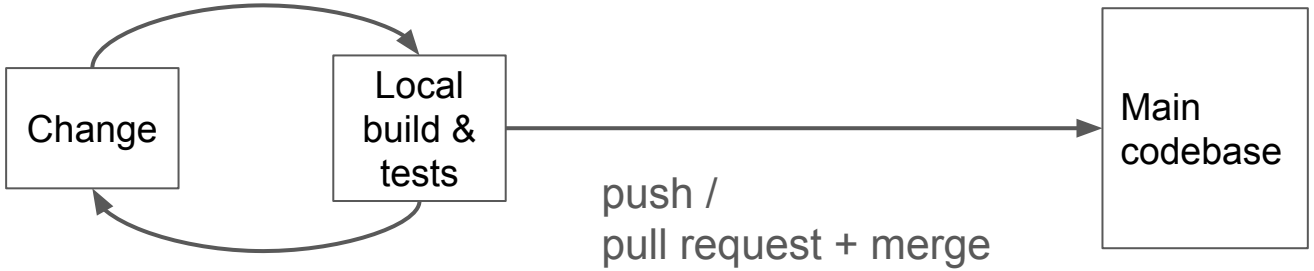
- Manual building - may suffice for extremely small projects with minimal dependencies
- Build tools - highly recommended if we have anything more complex

GNU make, Maven, Gradle, Vite, …

Key functionalities

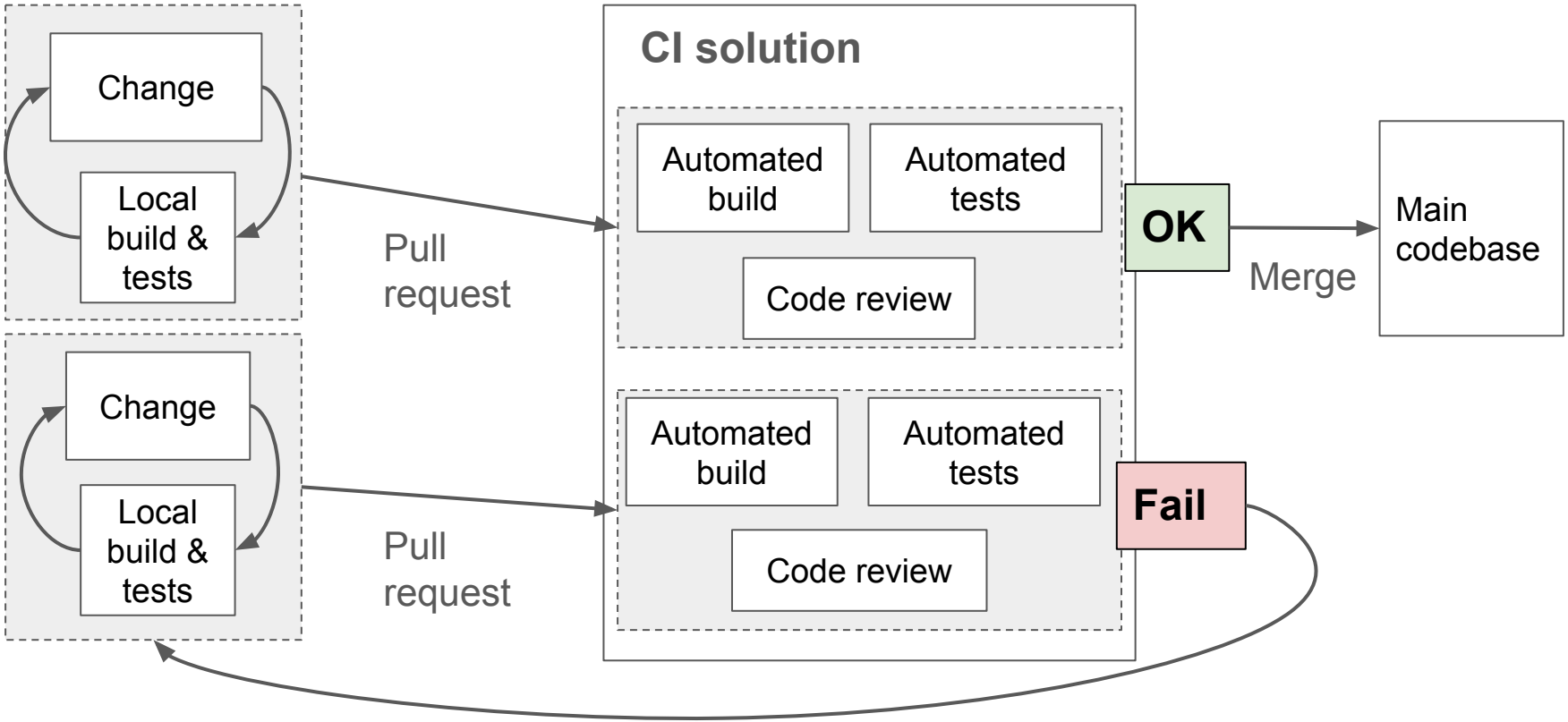
- Build automation
  - Scripting tasks for efficient and repeatable builds
- Build configuration
  - Defining different build configurations (e.g., development, testing, production).
- Dependency management
  - Automatically managing external libraries needed by the software
- Running automated tests
- Facilitating Continuous Integration (CI)

Simple projects



- Build tools might not be needed
- Code quality is only checked locally

Continuous Integration (CI) workflow



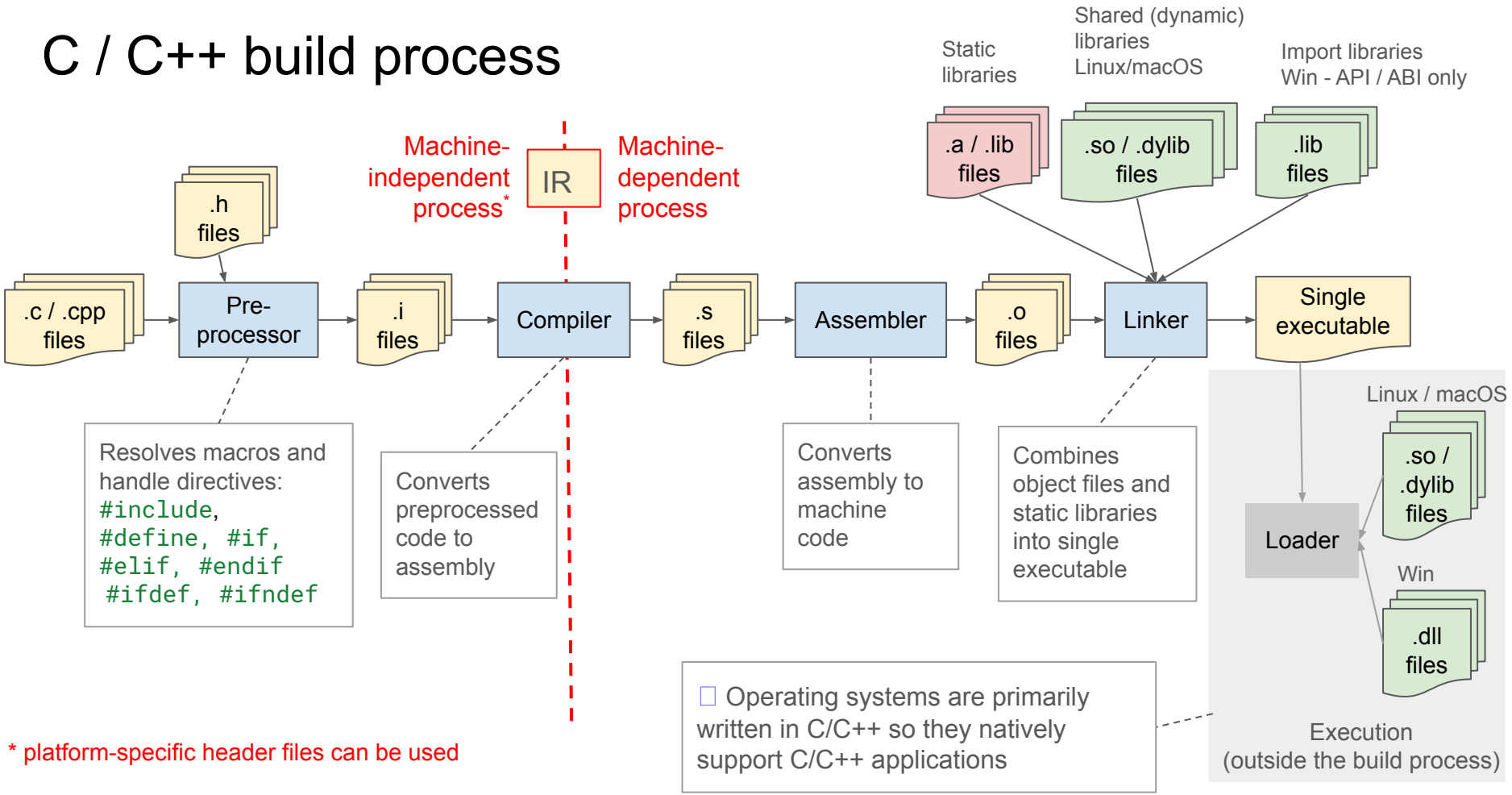
Local builds vs CI builds

- Local builds
  - Provide a fast feedback loop for developers during their coding process
  - Help to catch errors early on before committing code
- CI builds
  - Offer a “safety net” by ensuring code integrates and functions correctly before merging into the main codebase
  - Provides feedback to the developer on the success or failure of the build and test
  - Requires **automatization of build process**

- Together, they help maintain **code quality** and catch issues early in the development cycle
- Both of them should use the same build tool and similar configuration
  - Some differences may appear (e.g., omitting some tests in local environment)

C / C++

C / C++ build process



* platform-specific header files can be used

Static vs shared libraries

	Static linking (static libraries)	**Dynamic linking (shared libraries)**
Library contents	Copied into the executable	Referenced
Executable size	Larger	**Smaller**
Runtime dependency	**None**	Yes
Library updates	Requires rebuilding the application	No rebuilding (supposing API / ABI does not change)
Portability	**Portable**	Fragile
Startup performance	**Usually faster**	Usually slower due to runtime linking
Memory usage	Generally larger	Generally **smaller**, especially if many application use the library

Header files

- “Interface” of a library or another source file
  - Declarations - function prototypes, external variables, structures, typedefs, classes (C++)…
  - Definitions (rarely)
  - Macros
- Header file need to be included for each
  - set of functions used from a static library
  - set of functions used from a shared library
  - set of functions used from another source file

→ need by compiler: type checking, separate compilation

System libraries

- Provide system functions such as memory management, process creation, file handling,..
- C standard library
  - Shared version (.dll / .so) always globally available
  - Static version is mostly available or can be installed
  - Linker mostly finds it automatically, no need to provide the path/name manually
  - Header files: Part of C/C++ specifications, OSs should follow it (OS-specific extensions exist)
  - Implementation: Different, OS-specific
- Other
  - Win - Windows API: kernel32.dll, user32.dll, and gdi32.dll, …
  - Linux - libpthread, libm, libX11 …
  - macOS - libobjc, libpthread.libm, …
  - Available often only as shared libraries

DLL hell

- Windows 95/98/NT
- Applications crashing because of incompatible DLL versions, missing / conflicting DLLs ,...

C / C++ 3rd party and user-defined libraries

- 3rd party
  - Examples: *OpenSSL, SQLite, libcurl (network transfer), ..*
  - Mostly provided in both static and shared version
- When to use your own library
  - Common reasons (modularity, reusability, encapsulation, ...)
- When to use your own static library
  - You want to avoid surprises at runtime
  - You want single executable deployment
  - You don’t mind larger executable
- When to use your own shared library
  - More applications uses the same functionality and you want to take advantage of memory sharing
  - You want to update the library independently of the applications that use it
  - You want smaller executable size

Where OSs look for shared libraries at runtime

Linux (ld.so, ld-linux.so)

1. rpath
2. LD_LIBRARY_PATH
3. /lib, /usr/lib, /lib64, usr/lib64
4. /etc/ld.so.conf

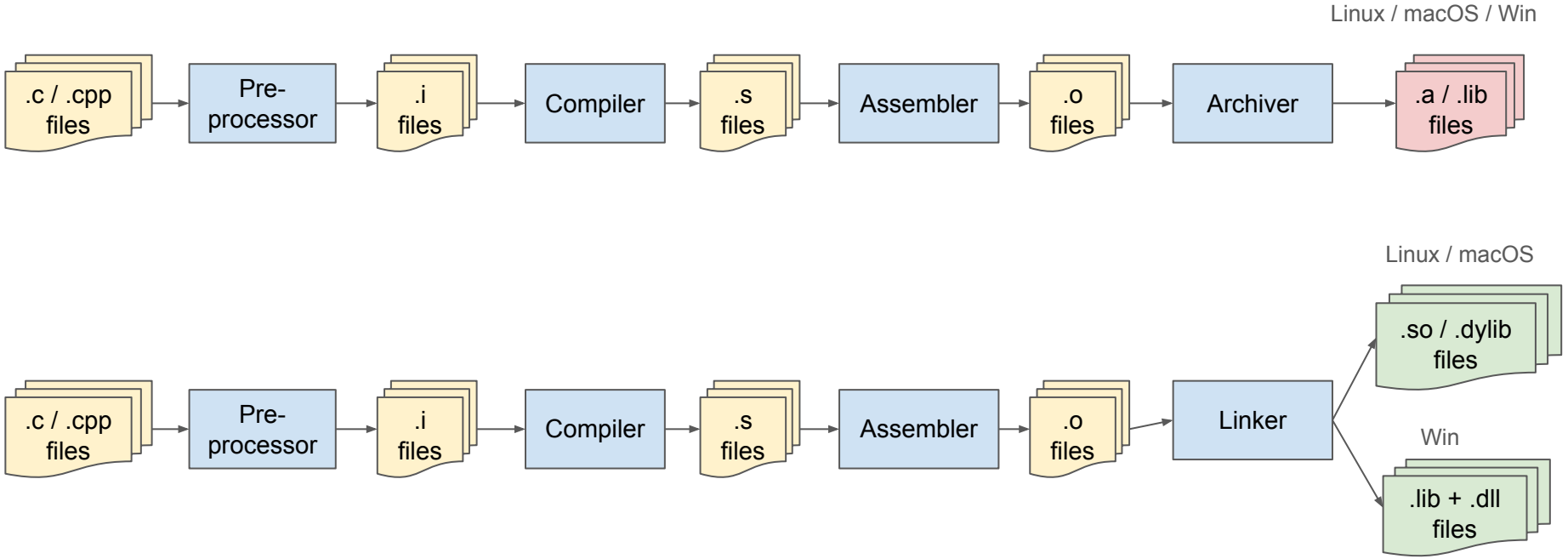
Windows (kernel32.dll / ntdll.dl)

1. Directory of the executable
2. Current working directory
3. C:\Windows\System32, C:\Windows\SysWow64, C:\Windows
4. PATH
5. Registry

MacOS (dyld)

1. rpath
2. DYLD_LIBRARY_PATH
3. /usr/lib, /System/Library/Frameworks, Library/Frameworks
4. dyld cache

Building C/C++ libraries



GNU make

Incremental build - aims to minimize the amount of work needed to rebuild a project after changes have been made

- Emerged in the late 1970s and still used today
- Popular for C/C++, can be used also for other languages
- Configuration file: Makefile

Main features

- Tracks dependencies between target files and their prerequisites efficiently
- Rebuilds only what's necessary based on changes
- Extensible through scripting and external tools

Limitations

- Relies on shell scripting
- Works well with small / medium sized projects, large projects can become hard to maintain
- Incremental build does not work well for file collections
- No support for managing external dependencies

} limits *make* usage for Java projects

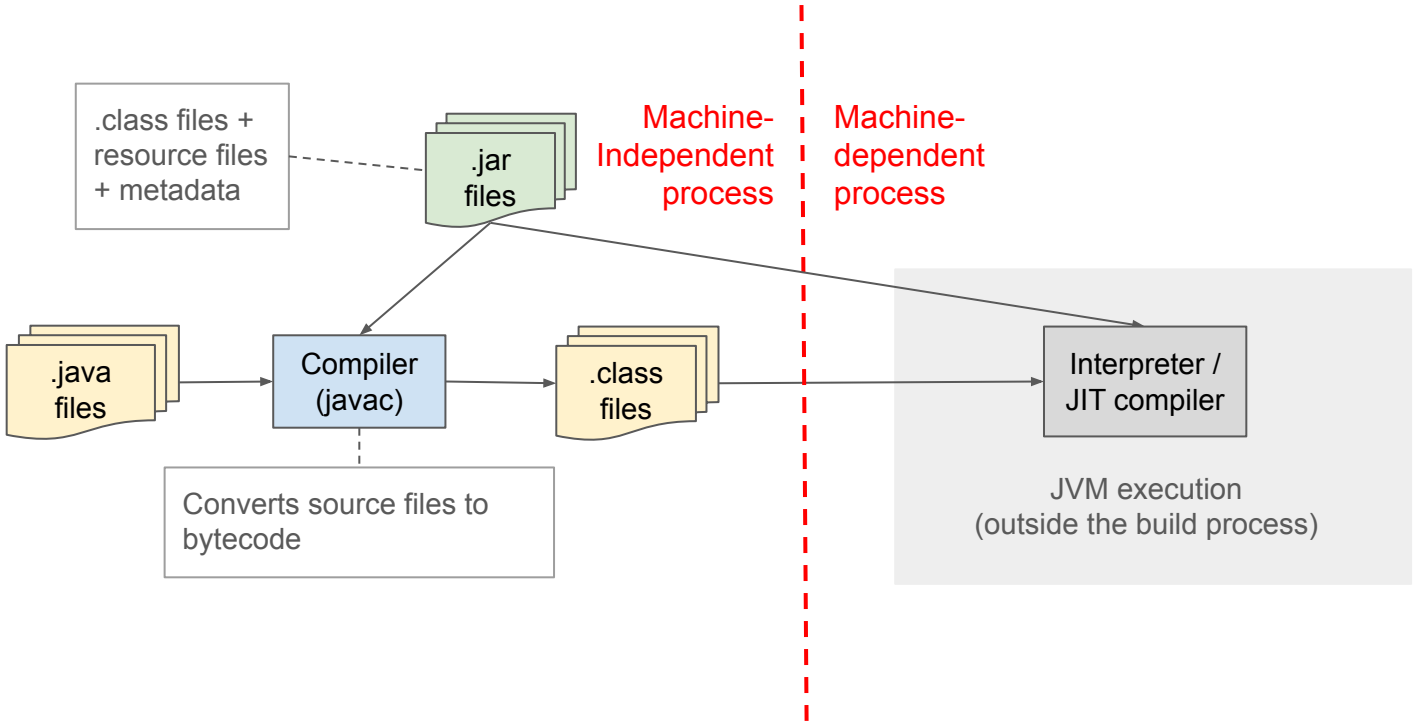
Manual: https://www.gnu.org/software/make/manual/

CMake

- First release in 2000
- Cross-platform
- Uses compiler-independent instructions for building applications
- Provide generators that translates these instructions to native build tools
  - Makefile, Visual Studio project files, …
  - Generated files are not intended to be edited manually

Java

Java - build process

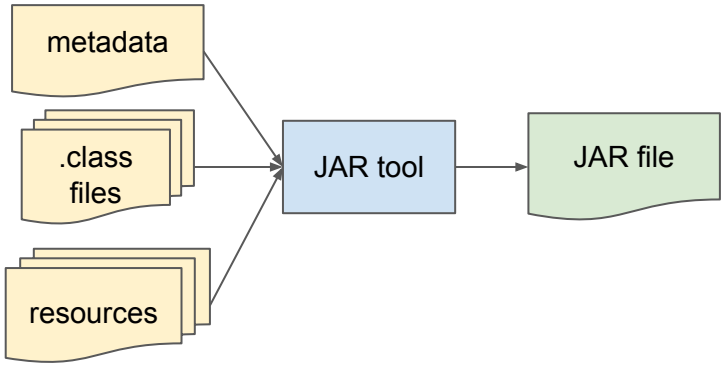


Java libraries

JAR files

= standard way to represent Java libraries

- They typically consist of classes, resource files and metadata (manifest file)
- JARs are runtime dependencies, dynamically loaded by JVM's class loader
- Each Java application typically loads classes from JARs into its own memory space (some mechanisms for class sharing exist - e.g., Class Data Sharing, Application Servers)



Java build tools

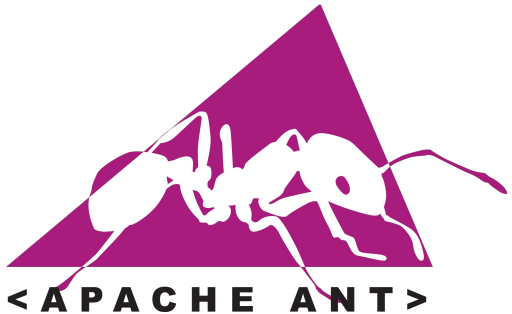
Make ❓ - issues arise:

- Java language-level dependencies during compilation
  - javac handles these dependencies within a given set of source files automatically
  - Makefile is primarily designed to have a single target file per rule - it doesn't work well at collection level (everything is recompiled even in case of a single change)
- Java project structure
  - Java projects typically organize source code by package structure that often result in deeply nested directories such as src/main/java/com/example/myapp/model/
  - Makefile can become cumbersome and difficult to maintain.
- External dependencies, manual JAR creation, classpath handling..

Ant, Maven, Gradle ✅

- Better choice

Apache Ant



- Released in 2000
- Configuration file: build.xml

Main features

- Uses XML
- Specifies build steps and tracks compilation dependencies similarly to “make” tool (imperative approach)
- **<javac>, <jar> and <java> tasks** simplify build and run
- Incremental build for collection of source files
- Limited management of external dependencies
- Cross-platform, extensible, adapts to existing project layout

Limitations

- Limited external dependency management - Ant has no means of downloading external dependencies from a central repository or resolve version conflicts
- No direct support for running automated tests

Apache Maven

- Released in 2004
- Emerged from within the Java ecosystem to address its specific needs
- Configuration file: pom.xml

Main features

- It uses XML
- Predefined build phases - build process is defined using plugins and goals, build phases are performed based on this configuration (declarative approach)
- Less verbose than Ant
- Robust dependency management - libraries are downloaded from maven repository, versions are tracked explicitly allowing version conflict detection

Limitations

- Steeper learning curve
- It works best with standard Maven project structure
- Customization might be complex - it is required to be familiar with Maven plugin development

Apache Maven - default build phases

- validate
- compile
- test
- package (e.g. to JAR / WAR / EAR file)
- integrate-test
- verify (additional checks on the packaged artifact)
- install (installing packaged artifact into local Maven repository)
- deploy (deploying packaged artifact into remote repository)

Gradle

- Released in 2009
- Configuration file: build.gradle

Main features

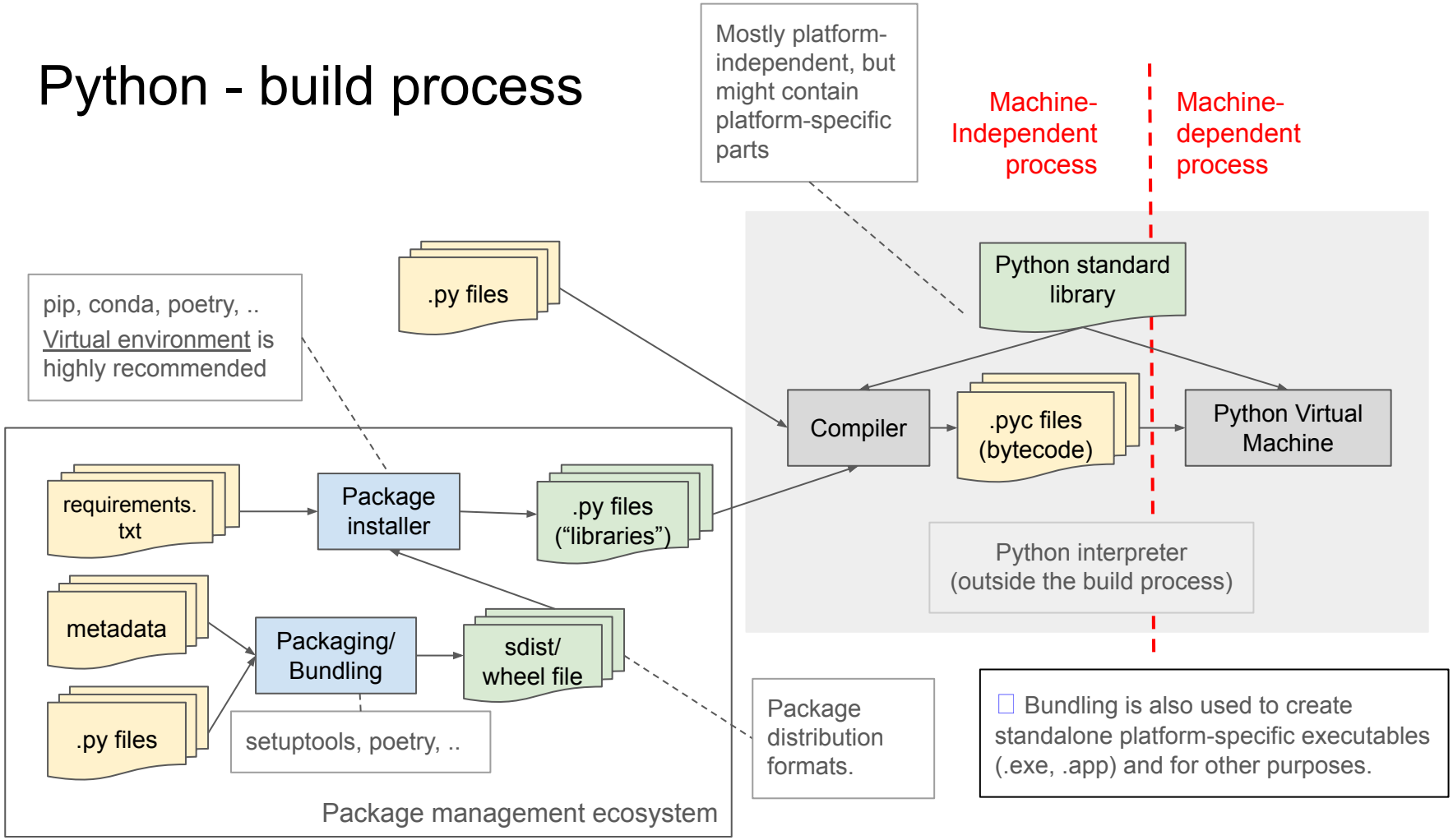
- Concise syntax - uses Groovy and Kotlin-based DSL
- Flexible approach to determine the execution order of the tasks (DAG)
- Robust dependency management

Limitations

- High flexibility may lead to complex configuration files with hard-to-understand semantics

Python

Python - build process



	Pre-process-ing	Compilation	Static linking	Build tools / package managers (PM) - examples	(External) dependency management	Packaging / bundling
C / C++	Yes	Yes	Yes	make, CMake PMs: vcpkg, Conan System PMs: apt, yum	Generally not needed (static linking & pre-installed standard libraries). Some package managers are available.	Generally not needed (static linking results in single executable).
Java	No	Yes (to bytecode)	No	Ant, Maven, Gradle	Recommended (Maven, Gradle)	Recommended (jar / war / ear)
C#	No	Yes (to CIL / bytecode)	No	MSBuild PM: NuGet	Optional (pre-installed standard libraries vs. NuGet)	Recommended (MSBuild)
Python	No	No	No	pip + venv + setuptools Poetry, Hatch	Recommended (for example PyPI repository & pip)	Recommended
JavaScript	No	Optional / transpiling to older JS	No	Webpack, Parcel, Vite, esbuild PMs: npm, yarn	Recommended (npm, yarn)	Recommended

References

- Wikipedia: Software build
- ISO/IEC 9899:2024 (en) — N3220 working draft (PDF), open-std.org. 2024-02-22 (almost identical to official C23 standard)
- GNU make: https://www.gnu.org/software/make/
- CMake: https://cmake.org/
- Apache Ant: https://ant.apache.org/
- Apache Maven: https://maven.apache.org/
- Gradle: https://gradle.org/

Docker

Docker

- First released in 2013
- OS-level virtualization
- Docker container packages an application and its dependencies that can run as an isolated process on the host system
  - Self-contained
  - Isolated
  - Independent
  - Portable
- Docker container vs virtual machine

More info:

- https://en.wikipedia.org/wiki/OS-level_virtualization
- https://en.wikipedia.org/wiki/Docker_(software)
- https://www.youtube.com/watch?v=8fi7uSYlOdc (insights into how the containers work)

Docker concepts

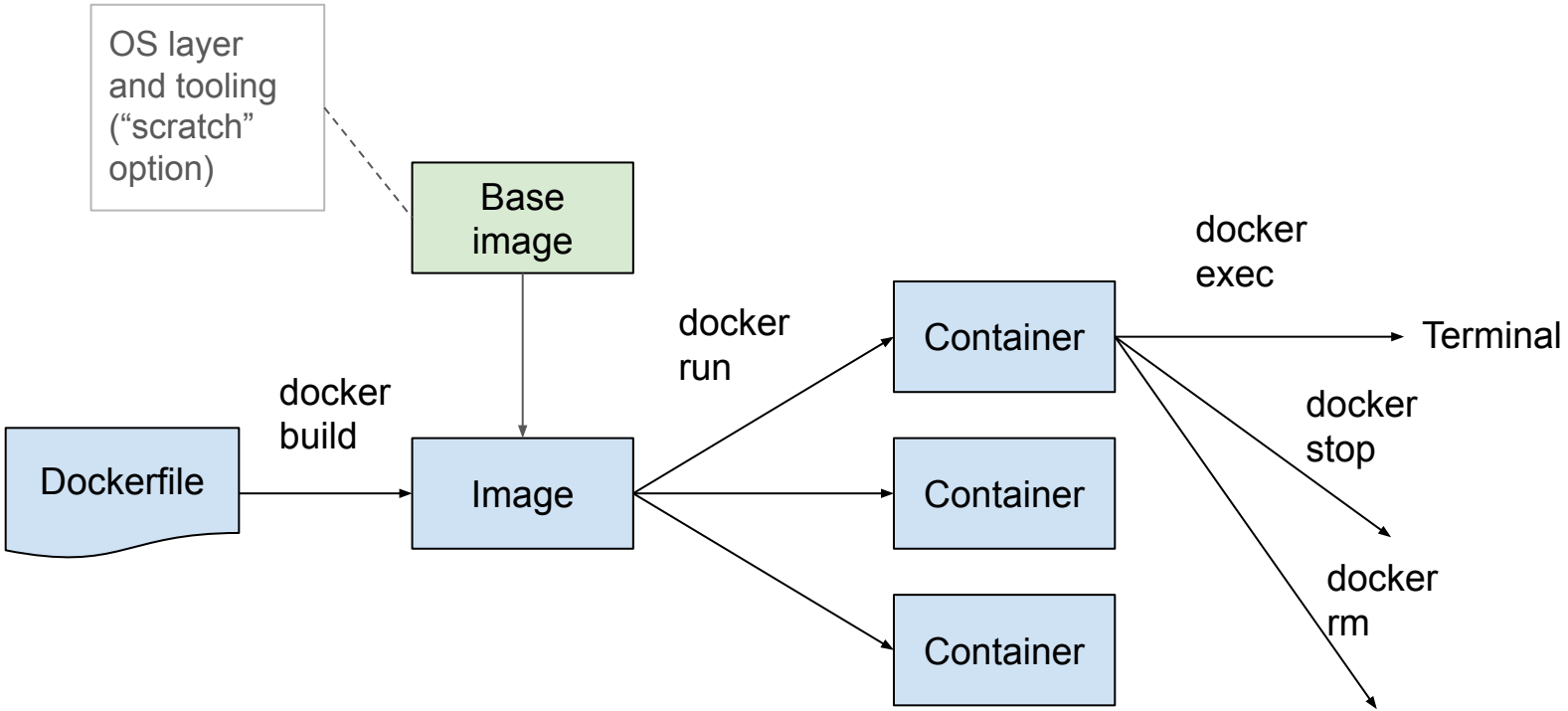
We mostly use 101 course from https://www.docker.com/101-tutorial

SW:
- Docker daemon
- Docker client program

Objects:
- Docker container
- Docker image
- Docker service (we will not speak about this today)

Registries:
- Docker registry (default one Docker Hub)

Tools:
- Docker Dashboard
- Docker Compose
- Docker Swarm (we will not speak about this today)


OS layer and tooling (“scratch” option)
Base image
Dockerfile
docker build
Image
docker run
Container
Container
Container
docker exec
Terminal
docker stop
docker rm

References

- https://en.wikipedia.org/wiki/OS-level_virtualization
- https://en.wikipedia.org/wiki/Docker_(software)
- https://docs.docker.com/engine/reference/commandline/docker/ - CLI reference
- https://www.youtube.com/watch?v=8fi7uSYlOdc (insights into how the containers work)
- R. Lukoťka: Docker