

Introduction to Gradle

The fundamentals of building projects with Gradle

Install

You **must** have a JDK and the latest version of Gradle installed.

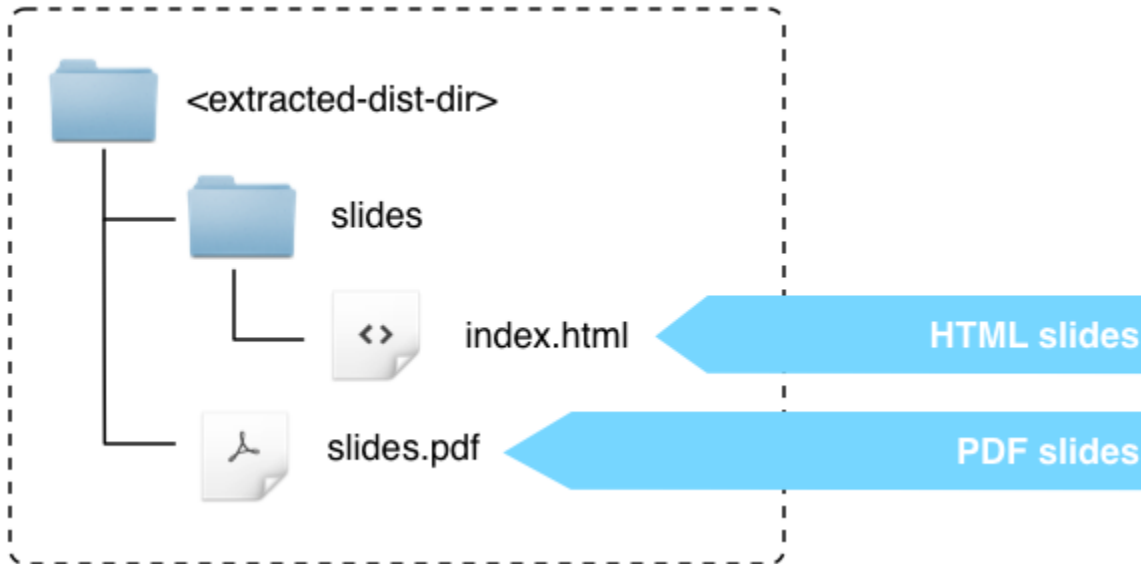
See the `setup-instructions.pdf` in your class materials for download links to install a JDK and Gradle.

HTTP Proxy

If you're behind a proxy, follow the setup instructions to configure Gradle to use your proxy.

Slides

- Available in different formats
- Same content as today's presentation



Practical labs

- Each lab has a README with instructions
- Take your time and experiment
- The labs are not a test!
- Hints can be found in the slides

Objectives

- A solid understanding of basic Gradle concepts
- Knowledge of how to use the Java plugins (`application` and `java-library`)
- Basic knowledge of how to work with dependencies
- Understanding of features to organize and customize Gradle builds
- Exposure to more advanced build capabilities

Ask questions

- Please ask questions at any time!

Topics we won't cover

- Android or other JVM language builds
- Continuous Integration/Delivery
- Advanced plugin development techniques
- Advanced dependency management techniques

Agenda

- About Gradle
- Gradle overview
 - Build scripts
 - Using plugins and tasks
 - Running builds
- Building Java projects
- Dependency management basics
- Structuring a build
- Writing custom build logic
- More resources

Gradle

About the project

Gradle

Gradle is a build and automation tool.

Gradle can automate the building, testing, publishing and deployment of your software.



gradle.org

Gradle Project

- Open Source, Apache v2 license - Completely free to use
- Source code on Github - github.com/gradle/gradle
- Active user community, centered around discuss.gradle.org
- Frequent releases (minor releases roughly every 6-8 weeks)
- Strong quality commitment
 - Extensive automated testing (including documentation)
 - Backwards compatibility & feature lifecycle policy
- Developed by domain experts
 - Gradle, Inc. staff and community contributors

Gradle Documentation

- User Manual
 - Many chapters and [self-contained downloadable samples](#)
 - [HTML](#)
 - [PDF](#)
- Build Language Reference (gradle.org/docs/current/dsl/)
 - Best starting point when authoring build scripts
 - Javadoc-like, but higher-level
 - Links into [Javadoc](#)

Other Gradle Resources

- Install the latest release from gradle.org/install
- Older Gradle releases gradle.org/releases
- help.gradle.org
 - Portal to other resources

Getting Information

Print command line options:

```
$ gradle -h
```

Print the available tasks in a project:

```
$ gradle :tasks
```

Print basic help information:

```
$ gradle :help
```

Getting help on a specific task

```
$ gradle :help --task taskname
```

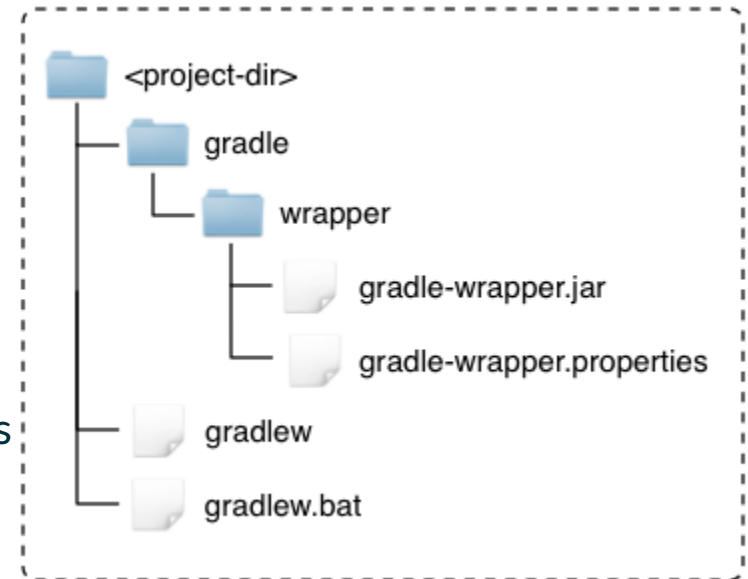
Best Practices for Running Gradle

- Always use the wrapper
- Keep up-to-date with new releases
 - Performance bottlenecks are removed
 - New features are added
 - Deprecation warnings prevent surprises

Gradle Wrapper

A way to make sure everyone uses the same version of Gradle to build a project.

- **gradle-wrapper.jar**: Micro-library for downloading distribution
- **gradle-wrapper.properties**: Defines distribution download URL and options
- **gradlew**: Gradle executable for *nix systems
- **gradlew.bat**: Gradle executable for Windows systems
- Downloaded distributions go into your `GRADLE_USER_HOME/wrapper/dists` directory



Running a build with the wrapper:

```
$ ./gradlew build
```

The very first time you run a build with the wrapper, Gradle will download a copy of the distribution.

Wrapper task

- Bootstrap using the wrapper with your build with the `wrapper` task
- Wrapper task is built-in and generates:
 - wrapper scripts
 - wrapper jar
 - wrapper properties

```
$ gradle wrapper --gradle-version=6.8.3
```

- The `--gradle-version` flag lets you specify a particular version of Gradle to use.
- The `--distribution-type` flag lets you specify `all` if you want the complete distribution (the default is `bin`). The result is larger, but includes the source and documentation.

Gradle Build Scans

Creating build scans

- [Creating a build scan is free.](#)
- Build scans are a centralized and shareable record of a build.
- Build scans offer insight into how you are building your software.
- **All build scans created during this course will be uploaded to the public build scan service.**
- A [self-hosted version](#) is also available with more features.
- See the [latest build scans for the Gradle project itself](#).

We encourage you to generate a build scan if you have a problem with a lab, so we can help you solve your problem. Just run your build with `--scan`.

Creating build scans

- Gradle will ask to accept 'Gradle Terms of Service' before uploading scan data
- To allow the upload permanently for a build, add the following to `settings.gradle(.kts)`

```
plugins {  
    id("com.gradle.enterprise") version "3.6"  
}  
gradleEnterprise {  
    buildScan {  
        termsOfServiceUrl = "https://gradle.com/terms-of-service"  
        termsOfServiceAgree = "yes"  
    }  
}
```

Gradle Basics

Gradle DSLs

Gradle is implemented in Java, with Kotlin and Groovy DSL layers.

Kotlin and Groovy bring:

- Domain Specific Language (DSL) capabilities
- Better readability and comprehensibility
- Many useful utilities built-in
- IDE support for build scripts (Kotlin in IntelliJ)

Gradle Build Scripts

- Files ending in `.gradle.kts` are compiled as Kotlin code
- Files ending in `.gradle` are compiled as Groovy code
- Build script files (`build.gradle[.kts]`) delegate to `org.gradle.api.Project`
- Settings script files (`settings.gradle[.kts]`) delegate to `org.gradle.api.initialization.Settings`
- Groovy DSL and Kotlin DSL syntax is similar but may differ in details
- The slides use syntax that works in both Groovy DSL and Kotlin DSL

Typical script files

Settings script

```
rootProject.name = "name-of-build"

include("subproject")
include("another-subproject")
```

Build script

```
plugins {
    id("java-library")
}

repositories {
    mavenCentral()
}

dependencies {
    implementation("com.google.guava:guava:30.0-jre")
}
```

Configuration & Execution

Build Lifecycle

- Initialization Phase
 - Configure environment (init.gradle, gradle.properties)
 - Find projects and build scripts (settings.gradle)
- Configuration Phase
 - Evaluate all build scripts
 - Build object model (Gradle -> Project -> Task, etc.)
 - Build task execution graph
- Execution Phase
 - Execute (subset of) tasks

A key concept to grasp.

Plugins and Tasks

Tasks are the basic unit of work in Gradle.

- Plugins provide ready to use tasks
- For example the `application` plugin adds tasks like `:compileJava`, `:test` or `:run`
- For individual needs, custom tasks can be created by the user (we will look at that later)

Discovering tasks

The built-in task `:tasks` lists available tasks in a project, either defined in the build script or provided by applied plugins.

```
$ gradle :tasks
```

Tasks in the output are organized by assigned group property e.g. `Build Setup` show up under the header `Build Setup tasks`.

The description property describes the purpose of a task.

Built-in tasks

Every Gradle project provides several tasks out-of-the-box.

Built-in tasks provide useful and commonly used functionality without having to apply any plugins.

Examples:

- `:wrapper` - Generates the Wrapper files for this build.
- `:help` - Demonstrates how to run Gradle from the command line.
- `:dependencies` - Renders a tree of dependencies defined in the build.

Tasks provided by plugins

Applying a plugin, like `application` or `java-library`, in the `build.gradle(.kts)` script adds additional tasks.

```
plugins {  
    id("application")  
}
```

Examples:

- `:assemble` - Assembles the outputs of this project
- `:test` - Runs the unit tests
- `:run` - Runs this project as a JVM application
- `:build` - Assembles and tests this project

Task Groups

By default, the tasks report only shows tasks that have a group.

- Other tasks are hidden, because you usually do not need to call them directly.
- Tasks depend on each other and dependencies are automatically executed.
- E.g., if you execute `:assemble` it automatically executes `:compileJava`

Still, you can use `:tasks --all` to see these tasks.

Examples of hidden tasks:

- `:compileJava` - Compiles main Java source.
- `:compileTestJava` - Compiles test Java source.
- `:processResources` - Processes main resources.
- `:processTestResources` - Processes test resources.

Task Dependencies

- Tasks depend on each other
- Semantic relationship (A produces something that B consumes)
- E.g., The `:jar` task picks up the classes compiled by the `:compileJava` task
- Executed tasks form a directed acyclic graph
- E.g., if you execute `:assemble` on a Java application project many tasks are executed

```
$ gradle :assemble --console=plain
```

```
> Task :compileJava
> Task :processResources NO-SOURCE
> Task :classes
> Task :jar
> Task :startScripts
> Task :distTar
> Task :distZip
> Task :assemble
```

Fuzzy name matching

Save your fingers by only typing the bare minimum to identify a task.

```
$ gradle cJ  
  
> Task :compileJava
```

Gradle understands how to match against camel-case task names. Ambiguous matches fail the build.

Lab

01-lab-gradle-basics

Plugins

Gradle Plugins

Plugins are just packaged build logic.

Plugins can do anything that you can do in a build script, and vice versa.

Plugins aid:

1. **Reuse** - avoid copy/paste
2. **Encapsulation** - hide implementation detail behind a DSL
3. **Modularity** - clean, maintainable code
4. **Composition** - plugins can complement each other

Typical Plugin Functions

Some things plugins typically do:

- **Extend the Gradle model** with new elements
- Configure the project according to **conventions**
 - Add new tasks
 - Configure existing model elements
 - Add configuration rules for future elements
- Apply some very **specific configuration**
 - Configure the project for very specific standards

Applying plugins

Plugins are applied in a `plugins` block:

```
plugins {  
    id("name-of-plugin")  
}
```

Plugins can also have versions (if they are not built-in plugins)

```
plugins {  
    id("name-of-plugin") version "1.0"  
}
```

- Gradle discovers versioned plugins in the [Gradle Plugin Portal](#)
- You can configure [other repositories as sources for plugins](#).

Plugins can be defined [as part of the build itself to reuse custom logic or build configuration](#).

- We will get back to this topic later.

Building Java projects

Java Plugins

- `java` plugin
 - The basis of Java development with Gradle
 - "main" and "test" source set conventions
 - Incremental compilation
 - Dependency management
 - JUnit & TestNG testing
 - Javadoc generation
- `java-library` plugin
 - Automatically applies `java` plugin
 - Adds the ability to declare "api" dependencies
- `application` plugin
 - Automatically applies `java` plugin
 - Requires a Main class to run and package an application

JVM Language Plugins

- Other JVM plugins can be combined with `java-library` or `application`
 - `groovy`, `scala`, `org.jetbrains.kotlin.jvm`

```
// To write an application in Kotlin:
plugins {
    id("application")
    id("org.jetbrains.kotlin.jvm") version "1.4.31"
}
```

```
// To write a library in Groovy:
plugins {
    id("java-library")
    id("groovy")
}
```


api **VS** implementation

Java libraries can [separate their implementation and API dependencies](#).

Dependencies appearing in the `api` will be transitively exposed to consumers of the library when compiling. Dependencies found in the `implementation` will not be exposed to consumers when compiling but will be available at runtime.

This has many advantages over a single `compile` time dependency scope.

Source Sets

A logical compilation/processing unit of sources.

- Java source files
- Non compiled source files (e.g. properties files)
- Classpath separation (compile & runtime)
- Output class files
- Compilation tasks

```
// Example: change layout to a typical Eclipse project layout
sourceSets {
    main {
        java {
            srcDir("src") // default was "src/main/java"
        }
        resources {
            srcDir("resource") // default was "src/main/resources"
        }
    }
}
```

Lifecycle Tasks

The `java-library` plugin, and most other plugins, provides a set of “lifecycle” tasks for common tasks.

- `:clean` - delete all build output
- `:classes` - compile code, process resources
- `:test` - run tests
- `:assemble` - make all archives (e.g. zips, jars, wars etc.)
- `:check` - run all quality checks (e.g. tests + static code analysis)
- `:build` - combination of `:assemble` & `:check`

Testing

Built-in support for JUnit 4, JUnit 5 and TestNG.

- Pre-configured "test" task
- Automatic test detection
- Forked JVM execution
- Parallel execution
- Configurable console output
- Human-readable HTML reports
- Machine-readable reports for further processing (e.g. XML)

Parallel Test Execution

Gradle can run test classes in parallel

```
// Kotlin DSL
tasks.test {
    maxParallelForks = 8
}

// Groovy DSL
tasks.named("test") {
    maxParallelForks = 8
}
```

IDE integration

- [Eclipse Buildship](#)
- [IntelliJ](#)

IDEs can delegate to Gradle to run tests and other arbitrary tasks.

Configuring details (Kotlin DSL)

```
// General project configuration
version = "1.0"
group = "com.example"

// Java specific configuration
java {
    toolchain {
        languageVersion.set(JavaLanguageVersion.of(11))
    }
}

// Task specific configuration
tasks.test {
    useJUnitPlatform() // JUnit 5
}

tasks.jar {
    manifest {
        attributes("Implementation-Version" to project.version)
    }
}
```

Configuring details (Groovy DSL)

```
// General project configuration
version = "1.0"
group = "com.example"

// Java specific configuration
java {
    toolchain {
        languageVersion.set(JavaLanguageVersion.of(11))
    }
}

// Task specific configuration
tasks.named("test") {
    useJUnitPlatform() // JUnit 5
}
tasks.named("jar") {
    manifest {
        attributes("Implementation-Version": project.version)
    }
}
```


Lab

02-java-plugins

Dependency Management

Dependency Management

Gradle supports managed dependencies.

Basics

- Declare repositories to find binary dependencies
- Declare dependencies using coordinates (group, artifact name, version)

Advanced

- Declare dependency constraints to control versions and resolve version conflicts
- Select from multiple variants for a module

Publishing

- Publish your own libraries to repositories

Repositories

- Any Maven or Ivy repository can be used
- Very flexible layouts are possible for Ivy repositories

```
repositories {  
    mavenCentral()  
  
    maven {  
        url = "https://repo.example.com"  
    }  
  
    ivy {  
        url = "https://repo.example.com"  
        layout("gradle") // default  
    }  
}
```

Starting with Gradle 6.8, [repositories can be defined in settings.gradle\(.kts\).](#)

Configurations for Declaration

Dependencies are declared on *configurations*. [See `java-library` defined configurations.](#)

```
dependencies {  
    implementation("...") // internal dependencies of library  
  
    api("...") // dependencies library re-exports for compilation  
  
    runtimeOnly("...") // dependencies only for runtime of library  
    compileOnly("...") // internal dependencies of library (hidden at runtime)  
  
    compileOnlyApi("...") // re-exported dependencies (hidden at runtime)  
  
    testImplementation("...") // dependencies for testing library  
    testCompileOnly("...")    // dependencies only for compiling tests  
    testRuntimeOnly("...")    // dependencies only for running tests  
}
```

See [Declaring Dependencies](#) in User Manual for more details.

Dependencies with a version

- Declaring dependencies to modules

```
dependencies {  
    // compact declaration assumes a "required" version  
    implementation("com.fasterxml.jackson.core:jackson-core:2.11.3")  
  
    // declaration using rich version API, this is equivalent  
    implementation("com.fasterxml.jackson.core:jackson-core") {  
        version {  
            // Must be at least 2.11.3, but may be upgraded  
            require("2.11.3")  
        }  
    }  
}
```

Dependencies without a version

- You can declare dependencies without versions
- You can depend on a platform - aka Bill-of-Material (BOM) - to pull in versions; such as
 - [org.springframework.boot:spring-boot-dependencies](https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-dependencies)
 - [org.junit:junit-bom](https://mvnrepository.com/artifact/org.junit/junit-bom)
 - [com.fasterxml.jackson:jackson-bom](https://mvnrepository.com/artifact/com.fasterxml.jackson/jackson-bom)
- A local Gradle project that uses the [java-platform plugin](#)

```
dependencies {  
    // platform (BOM) provides versions  
    implementation(platform("com.fasterxml.jackson:jackson-bom:2.11.3"))  
  
    // compact declaration without a version  
    implementation("com.fasterxml.jackson.core:jackson-core")  
}
```

SNAPSHOT and Dynamic Deps

Version numbers ending in `-SNAPSHOT` are *changing dependencies*.

```
dependencies {  
    implementation("org.company:some-lib:1.0-SNAPSHOT")  
}
```

Dynamic dependencies do not refer to concrete versions.

```
dependencies {  
    implementation("org.company:some-lib:2.+") // latest 2.x version  
    implementation("org:somename:latest.release") // latest stable version  
}
```

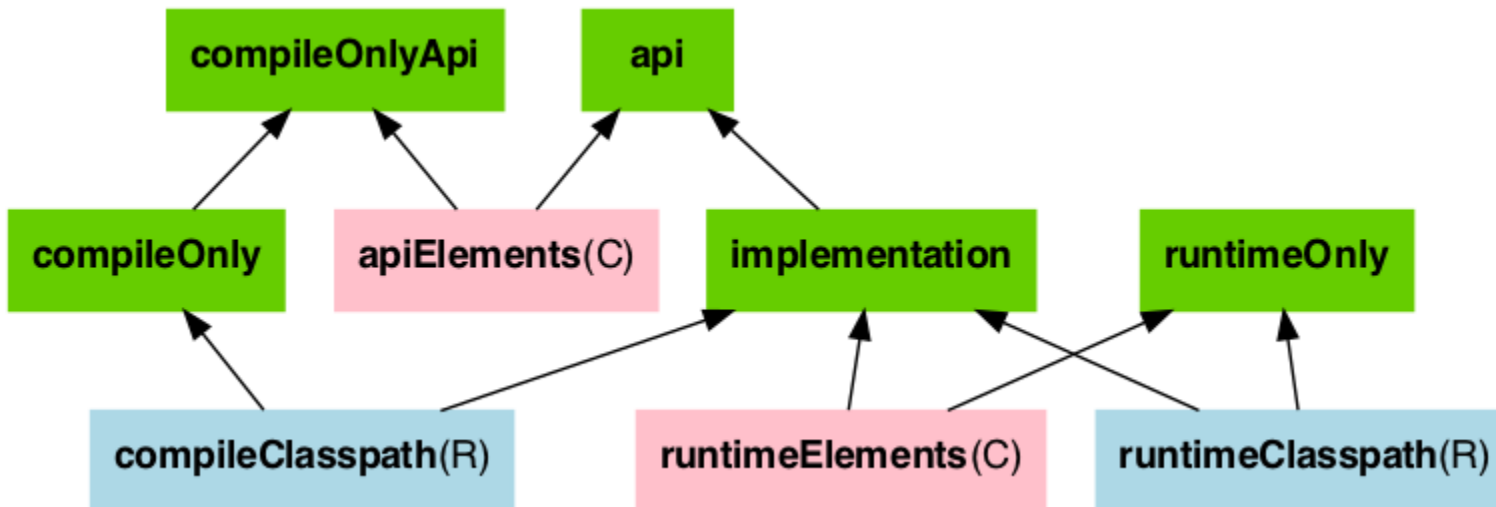
Be careful. Changing and dynamic dependencies break reproducibility of builds and cause Gradle to re-resolve dependencies frequently.

You can combine them with [dependency locking](#) to fix that.

Dependency Resolution

- Tasks can have *resolvable configurations* as inputs.
- Right before executing such a task, Gradle fetches the dependencies (including transitives).
 - `:compileJava` has the *compileClasspath* as input
 - `:run` has the *runtimeClasspath* as input.

Declared dependencies (green) are included in resolvable configurations (blue):



apiElements and *runtimeElements* (consumable configurations) represent the dependencies re-exported for compilation and runtime.

Transitive Dependencies

Gradle fetches dependencies of your dependencies. This can introduce version conflicts.

Only one version of a given dependency can be part of a resolution result.

Options for dealing with conflicts (details on next slides):

- Often the default behavior (highest version number wins) is sufficient
- Use [rich version constraints](#) to limit the range of allowed versions or enforce a single version
 - e.g., enforce a version, or a version range, with [strict versions](#)
- [Exclude](#) conflicting dependencies completely

Transitive dependencies (cont)

More advanced options:

- Use [component metadata rules](#) to fix transitive dependency declarations
- [Version alignment](#)
- [Handling mutually exclusive dependencies](#)
- [Customizing resolution through resolution rules](#)
- [Resolution strategy tuning](#)

Constraints with rich versions

- Declared in a **constraints {}** block
- Use rich versions (prefer, require, strictly, reject) - see [user manual](#) for details
- **Important** This doesn't add a dependency. This influences which version is selected

```
dependencies {
    constraints {
        api("com.google.guava:guava:24.1.1-jre!!") // shortcut for strictly
        api("com.google.inject:guice") {
            version {
                strictly("[4.0, 5.0)")
                // require("4.0")
                // prefer("4.2.0")
                // reject("4.2.1")
                // rejectAll()
            }
            because("Only version 4 of Guice has all DI features we need.")
        }
    }
}
```

Constraints with rich versions

- **require("4.0")** - Use version "4.0", higher versions are allowed in case there is a conflict
 - Most common constraint; a simple string version like "4.0" is treated as *require*
- **strictly("4.0")** - Use version "4.0"; if there is a conflict, still use "4.0" (or fail)
 - Short notation is "4.0!!"
- **strictly("[4.0, 5.0)")** - Use a version from range; other constraints may influence which one
- **reject("4.2.1")** - Whatever is selected due to other rules, never select "4.2.1"
- **rejectAll()** - Always fail if a dependency is in the result (for all versions of that dependency)

Excludes

If you depend on a library, and you do not need a transitive dependency because you do not use a certain part of that library, you can [exclude it](#).

```
dependencies {  
    implementation("commons-beanutils:commons-beanutils:1.9.4") {  
        // We only use 'PropertyUtils.setSimpleProperty()'   
        // which does not require commons-collections  
        exclude group: "commons-collections", module: "commons-collections"  
    }  
}
```

- Sometimes, excludes are used for things better solved with constraints
 - E.g. exclude a needed dependency just because it has the wrong version
- In Maven, excludes are often the only solution, because constraints are not supported
- In Gradle, double check if using exclude is the best solution

Unmanaged Dependencies

You can also define dependencies directly to local files. This is not recommended, but can be useful during migration from an existing build setup that does not have managed dependencies.

```
dependencies {  
    implementation(fileTree(dir: "lib", include: "*.jar"))  
}
```

Component Metadata

- Modules in repositories have metadata which e.g. contains the dependency definitions
- When downloading modules from repositories, Gradle fetches the metadata for additional information
 - E.g. transitive dependencies and their versions
- Gradle supports three metadata formats
 - Gradle Module Metadata (GMM): `.module` file
 - Maven POM: `.pom` file
 - Ivy XML: `ivy.xml` file
- GMM is a rich format and compliments one of the other formats
 - E.g. `org.junit.jupiter:junit-jupiter-api` has both a `.pom` and a `.module` file. Tools like Maven or older Gradle versions can only understand the pom file. Recent Gradle versions can read the `.module` file with more information about the module (e.g. rich versions).

Component Metadata Rules

Relying on the metadata of a component you do not publish yourself can lead to wrong dependency resolution results, if the metadata is incomplete or wrong.

- Many modules today are published with POM metadata only and miss information that cannot be represented in POM.
- Sometimes, a mistake was made and some information the metadata is wrong.

[Component metadata rules](#) are a mechanism to modify the metadata after it has been downloaded from a repository.

Dependency Cache

Gradle caches everything it downloads: artifacts (e.g. jars) and metadata.

Default location: `~/.gradle/caches/...`

- Multi-process safe
- Source location aware
- Optimized for reading (finding dependencies is fast)
- Checksum based storage
- Avoids unnecessary downloading
 - Finds local candidates
 - Uses checksums/etags

The cache can be [relocated](#) to avoid repeating downloads in ephemeral CI setups.

Dependency Cache

Dynamic and SNAPSHOT (changing) dependency information is only cached for a certain period
- default TTL is 24 hours.

```
configurations.all {  
    resolutionStrategy.cacheChangingModulesFor(4, "hours")  
    resolutionStrategy.cacheDynamicVersionsFor(10, "minutes")  
}
```

- `--offline` - don't look for updates, regardless of TTL
- `--refresh-dependencies` - look for updates, regardless of TTL
 - Gradle will re-download all metadata. Actual artifacts, like Jars, may be taken from cache if their content (checked via hash) has not changed.

Dependency Reports

View the dependency graph via built-in task.

```
$ ./gradlew :<subproject>:dependencies [--configuration «name»]  
  
$ ./gradlew :app:dependencies --configuration compileClasspath
```

View a dependency in the graph via built-in task.

```
$ ./gradlew :<subproject>:dependencyInsight --dependency «name»  
                                           --configuration «name»  
  
$ ./gradlew :app:dependencyInsight --dependency com.google.guava:guava  
                                   --configuration runtimeClasspath
```

Explore dependencies in a build scan - navigate to "Dependencies" tab.

```
$ ./gradlew :app:build --scan
```

Lab

03-dependency-management

Publishing

- Publish your artifacts to any Maven/Ivy repository
- Metadata file (`.module + .pom/ivy.xml`) is generated
- Repository metadata (e.g. `maven-metadata.xml`) is generated
- [Maven Publish Plugin](#) for publishing to Maven repositories
- [Ivy Publish Plugin](#) for publishing to Ivy repositories

Publishing to Maven Repositories

```
plugins {  
    id("java-library")  
    id("maven-publish")  
}  
  
publishing {  
    publications {  
        create<MavenPublication>("maven") {  
            // create("maven", MavenPublication) { <- Groovy DSL  
                from(components["java"])  
            }  
        }  
    }  
    repositories {  
        maven {  
            url = uri("https://my.org/m2repo")  
        }  
    }  
}
```

- For Artifactory, JFrog provides an `artifactory-publish` plugin

Structuring Builds

Structuring Builds

Two mechanisms

- Multi-project builds
 - One build with multiple subprojects
 - Usually released together
- Composite builds
 - Multiple builds put together into one product
 - Each build may release separately
 - Each build has one purpose

Real world examples

- [Spring Boot](#)
- [Gradle](#)
- [Groovy](#)
- [Spock](#)

Multi-project Builds

To structure one "software component" internally.

- Flexible directory layout
- Project dependencies & partial builds
- Each project can use different plugins

Composite builds

To structure a "software product" composed of multiple "software components".

- [Composite builds](#) are a way to combine multiple builds into a single build.
- A composite build is made up of a root build and one or more "included builds"
- You can use composite builds to combine independently developed builds or decompose a large build into separate chunks.
- To make one build know to each other, use `includeBuild("path/to/build")` in `settings.gradle(.kts)`
- Learn more from the [composite build samples](#).

We focus on multi-project builds today, but the sample we will look at could be converted into a composite build as well.

Defining a Multi-project Build

- Build structure is defined in `settings.gradle(.kts)`

```
// define the name of the build (defaults to directory name)
rootProject.name = "main"

// declare projects:
include("api", "shared", "webservice")
```

- You can customize location and build script name for each project (usually not needed)

```
// by default, api subproject is in directory 'api'
project(":api").projectDir = file("/myLocation")

// by default: build files are "build.gradle" or "build.gradle.kts"
project(":shared").buildFileName = "shared.gradle"
```

Task/Project Paths

- All projects have a path that uniquely identifies them.
- Gradle uses `:` as a path separator.
- When running tasks on the command-line, you can combine the project path and the task name to select a task to execute.
 - `:` refers to the root project
 - `:clean` means to run the clean task in the root project only
 - `:api` refers to the api project
 - `:api:clean` means to run the clean task in the api project only

Implicit task selection

Running a task found only in subprojects from the root project will implicitly execute those tasks in the subproject

Runs clean in all subprojects:

```
$ ./gradlew clean
```

Runs assemble in all subprojects:

```
$ ./gradlew assemble
```

Runs test in all subprojects:

```
$ ./gradlew test
```

Fuzzy name matching

Like tasks, fuzzy name matching works for project paths too.

If you had a project named `really-long-name`, you could run `clean` in that project with:

```
$ ./gradlew rLN:clean
```

Project Dependencies

Instead of using `group:name:version` to declare dependencies between projects, you can use project dependencies.

The method `project(String)` takes the path to the other project

```
dependencies {  
    implementation("commons-lang:commons-lang:2.4")  
    // Depends on the "shared" project  
    implementation(project(":shared"))  
}
```

Gradle automatically builds parts of the other project as needed (e.g. compiles classes for the `compileClasspath` or bundles a jar for the `runtimeClasspath`).

Defining shared build logic

- If you have multiple projects (or builds) you often want to share build logic
- Use *convention plugins* in a separate build for this.
 - You can use `buildSrc` which is a built-in included build. Just add a `buildSrc` directory into the root of your multi-project enables it.
 - Or include another build using the `includeBuild(...)` statement
- The `build.gradle(.kts)` file should apply one plugin for plugin development
 - `groovy-gradle-plugin` – write convention plugins as `.gradle` files in `src/main/groovy`.
 - `kotlin-dsl` – write convention plugins as `.gradle.kts` files in `src/main/kotlin`.
 - `java-gradle-plugin` – write convention plugins as `.java` classes that implement the `Plugin<Project>` interface in `src/main/java`.

Write a convention plugin

buildSrc/build.gradle

```
plugins {  
    id ("groovy-gradle-plugin") // Use Groovy DSL for plugins  
}  
  
repositories {  
    gradlePluginPortal()  
}
```

Write a convention plugin

buildSrc/src/main/groovy/java-library-conventions.gradle

```
// A convention plugin looks just like a build script
plugins {
    id("java-library")
}
tasks.named("test") {
    useJUnitPlatform() // Use JUnit 5 for unit tests
}
dependencies {
    testImplementation("org.junit.jupiter:junit-jupiter-api:5.6.2")
    testRuntimeOnly("org.junit.jupiter:junit-jupiter-engine")
}
```

lib/build.gradle

```
plugins {
    // Use our convention plugin instead of "java-library"
    id("java-library-conventions")
}
```

Lab

04-lab-build-logic

Custom Build Logic

Tasks and Files

- Files are predominately the inputs and outputs of tasks
- Configuration parameters (e.g. String, Integer, ...) can be also be used as inputs
- Gradle tracks the implementation of the task as an input
- Gradle offers its own types for representing files/directories
- Properties using these types have knowledge about the task that produces them
 - If an output of task A becomes the input of task B, Gradle automatically executes A before B

File/Directory types

- File/Directory types that are immutable:
 - `RegularFile` points at a concrete file
 - `Directory` points at a concrete directory
 - `FileCollection` a collection of files that can be located in different places
- File/Directory types that are configurable:
 - `RegularFileProperty` a file that can be changed during build configuration
 - `DirectoryProperty` a directory that can be changed during build configuration
 - `ConfigurableFileCollection` add/remove files during build configuration

File/Directory types (cont)

- Typically, you'll need locations relative to the project directory or build directory
 - `layout.buildDirectory.dir("myTargetDir")` directory relative to the build folder
 - `layout.buildDirectory.file("myTargetFile")` file relative to the build folder
 - `layout.projectDirectory.dir("myTargetDir")` dir relative to the project folder
 - `layout.projectDirectory.file("myTargetFile")` file relative to the project folder
- All file/directory types can be converted back to `java.io.File` during task execution.
 - `RegularFile.asFile`, `Directory.asFile`, `FileCollection.files()`
- Be careful to not do this during configuration time:
 - The property may not be configured yet
 - The file may not be created yet (this produces a deprecation warning)

Writing a task

- An abstract class that extends `DefaultTask`
- This can be written in pure Java, Groovy, Kotlin or another JVM language
- Define properties as annotated abstract getter methods (e.g., `getPropertyName()`)
 - `@InputFile`, `@InputFiles`, `@OutputFile`, `@OutputDirectory`
 - See for [user manual](#) more details
- Each task should have at least one input and one output
 - Gradle uses this to track if the task needs to be executed again or if the previous execution result can be reused
 - It is also used to cache results in the [build cache](#)
 - If input and/or outputs are missing, the task will never be `UP-TO-DATE` or `FROM-CACHE`
- Each task has at least one action implemented in a method annotated with `@TaskAction`
 - Usually you have only one action method
 - From there you can delegate to other methods or classes or even call external tools

Example task

```
public abstract class JarSize extends DefaultTask {

    @InputFile
    public abstract RegularFileProperty getJar();

    @OutputFile
    public abstract RegularFileProperty getJarSizeTxt();

    @TaskAction
    public void calculateSize() throws IOException {
        java.io.File jarFile = getJar().get().getAsFile();
        java.io.PrintWriter txtFile = new java.io.PrintWriter(getJarSizeTxt().get(
    ).getAsFile());
        txtFile.append("Jar size is " + jarFile.length());
        txtFile.close();
    }
}
```

Wiring inputs and outputs

- Use `tasks.register<TaskClass>("taskName") { /* task configuration */ }`
- You set the properties in the configuration block
- If you set an input property to the output of another task, Gradle will treat this as a task dependency

```
// Kotlin DSL
tasks.register<JarSize>("jarSize") {
    // Set the input to the 'archiveFile' output of the 'jar' task
    jar.set(tasks.jar.get().archiveFile)
    // Set the output to a file in the 'buildDirectory'
    jarSizeTxt.set(layout.buildDirectory.file("reporting/jarSize.txt"))
}
```

```
// Groovy DSL
tasks.register("jarSize", JarSize) {
    jar.set(tasks.jar.archiveFile)
    jarSizeTxt.set(layout.buildDirectory.file("reporting/jarSize.txt"))
}
```

Extensions

- Extensions are a central place to configure a plugin (e.g. `java { }`)
- Use extensions to avoid direct task configurations in build files
- Extensions are implemented as plain abstract classes, like tasks

Extension (definition)

- Define the extension type (e.g. as Java interface)

```
public interface JarSizeExtension {  
    RegularFileProperty getTxtFile();  
}
```

- In your plugin: create the extension and wire it to the task(s)

```
// Kotlin DSL  
val size = extensions.create<SizeExtension>("size")  
tasks.register<JarSize>("jarSize") {  
    jar.set(tasks.jar.get().archiveFile)  
    jarSizeTxt.set(size.txtFile)  
}
```

```
// Groovy DSL  
def size = extensions.create("size", SizeExtension)  
tasks.register("jarSize", JarSize) {  
    jar.set(tasks.jar.archiveFile)  
    jarSizeTxt.set(size.txtFile)  
}
```

Extension (usage)

- In the `build.gradle(.kts)` file you can then use the extension

```
size {  
    txtFile.set(layout.buildDirectory.file("custom/size.txt"))  
}
```

Lab

05-custom-tasks

Wrapping up

Performance features

Build caching

Gradle will [skip execution of some work](#) if it has been done before, even on other machines.

```
$ gradle build --build-cache
```

File system watching

Gradle will [watch files on disk](#) and skip to executing tasks more quickly.

```
$ gradle build --watch-fs
```

Parallel Builds

Run independent tasks from different projects in parallel.

```
$ gradle build --parallel
```

Useful features

Continue after Failure

```
$ gradle build --continue
```

Especially useful for CI builds.

Continuous Build

When the build completes, instead of exiting, [watch the inputs of executed tasks](#) and re-run the build when an input changes.

```
$ gradle build --continuous
```

Standard Gradle plugins

Gradle ships with many useful plugins.

Some examples:

- `java-library` - compile, test, and package Java projects
- `checkstyle` - static analysis for Java code
- `maven-publish` - upload artifacts to Apache Maven repositories
- `groovy` - compile, test, package, upload Groovy projects
- `scala` - compile, test, package, upload Scala projects
- `application` - support packaging your Java code as a runnable application
- `cpp-library` - support building native binaries using gcc, clang or visual-cpp


Many more, [listed in the Gradle User Manual](#).

Gradle Plugin Portal

- Search and discover community plugins on plugins.gradle.org/
- Plugin JARs and their metadata are hosted by Gradle Inc.



Search Gradle plugins

 com.gradle.enterprise

Want to include your Gradle plugin here?

Plugin

Latest Version

[com.gradle.enterprise](#)

3.4.1

Gradle Enterprise gives you the data to speed up your build, improve build reliability and accelerate build debugging.

(20 August 2020)

[#analytics](#) [#debugging](#) [#scans](#) [#performance](#) [#insights](#)

[com.gradle.enterprise.test-distribution](#)

1.1.2

Gradle Enterprise test distribution takes your existing test suites and distributes them across remote agents to execute them faster.

(28 August 2020)

[#test](#) [#performance](#)

Notable community plugins

- [Kotlin JVM plugin](#) - Builds Kotlin code
- [Spring Boot plugin](#) - Builds Spring boot applications
- [Shadow plugin](#) - Builds shaded jars
- [Spotbugs plugin](#) - Runs SpotBugs analysis on Java code

Other Gradle Inc plugins

- [Gradle Enterprise](#) - Generates build scans.
- [Gradle Enterprise Test distribution](#) - Distributes tests across multiple machines.
- [Gradle Test Retry](#) - Mitigate flaky tests by retrying tests when they fail.

Thank You!

- Thank you for attending!
- Questions?
- Feedback?
- gradle.org
- gradle.com