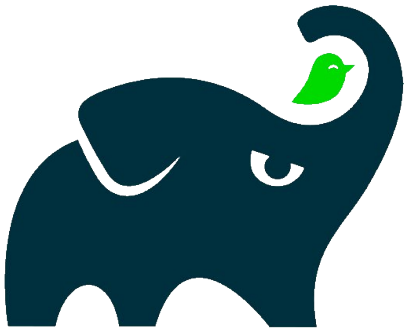


The Gradle build tool

2022-23

Introduction

- Gradle is a build and automation tool
  - Implemented in Java
  - Open source
  - Available stand-alone or integrated in IntelliJ Idea
- Used to build apps in different ecosystems
  - JVM-based (Java, Kotlin, Groovy, …)
  - Android
  - Native (C, C++, Swift, …)
  - Other (Python, Go, …)

General principles

- Convention based tool
  - Makes assumptions about the locations of source files, test files, resources, …
- Supports multi-project builds
- Easy customizable via scripts
- Support dependency management
  - Automatically downloading needed modules and libraries from listed repositories

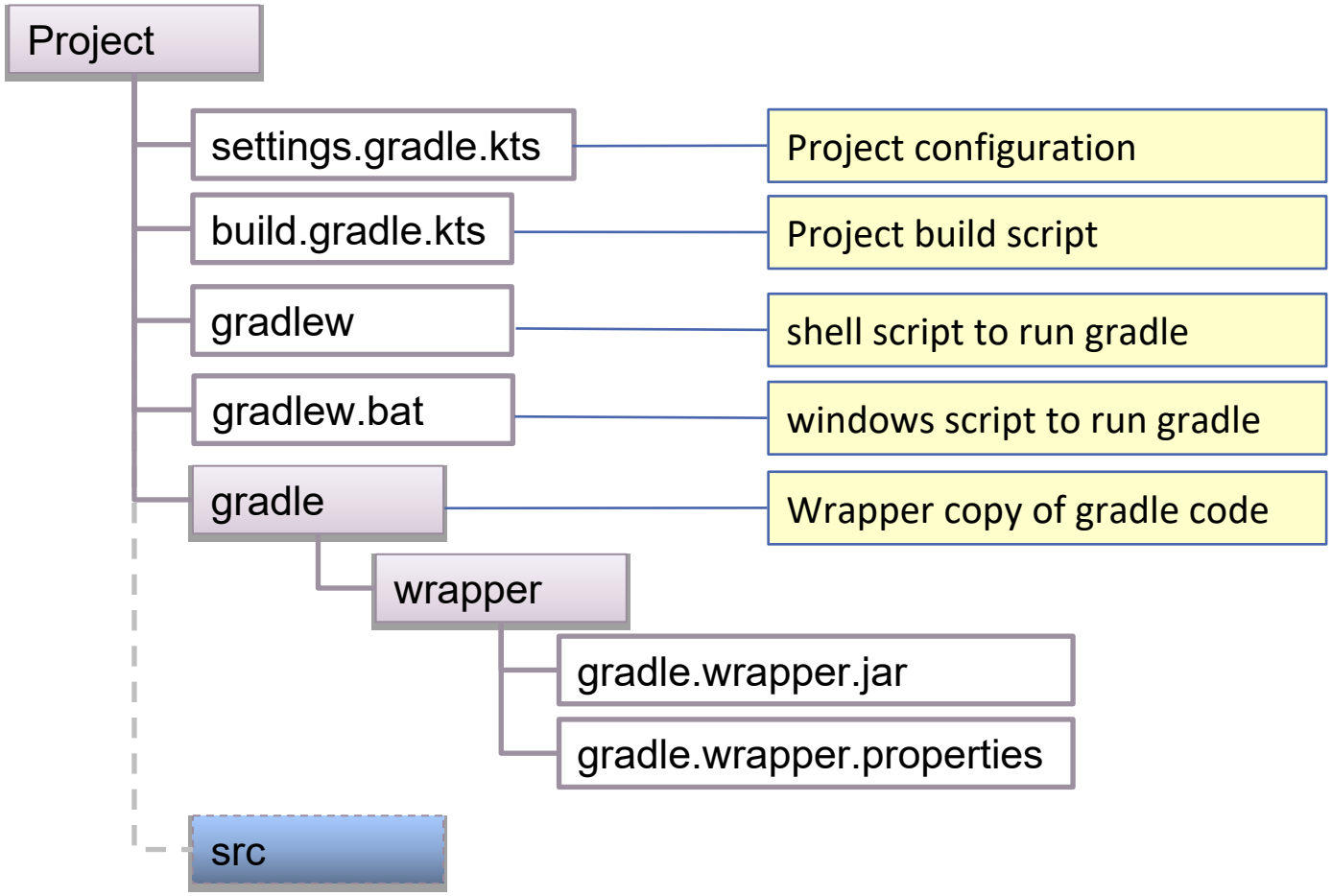
Describing the build process

- The entire build process is described in one or more script files
  - Listing the project configuration, dependencies, and plugins needed to automate the process and reuse functionalities
- Scripts can be authored in different languages (DSL)
  - Groovy
  - Kotlin

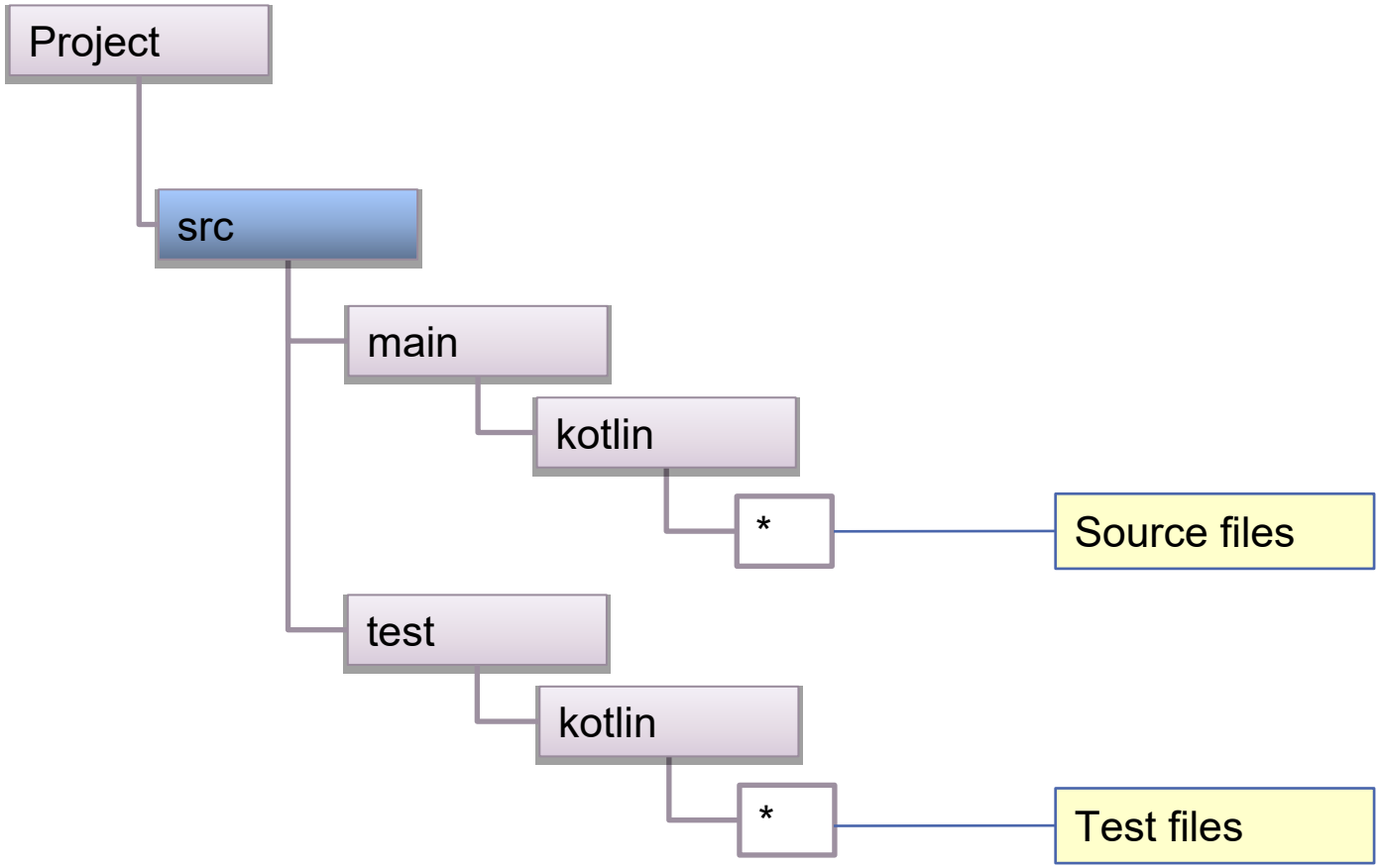
Initializing a Gradle project

- Using CLI
  - > mkdir project_name
  - > cd project_name
  - > gradle init
- Using IntelliJ Idea
  - File → New → Project → Gradle
- Online
  - Visit https://gradle-initializr.cleverapps.io
  - Download generated file

Project structure



Project structure



Project script file

- Contains a description of the build process, in terms of:
  - **plugins**, that define which tasks should be executed
  - **repositories**, that list web sites from which libraries can be downloaded
  - **dependencies**, i.e., a list of libraries with their version that are part of the project
  - further information about the project configuration

Gradle projects

- A project is the description of how a given software artifact is built
  - Sometimes, more than one artifact need to be built: in this case a build can have more than one project
- A project has one more more tasks
  - Aimed at a specific operation
- Plugins extend projects
  - Adding custom behaviours
- The project script file (build.gradle.kts) specifies all the tasks
  - An optional settings file (settings.gradle.kts) provides build level information (i.e., pertaining all projects)

Project script file

```
plugins {
  kotlin("jvm") version "1.6.10"
  application
}

repositories {
  jcenter()
}

dependencies {
  implementation(
     platform("org.jetbrains.kotlin:kotlin-bom"))
  implementation(
      "org.jetbrains.kotlin:kotlin-stdlib-jdk8")
  testImplementation("org.jetbrains.kotlin:kotlin-test")
}
application {
  mainClass.set("g1.AppKt")
}
```

Gradle tasks

- The build system is triggered by a set of commands, each targeted to a specific goal
  - All these are available from the CLI, thus making it possible to easily script them
  - IDEs make these commands available also from GUI
- Basic command structure
  - ./gradlew <task_name>
  - Using ./gradlew guarantees that the correct version of gradle is used
- The set of all available task is accessible using the "tasks" task name

Standard task

- Application tasks
  - run → executes the application
- Build tasks
  - assemble → Assembles the outputs of the project
  - build → Assembles and tests the project
  - buildDependents → Assembles and tests the project and all projects that depend on it
  - buildNeeded → Assembles and tests the project and all projects it depends on
  - classes → Assembles main classes
  - clean → Deletes the build directory
  - jar → Assembles a jar archive containing the main classes
  - testClasses → Assembles test classes

Standard tasks

- Build setup tasks
  - init → Initializes a new Gradle build
  - wrapper → Generates Gradle wrapper files
- Distribution task
  - assembleDist → Assembles the main distributions
  - distTar → Bundles the project as a distribution
  - distZip → Bundles the project as a distribution
  - installDist → Installs the project as a distribution as-is
- Documentation tasks
  - javadoc → Generates Javadoc API documentation

Standard tasks

- Help tasks
  - buildEnvironment → Displays all buildscript dependencies
  - components → Displays the components produced by the project
  - dependencies → Displays all dependencies
  - dependencyInsight → Displays the insight into a specific dependency
  - dependentComponents → Displays the dependent components of components in the project
  - help → Displays a help message

Standard tasks

- Help tasks (continues)
  - kotlinDslAccessorsReport → Prints the Kotlin code for accessing the currently available project extensions and conventions
  - model → Displays the configuration model
  - outgoingVariants → Displays the outgoing variants
  - projects → Displays the sub-projects
  - properties → Displays the properties
- Verification tasks
  - check → Runs all checks
  - test → Runs the unit tests

Custom tasks

- A task reference may be introduced by delegation
  - Standard tasks are referenced via

    ```
    val check by tasks.existing(...)
    ```
  - Custom tasks are introduced by

    ```
    val myTask by tasks.registering(...)
    ```
- Getting a reference to a task allows to customize its behaviour

Task execution

- Each task has some work to be done at the beginning
  - Specified via the `doFirst(…)` method
- Some work to be performed at the end
  - Specified via the `doLast(…)` method
- Some conditional work
  - Specified via the `onlyIf(…)` method

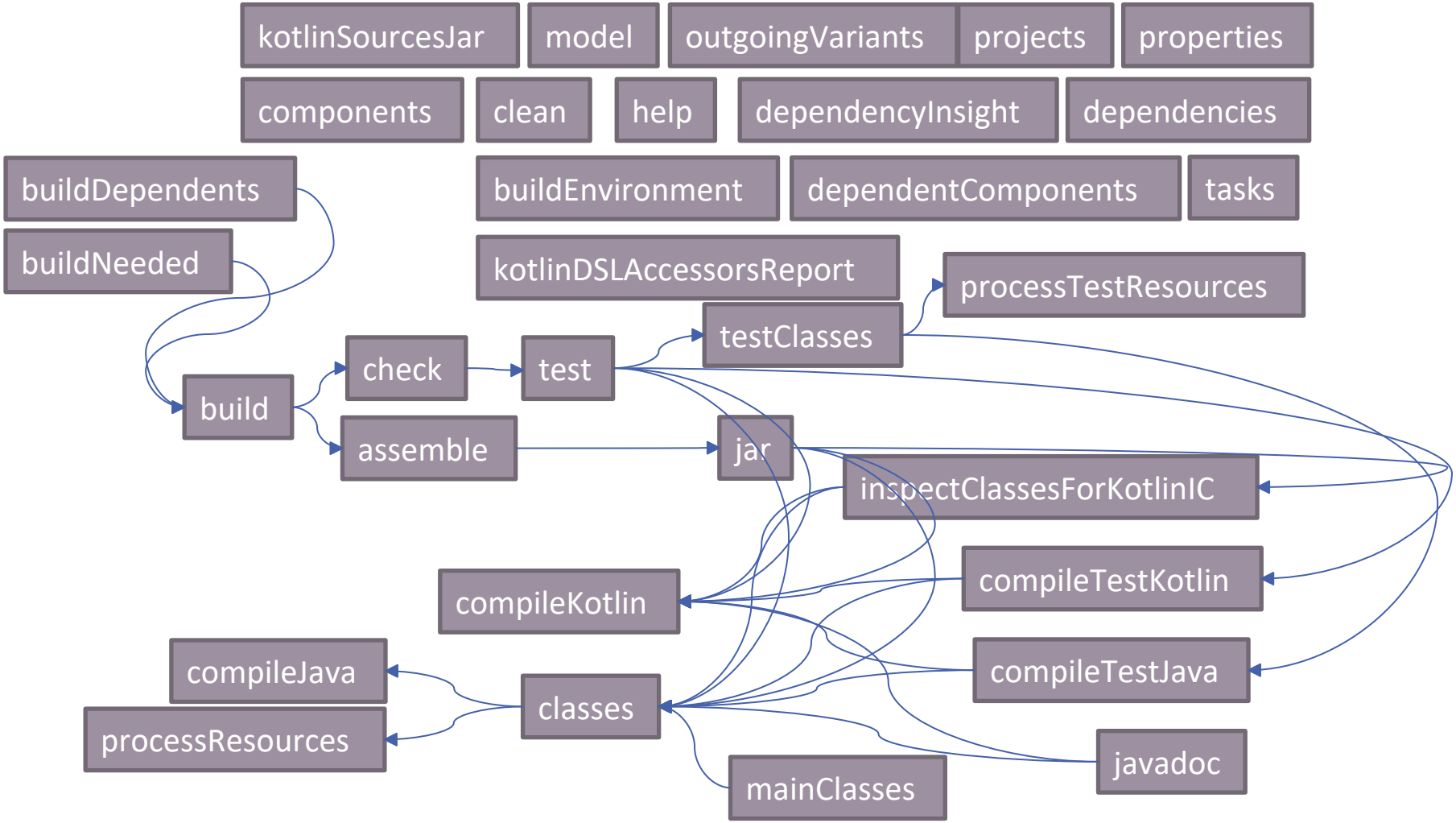
Task dependencies

- Each task can depend on zero or more other tasks
  - If Gradle is requested to execute a given task, all the tasks it depends on are executed first
- Standard tasks already have their dependency list set, custom tasks do not
  - In any case, it is possible to add extra dependencies via the dependsOn(…) function

```
val hello by tasks.registering
val compileKotlin by tasks.existing(KotlinCompile::class)

hello { doLast { println("Going to compile…") } }
compileKotlin { dependsOn(hello); }
```

Standard task dependencies



Plugins

- Software artifacts that extends project's capabilities
- Introduced by the plugins block
  - Plugins are either 'well-known' (maintained by Gradle) or community maintained
- Each plugin can introduce custom tasks, set-up dependencies, trigger specific behaviour

Core plugins

- JVM languages and frameworks
  - java - Provides support for building any type of Java project
  - `java-library` - Provides support for building a Java library
  - `java-platform` - Declares a set of modules that are published together or a set of recommended versions of heterogeneous libraries
- Packaging and distribution
  - application - Provides support for building JVM-based, runnable applications
  - `maven-publish` - Publishes build artifacts to an Apache Maven repository
  - distribution - Makes it easy to create ZIP and tarball distributions

Core plugins

- Code analysis
  - checkstyle - Performs quality checks on your project's Java source files using Checkstyle
  - pmd - Performs quality checks on your project's Java source files using PMD
  - jacoco - Provides code coverage metrics for Java code via integration with JaCoCo
- Utility
  - base - Provides common lifecycle tasks, such as clean
  - signing - Adds the ability to digitally sign built files and artifacts
  - `project-report` - Helps to generate reports containing useful information about your build

Community plugins

- Kotlin
  - `kotlin(platformName)` – Provides support for building a Kotlin project on the specified platform ("jvm", "js", "multiplatform")
  - `kotlin("kapt")` – Provides support for augmenting the Kotlin compiler with custom annotation processors

Describing a project structure

- By default, Gradle operates on conventions
  - It expects source files to be located in folder "./src/main", test files in "./src/test", …
  - It is possible to deviate from these conventions by specifying proper entries in the project script file
- The sourceSets entry can be used to define alternate folders for source and test files

```
sourceSets {
  main {
   java { setSrcDirs(listof("src/core")) }
  }
  test {
   java { setSrcDirs(listof("src/coreTest")) }
}
```

Creating a runnable application

- By applying the 'application' plugin, the 'run' task becomes available
  - It allows to launch the artefact that has been built
  - The application plugin extends the java plugin
  - the application block is used to specify the name of the main class

```
plugins {
  application
}

application {
  mainClassName = "com.something.Main"
}
```

Setting the java environments

- When the java plugin (or any other plugin that extends it) is applied, the java block can be used to configure the compilation and runtime environments
  - As well as specify further build instructions

```
java {
    sourceCompatibility = JavaVersion.VERSION_1_8
    targetCompatibility = JavaVersion.VERSION_1_8
    withJavadocJar()
    withSourcesJar()
}
```

Compiling Kotlin

- The kotlin plugin must be applied, specifying the language version
  - The kotlin block can then be used to specify alternates folder for source and test files, if needed
  - In order to define the proper jvm target version, the compileKotlin and compileTestKotlin tasks can be overridden

```
plugins {
  kotlin("jvm") version "1.6.10"
}
tasks {
  compileKotlin { kotlinOptions.jvmTarget = "1.8" }
  compileTestKotlin { kotlinOptions.jvmTarget="1.8"}
}
```

Adding code dependencies

- A project often requires libraries or other externally provided resources
  - These are collectively named dependencies
  - Being out of control of the project, care should be taken to identify the correct version of the artefact and location where it can be downloaded from
- Gradle allows to specify these information in different blocks
  - The dependencies block lists the artefacts needed by the project, together with their version
  - The repositories block list the web sites where they should be looked up

Transitive dependencies

- Some dependencies may depend on other libraries
  - The latter are named "transitive dependencies"
- Gradle takes care to locate and download all what is needed
  - `gradle -q dependencies` lists all project dependencies
  - `gradle -q dependencies --configuration implementation` lists dependencies needed throughout the artefact lifecycle

Code dependencies

- Dependencies can be located
  - In other projects
  - In the local file system
  - In Maven repositories
  - In Ivy repositories
- Dependencies are needed
  - At compile time and all though the artifact's life
  - At compile time only (being provided from other sources at runtime)
  - At runtime only
  - At test compilation time
  - At test execution time

Repositories

- The repositories block is used to indicate where dependencies are downloaded from
  - The `flatDir` sub-block indicate some locations in the local file system
  - `mavenCentral()` refers to Maven Central, a popular repository hosting open source libraries for consumption by Java projects
  - `jcenter()` returns a reference to Bintray's JCenter, an up-to-date collection of all popular Maven OSS artifacts
  - `google()` points to the Google repository, that hosts Android-specific artifacts including the Android SDK

Repositories

- Many organizations host dependencies in an in-house repository only accessible within the company's network
  - Gradle can declare Maven and Ivy repositories by URL
  - Credentials may be added in order to support different type of authentication

Repositories

```
plugins {
  application
}

repositories {
  flatDirs {
    dirs("./lib")
  }

  maven {
    uri("https://mycompany.com/maven2")
    credentials {
       username = "user"
       password = "pass"
    }
  }

  mavenCentral()
}
```

Dependencies configuration

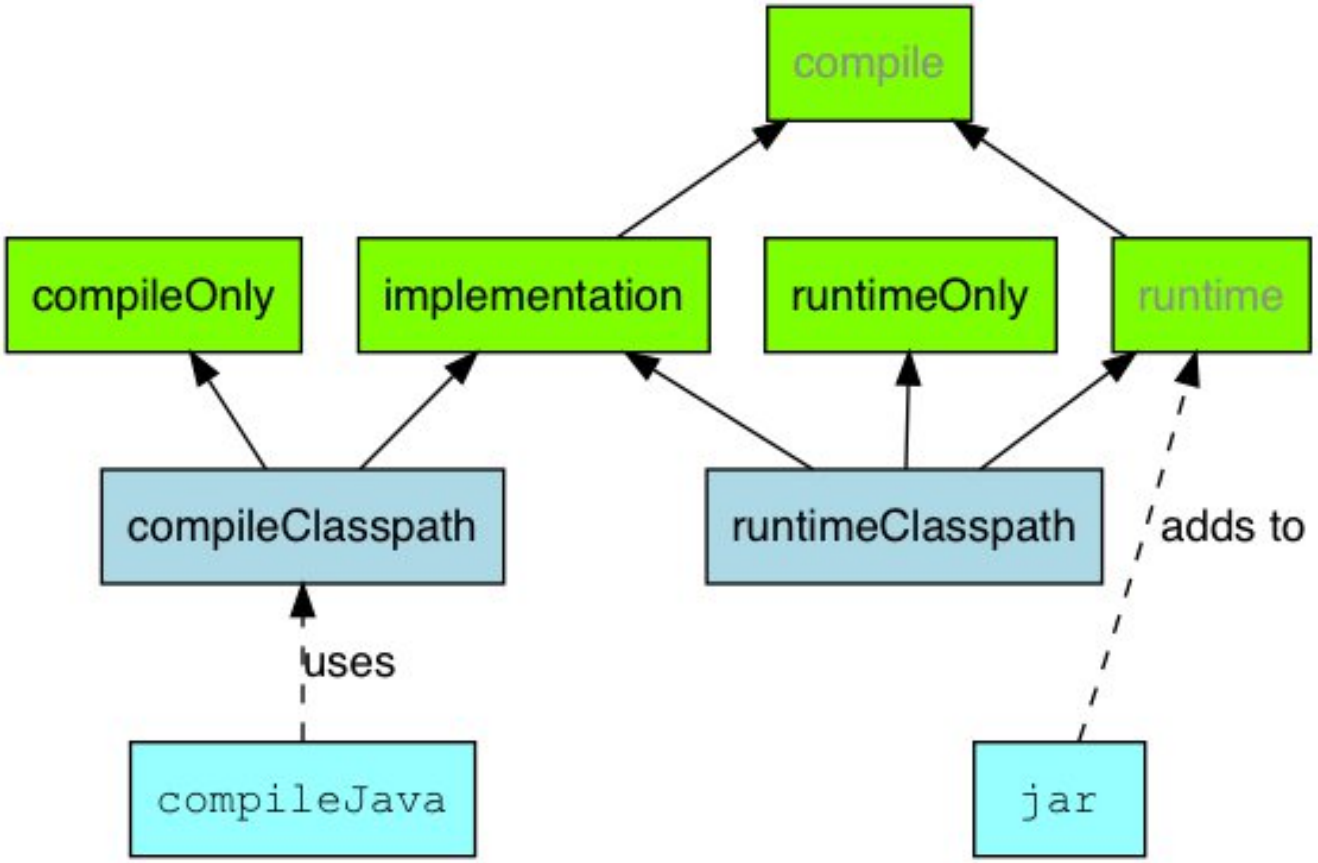
- Given a set of locations where dependencies may be download from, two more pieces of information are needed
  - What exact piece of dependency is needed
  - In which phase of the build process it is needed
- Dependencies are named out of three pieces of information
  - The group name
  - The artifact name
  - The artifact version
- These are written in a colon-separated string
  - `io.vertx:vertx-core:3.5.3`

Specifying the configuration

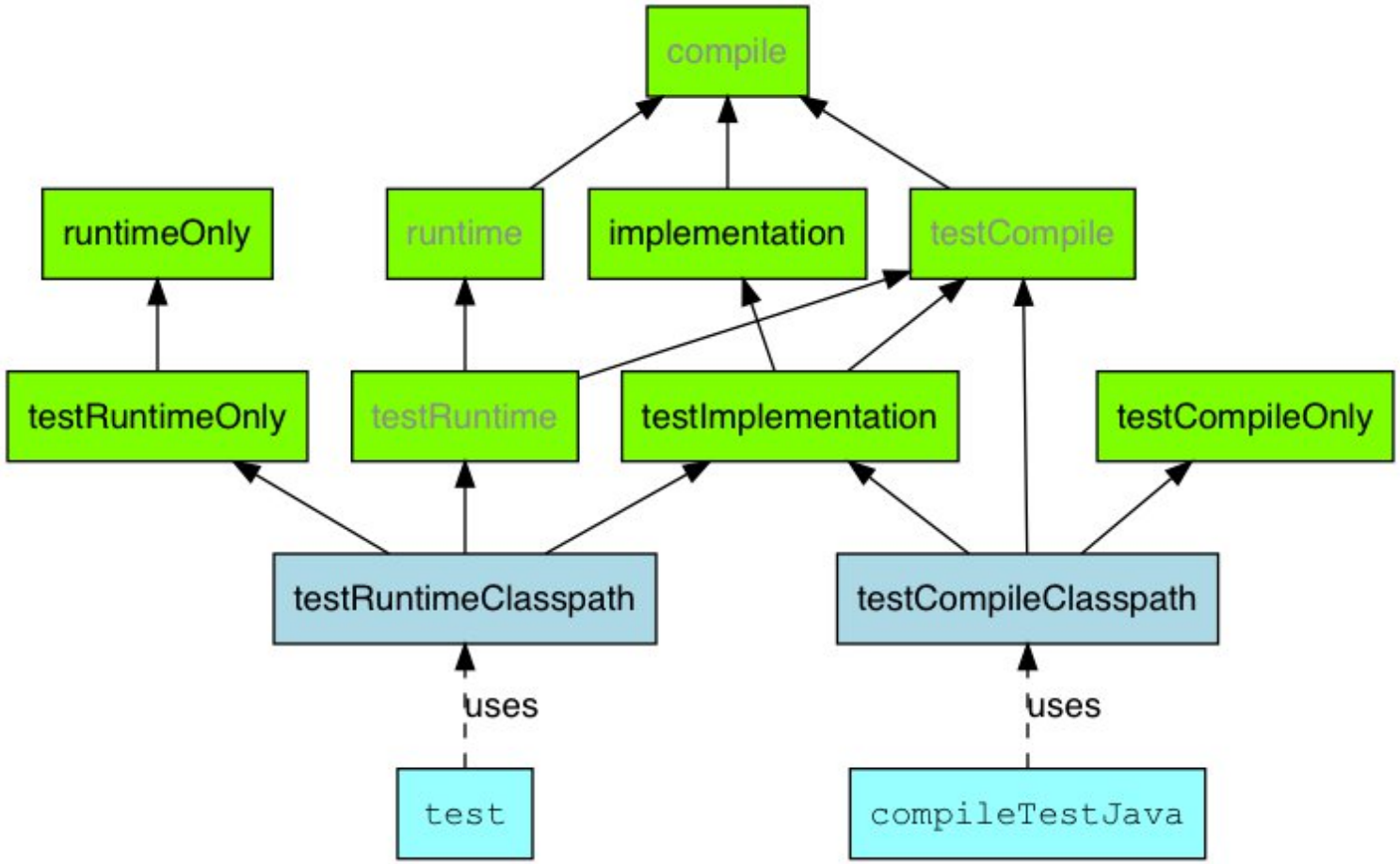
- The dependencies block introduces the list of dependencies of the current project
  - Each one is expressed as an argument of the corresponding configuration function
  - Functions differ in terms of visibility of the artefact at compile-time and/or runtime

```
dependencies {
    implementation("org.hibernate:hibernate-core:5.6.5")
    api("com.google.guava:guava:31.0.1-jre")
    testImplementation("junit:junit:4.+")
}
```

Main source files java dependency configuration



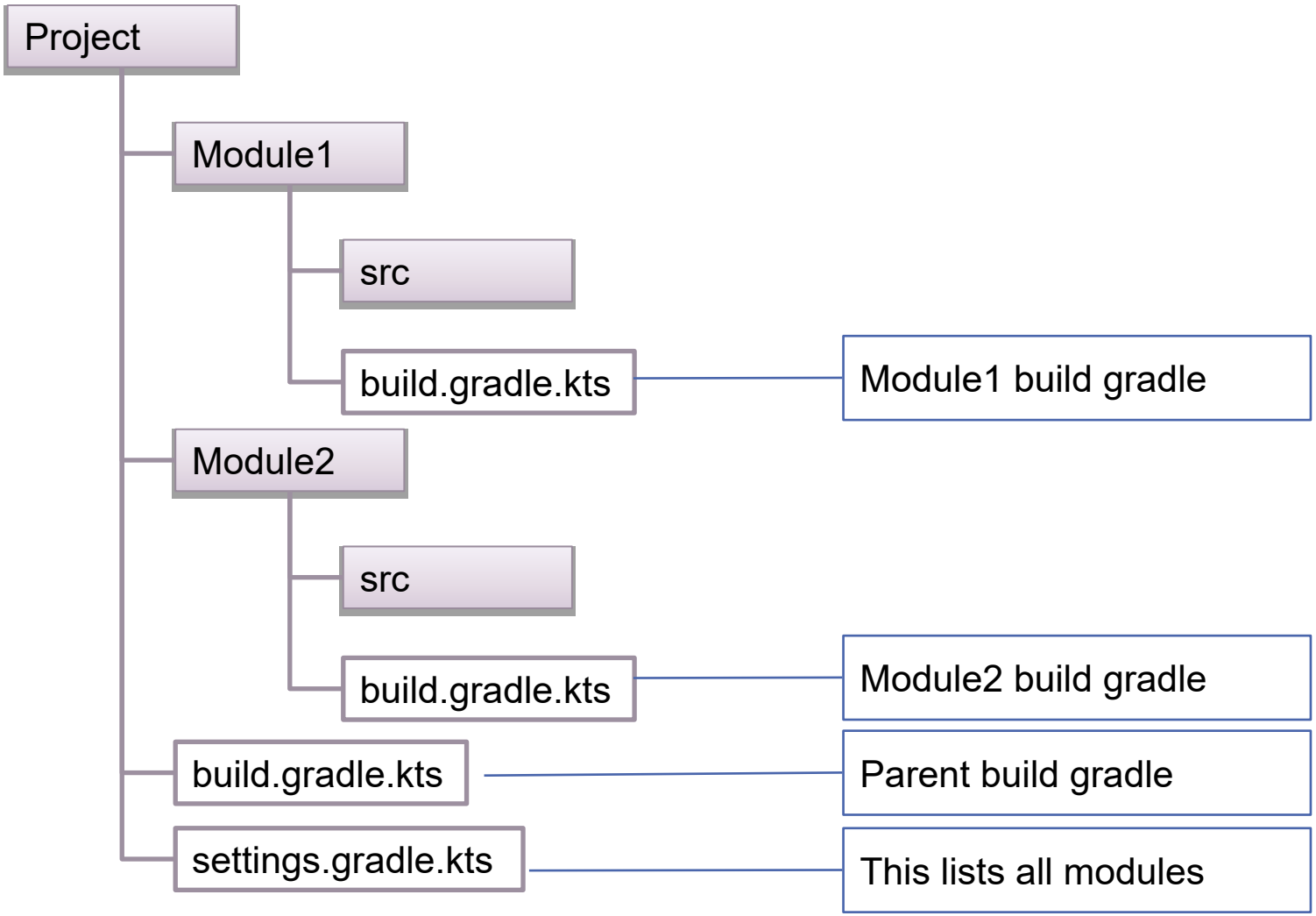
Test source files java dependency configuration



Modules

- Most projects start with a single codebase and build script (build.gradle.kts)
  - But, as they grow, they are often split into several interdependent modules
  - In order to improve readability and maintainability
- A module is a sub-project that targets the creation of an individual software artifact (e.g., a jar file)
  - Modules have their own codebases
  - They can have different dependencies

Multi-module project structure



Parent Build Gradle file

```
buildscript {
  repositories { mavenCentral() }
}
plugins {
  kotlin("jvm") version "1.6.10"
}

allprojects {
  group = "com.example"
  version = "0.0.1-SNAPSHOT"
  tasks.withType<KotlinCompile> {
    kotlinptions {
      jvmTarget = "1.8"
    }
  }
}
subprojects {
  repository { mavenCentral() }
}
```

Modules Build Gradle file

```
plugins {
  kotlin("jvm")  //no version here!
}

dependencies {   //repositories are inherited
  implementation(kotlin("stdlib-jdk8"))
  implementation(kotlin("reflect"))
}
```

Module1/build.gradle.kts

```
plugins {
  kotlin("jvm")  //no version here!
}

dependencies {   //repositories are inherited
  implementation(project(":module1")) //cross-reference
  implementation(kotlin("stdlib-jdk8"))
  implementation(kotlin("reflect"))
}
```

Module2/build.gradle.kts

Settings Gradle

- Here the overall project configuration is set-up
  - Modules need to be included in order to be built

```
rootProject.name = "example"

include("module1", "module2")
```