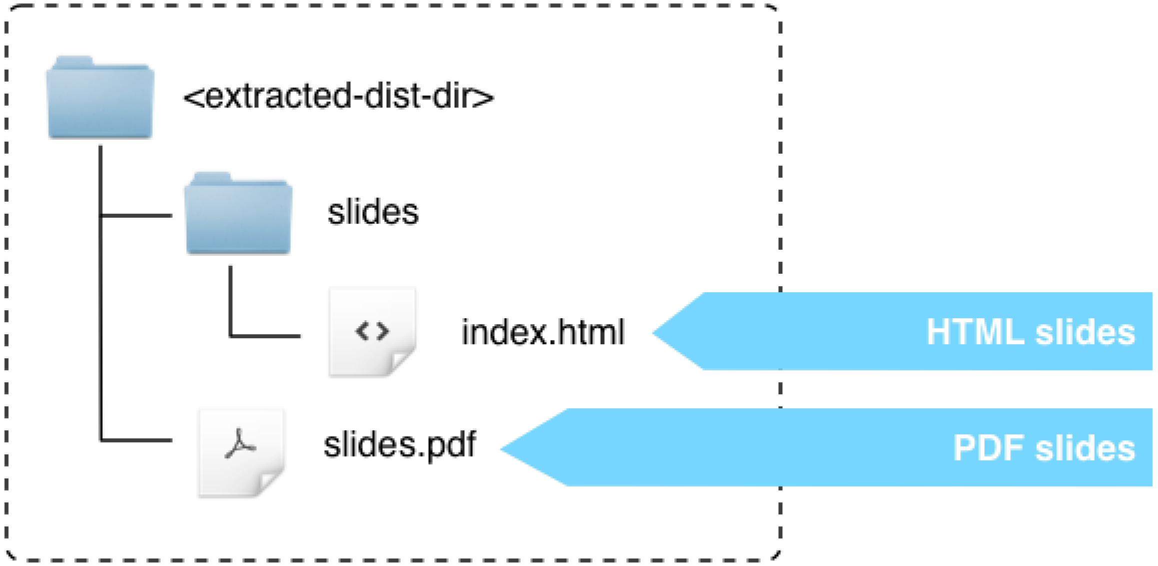
Advanced Fundamentals

In-depth with Gradle for Java projects

Gradle

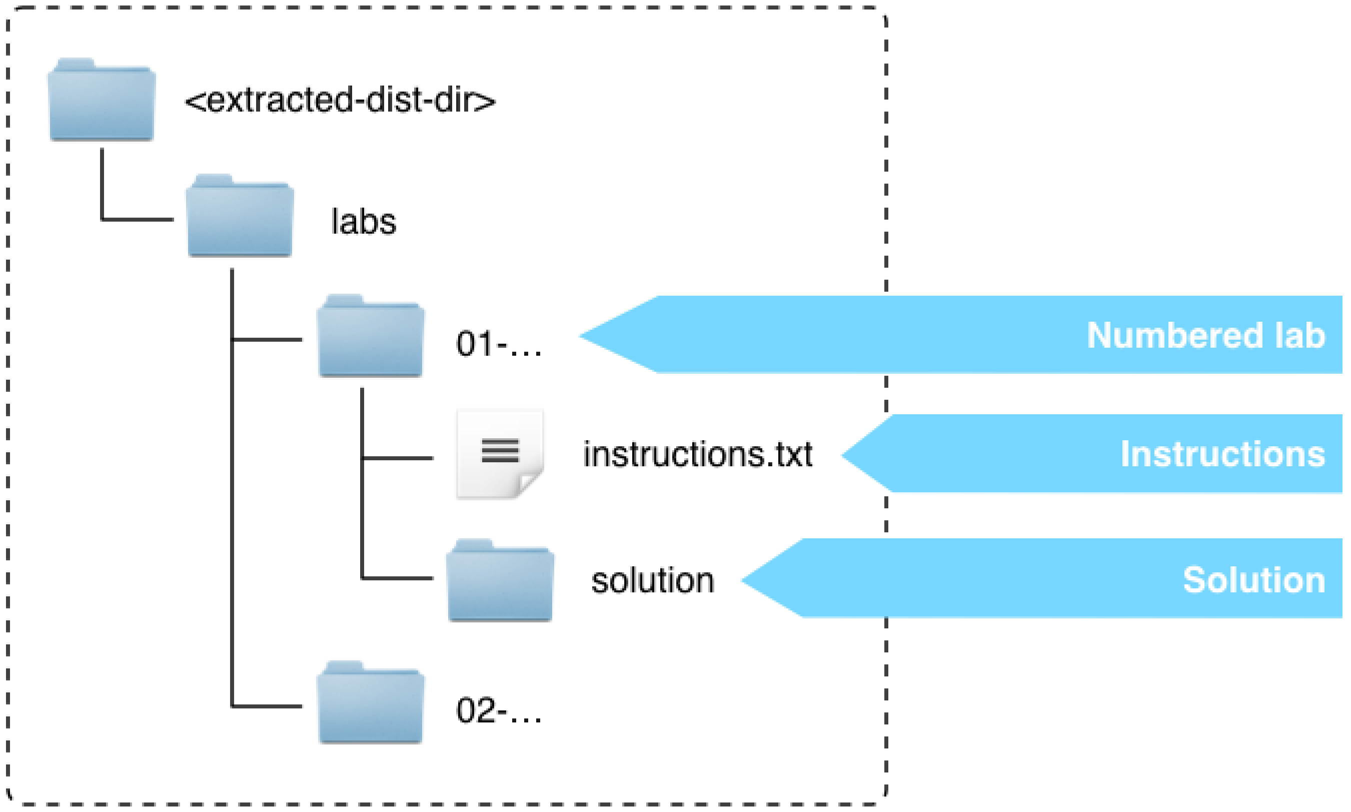
Slides

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



Practical labs

- Solutions are available (but don't overuse them)
- Take your time and experiment
- The labs are not a test!



Ask questions

- Please ask questions at any time!
- You control the speed of the presentation
- Q&A session at the end of the workshop

Objectives

Proficiency as a Gradle build master.

Understanding of:

- Core Gradle concepts, principles and philosophies
- The Gradle domain model

Being able to methodically create Gradle builds, rather than just adapting examples.

Specific Topics

- Gradle DSL basics
- Tasks & the Task Graph
- Build execution lifecycle
- The Plugin mechanisms
- Built-in tasks and plugins
- Dependency management
- Publishing
- Input and output
- Java support
- Multi-project builds
- Organizing logic and plugins
- Extensibility (init scripts, listeners …)
- Gradle Wrapper

And more.

Prerequisites

1. Ability to read/write Java/Groovy code
2. Familiarity with Gradle build scripts
3. Basic understanding of Gradle tasks, dependencies and building Java projects

Gradle Build Scans

Creating build scans

- Creating a build scan is free.
- Build scans are a permanent, 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 a Gradle, Inc server**. A self-hosted version is available.
- See Gradle Build Scans for more information.

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`.

Adding the Build Scan Plugin and License

Prior to Gradle 4.3, you needed to add the Build Scan plugin to your build and agree to the terms of service.

```
plugins {
    id 'com.gradle.build-scan' version '1.13.1'
}

buildScan {
    licenseAgreementUrl = 'https://gradle.com/terms-of-service'
    licenseAgree = 'yes'
}
```

In Gradle 4.3 and above, this plugin is applied and configured automatically if you run with the `--scan` flag. You will be prompted to accept the license on the command line.

Lab

01-create-build-scan

Tasks

DSL Syntax and Tasks

```
// << is synonymous with doLast()
// we'll use doLast() from here on
task hello << { println "Hello" }

// access existing task via its name
hello.dependsOn otherTask

// configure existing task via closure
hello {
  dependsOn otherTask
}

// configure new task
task greet {
  dependsOn otherTask
  doLast { println "Hello Gradler!" }
}
```

Quick Quiz

What does each individual line do?

```
task whatAmIDoing
whatAmIDoing
tasks.whatAmIDoing
whatAmIDoing {}
whatAmIDoing << {}
```

Ad-hoc vs Typed Tasks

```
task hello {
    onlyIf { day == "monday" }
    doFirst { println "Hello" }
}
```

Ad-hoc tasks implementations are written in the build script using `doLast()` or `doFirst()`.

```
task copy(type: Copy) {
  from "someDir"
  into "anotherDir"
}
```

Typed tasks are *configured* in the build script. Implementation is provided by the `Copy` class.

Implementing Task Types

- POJO extending `DefaultTask`
- Declare action with `@org.gradle.api.tasks.TaskAction`

```
class FtpTask extends DefaultTask {
  String host = "docs.mycompany.com"

  @TaskAction
  void ftp() {
    // do something complicated
  }
}
```

Task Type > Ad-hoc Task

Prefer implementing task types to implementing ad-hoc tasks.

- Avoid global properties and methods
- Separate the imperative from the declarative
- Easy to refactor (e.g. from build script to Jar)
- Easier to utilize other Gradle features

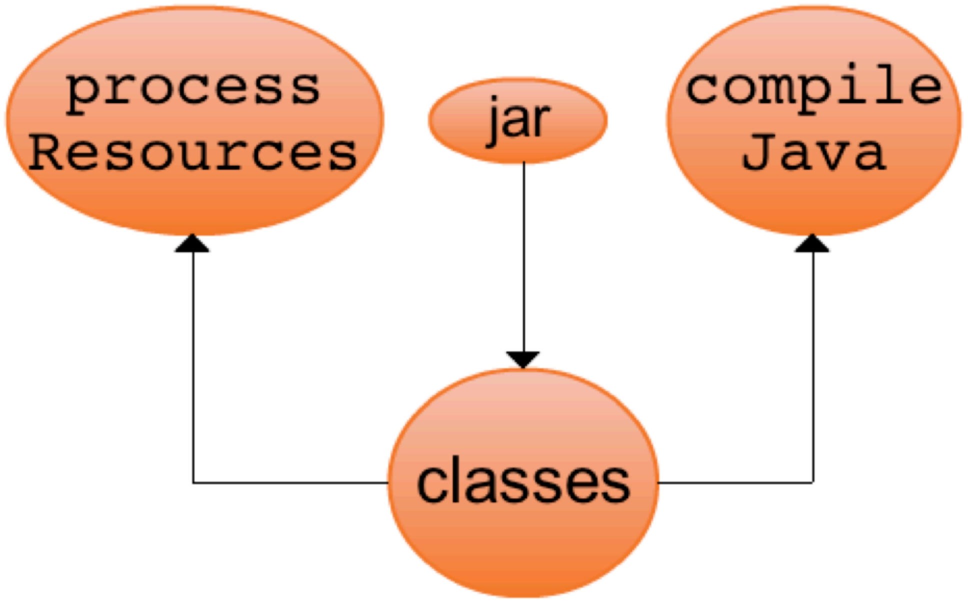
Ad-hoc tasks are OK for small simple tasks.

Lab

02-custom-tasks

Task Execution Graph

Task Execution Graph



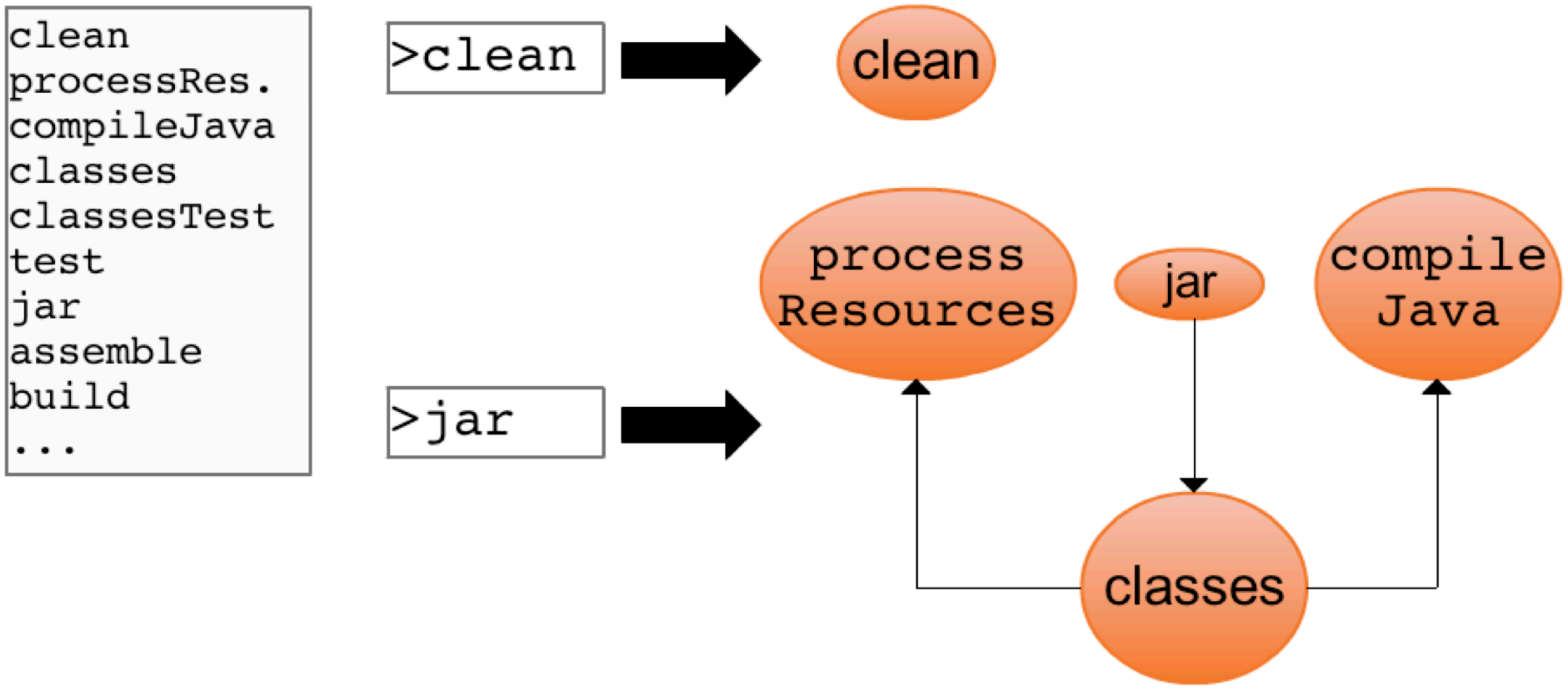
Before execution phase, Gradle arranges tasks into execution graph.

- Each task to be executed is a node
- The dependsOn relations define directed edges
- No cycles are allowed (acyclic)

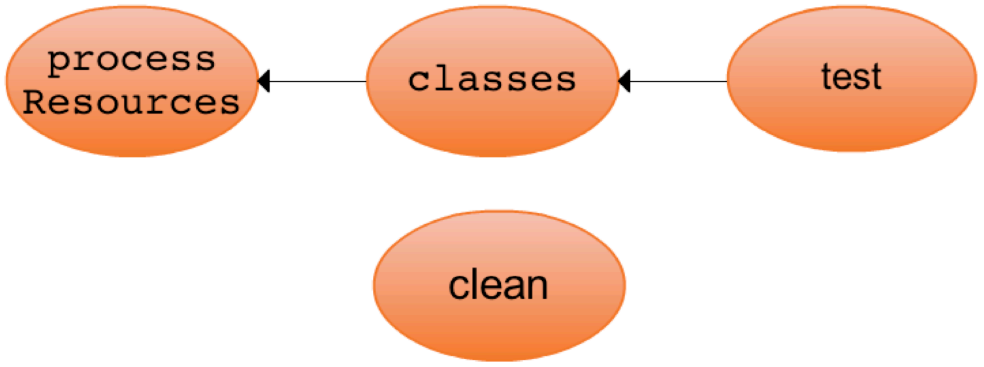
Known as a Directed Acyclic Graph (DAG).

Building the Task Ex. Graph

Running: "gradle clean jar"



Task is executed at most once



```
//'build' task runs only once:
> gradle build build

//'classes' task runs only once:
> gradle clean classes test

//'classes' task runs twice:
> gradle clean classes; gradle test
```

Lab

03-task-graph

Task Ordering

The order that tasks are executed in can be optimized.

```
task unitTests {}

task integrationTests {
    mustRunAfter unitTests
    // or: shouldRunAfter unitTests
}
```

Without instruction, task order is undefined.

Task Finalization

Runs a task even if a preceding task has failed.

```
task startWebServer {}
task stopWebServer {}

task integrationTests {
    dependsOn startWebServer
    finalizedBy stopWebServer
}
```

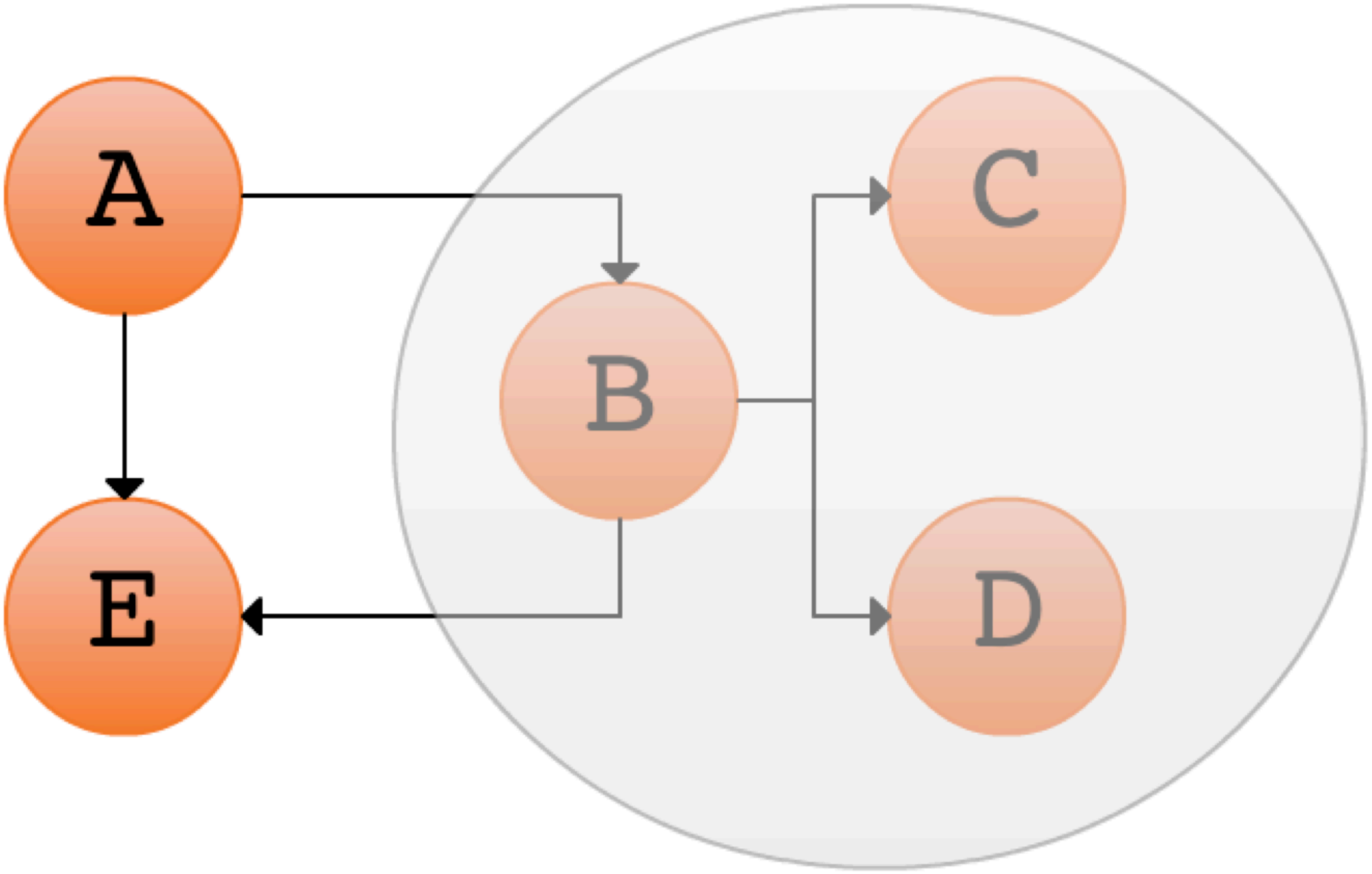
Often used for releasing resources (cf. Java's try-finally).

Lab

04-order-and-finalize

Excluding task subgraph

```
> gradle A -x B
```



Programmatic Exclusion

```
gradle.startParameter.excludedTaskNames.add "jar"
```

Skipping Tasks

- Actions are not executed
- Skipped tasks are part of the task execution graph
- Useful if task execution depends on runtime state

```
myTask.onlyIf { hasDocTaskGeneratedTitle() }
```

Adding Tasks Conditionally

```
if (isReleaseManagerUser()) {
  task ftpDistribution {
    doLast {
      // do something
    }
  }
}
```

Lab

05-excluding-tasks

Querying the Task Ex. Graph

- Gradle provides full access to the execution plan
- Fail fast if certain properties are not set
- Make decisions based on what will be executed

Querying the Task Ex. Graph

```
gradle.taskGraph.whenReady { graph ->
  if (graph.hasTask(":release")) {
    if (!project.hasProperty("releaseUsername")) {
      throw new GradleException("releaseUsername is not set")
    }
  }
}

task someTask {
  doLast {
    if (gradle.taskGraph.hasTask(":otherTask")) {
      // do something
    }
  }
}

// What happens here?
println gradle.taskGraph.allTasks
```

Lab

06-using-the-task-graph

Task Rules

Create tasks on demand, usually by naming patterns.

```
tasks.addRule("Pattern: ping<ID>") { String taskName ->
  if (taskName.startsWith("ping")) {
    task(taskName) {
      doLast {
        println "Pinging: " + (taskName - "ping")
      }
    }
  }
}

task groupPing {
  dependsOn pingServer1, pingServer2
}
```

Used to provide dynamic upload, build and clean tasks.

Shown at the bottom of `gradle tasks` output.

Logging

Logging

- 6 log levels: `error`, `quiet`, `warning`, `lifecycle`, `info`, `debug`
- Default log level (`lifecycle`) is minimalistic
- Command line options for setting different level

```
> gradle -i hello
> gradle hello -d
> gradle -q hello
```

When troubleshooting, `info` level is usually the most useful.

The `debug` level has a lot of output.

Logging from the Build Script

`Project` and `Task` objects come with a logger.

```
println "A message logged at QUIET level"

logger.quiet "A message that is always logged."
logger.error "An error log message."
logger.warn "A warning log message."
logger.lifecycle "A lifecycle log message."
logger.info "An info log message."
logger.debug "A debug log message."

task myTask {
  doLast {
    logger.info "Doing cool stuff…"
  }
}
```

Logging from Classes

```
import org.gradle.api.logging.Logger
import org.gradle.api.logging.Logging

Logger logger = Logging.getLogger("some-logger")
logger.info("An info log message")
```

Standard Gradle types expose a logger (e.g. `Task.getLogger()`)

Log messages from other logging toolkits are picked up:

- SLF4J
- Java Util Logging
- Jakarta Commons Logging
- Log4j

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 of the things plugins typically do:

- **Extend the Gradle model** with new elements (e.g. Java plugin's `sourceSets`)
- 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

Plugin Packaging

Plugins can be implemented as *scripts* or *classes*.

- Script plugins are just additional Gradle build scripts.
- Binary plugins are classes that implement the `Plugin` interface.

Plugins typically *apply* to the `Project` object, but not necessarily.

Plugins are applied using the Project.apply() method.

Script Plugins

Script plugins are trivially easy to write and consume.

`myPlugin.gradle`:

```
task taskFromPlugin() {
  doLast { println "added by a script plugin!" }
}
```

`build.gradle`:

```
apply from: "myPlugin.gradle"
```

Relative file paths are resolved relative to the applying project.

Remote Script Plugins

Script plugins can be sourced over HTTP.

```
apply from: "http://my.org/gradle-scripts/awesome-features-1.0.gradle"
```

- Supports a "push" model of reuse
- Updates are available to all consumers instantly
- Up to you to version and control

In Gradle 4.1 and older, these scripts are not cached. If the URL is not accessible, your build will fail.

Binary Plugins

Binary plugins are implementations of the `Plugin` interface.

```
package org.foo.plugins

class MyPlugin implements Plugin<Project> {
  void apply(Project project) {
    Task myTask = project.tasks.create("myTask")
    myTask.doLast {
      println "added by a binary plugin!"
    }
  }
}
```

Typically compiled and reused via JARs. (Adding plugin JARs to the classpath will be covered later.)

Applying Binary Plugins

Apply via their class instance…

```
apply plugin: org.foo.plugins.MyPlugin
```

Or via their *plugin ID*:

```
apply plugin: "org.foo.my-plugin"
```

It's harmless to apply the same plugin multiple times (i.e. application is idempotent).

Declaring Plugin IDs

Plugin types are mapped to IDs by searching the classpath for a conventional properties file.

build script:

```
apply plugin: "org.foo.my-plugin"
```

META-INF/gradle-plugins/org.foo.my-plugin.properties:

```
implementation-class=org.foo.plugins.MyPlugin
```

Name is: META-INF/gradle-plugins/«plugin id».properties

Standard Gradle Plugins

Gradle ships with many useful plugins.

Some examples:

- `java` - compile, test, package, upload Java projects
- `checkstyle` - static analysis for Java code
- `maven` - uploading artifacts to Apache Maven repositories
- `scala` - compile, test, package, upload Scala projects
- `idea` and `eclipse` - generates metadata so IDEs understand the project
- `application` - support packaging your Java code as a runnable application

Many more, listed in the Gradle User Guide.

Plugin Composition

Plugins can build upon other plugins. This is a common pattern.

- A `base` plugin provides generic *capabilities*
- Another plugin builds on the base, adding opinionated *conventions*

Example:

- `java-base` plugin adds the "source set" capability
- `java` plugin adds a `main` and `test` source set (and other defaults)

Allows users to back out of conventions if they don't suit.

Plugins Applying Plugins

```
class JavaPlugin implements Plugin<Project> {
  void apply(Project project) {
    project.apply(plugin: "java-base")
    project.sourceSets {
      main {
        …
      }
    }
  }
}
```

Safe because applying plugins is idempotent.

Lab

07-applying-plugins

Gradle File Types

Copy Specs

Abstract, composable, specification of content to be copied (not destination).

```
def baseSpec = copySpec {
  from "source"
  include "**/*.java"
}

task copy(type: Copy) {
  from "someFile.txt"
  into "target"
  with baseSpec
}

task copy2(type: Copy) {
  from "someFile2.txt"
  into "target2"
  with baseSpec
}
```

Custom Gradle File Types

Specialized types for dealing with collections of files.

- FileCollection: flattened set of files (e.g. classpath)
- FileTree (extends FileCollection): hierarchy of files (e.g. directory)

`Project` methods and their return types:

- `files()` -> `FileCollection`
- `fileTree()` -> `FileTree`
- `zipTree()` -> `ZipFileTree`
- `tarTree()` -> `TarFileTree`

Custom Gradle File Types

Used extensively through the Gradle API.

Key features:

- Path representations
- Ant integration (i.e. convert to Ant types)
- Relative paths are resolved against the project root
- Additive (you can add/subtract them)
- Lazily evaluated
- `Buildable` (more on this later)

FileCollection Examples

```
def f = files("my.txt", new File("/rootFile"), ["hello.txt"])

f.asPath

def txtFiles = f.filter { file ->
  file.name.endsWith(".txt")
}

def allTextFiles = txtFiles + files("new.txt")

f.from "other.txt"
assert allTextFiles.contains(file("other.txt"))

def noTextFiles = f - txtFiles

allTextFiles.each { file -> /* do something */ }
allTextFiles.files // returns a `Set` of files
```

FileTree Examples

`FileTree` extends `FileCollection`.

```
def tree = fileTree("someDir")

def jpgTree = fileTree("dir").matching {
  include "**/*.jpg"
}

def liveFilter = tree.matching {
  include "**/*.txt"
}

tree.exclude "**/new.*"
assert ! liveFilter.contains(file("someDir/new.txt"))

tree.visit { details ->
  // do something
}

tree.files // flattens the tree
```

Lab

08-gradle-file-types

Misc File Stuff

Delete Task

```
// Delete task
task myDelete(type: Delete) {
  delete "someFile", "someDir"
  delete file("otherDir")
  doFirst {
    println "Will delete: $targetFiles"
  }
}

// delete method
task myFileTask {
  doLast {
    delete "someFile", "someDir"
  }
}
```

Copy Method

You can copy files imperatively, using `Project.copy()`.

```
// Copy task
task myCopy(type: Copy) {
    from "somewhere"
    into "somewhere-else"
}

// copy method
task myTask << {
  copy {
    from "somewhere"
    into "somewhere-else"
  }
}
```

- Same API as `Copy` task
- Designed to be used by custom task implementations
- No up-to-date check
- Prefer `Copy` task whenever possible

Mkdir Method

`Project.mkdir()`.

```
task someTask {
  doLast {
    new File(mkdir("some/dir"), "foo.txt").text = "bar"
  }
}
```

Provides useful error messages and resolves relative paths.

Some Missing Bits

No move task/method (use the Ant task or Java API).

No jar/zip/tar methods (use tasks).

Ant Integration

Ant

- Ant is Gradle's friend, not its competitor
- Gradle uses Ant tasks internally
- You can use any Ant task from Gradle
- Ant tasks are an integral part of Gradle
- Gradle ships with Ant
- You can import any Ant build into Gradle

Ant Tasks

Projects provide an enhanced version of Groovy's `AntBuilder`.

```
ant.delete dir: "someDir"
ant {
  ftp(server: "ftp.comp.org", userid: "me", ...) {
    fileset(dir: "htdocs/manual") {
      include name: "**/*.html"
    }
    // high end
    myFileTree.addToAntBuilder(ant, "fileset")
  }
  mkdir dir: "someDir"
}
```

- Executed immediately
- Almost always go into task action

Basic rules for conversion

- XML elements become method calls
- XML attributes become Map arguments
- XML element text becomes a String argument
- Child elements are declared inside a closure argument

Ant task example

Ant:

```
<ftp server="ftp.comp.org" userid="me">
    <fileset dir="htdocs/manual">
        <include name="**/*.html"/>
    </fileset>
</ftp>
<echo>Hello!</echo>
```

Gradle:

```
ant.ftp(server: "ftp.comp.org", userid: "me") {
    fileset(dir: "htdocs/manual") {
      include name: "**/*.html"
    }
}
ant.echo("Hello!")
```

Importing Ant Builds

build.xml:

```
<project>
  <target name="hello" depends="intro">
    <echo>Hello, from Ant</echo>
  </target>
</project>
```

build.gradle:

```
ant.importBuild "build.xml"
hello.doFirst { println "Here comes Ant" }
task intro {
  doLast { println "Hello, from Gradle" }
}

$ gradle hello
Hello, from Gradle
Here comes Ant
[ant:echo] Hello, from Ant
```

Lab

09-ant-integration

Dependency Management

Dependencies

- Types of dependencies:
  - Repository dependencies
    - e.g. from Maven Central
    - with module descriptors (pom/ivy file)
  - Repository-less dependencies (specified by path)
  - Project dependencies in a multi-project build
- Domain objects:
  - Repository
  - Dependency
  - Configuration
  - Artifact

Working with Dependencies

- Configuration is a FileCollection
- Has a rich API

```
configurations.runtime.each { file ->
  println file
}

configurations.runtime.dependencies.matching { dep ->
  dep.group == "org.gradle"
}.each {
  println it
}

task copy(type: Copy) {
  from configurations.runtime
  into "someFolder"
}
```

Transitive Dependencies

- Advantage of repository dependencies
- pom/ivy model describes the transitive dependencies
- Default version conflict resolution is *newest*
- Option to use *fail* conflict resolution
- Transitive resolution is customizable

```
dependencies {
  compile("org.hibernate:hibernate:3.1") {
    force = true
    exclude module: "cglib"
  }
  compile("org:somename:1.0") {
    transitive = false
  }
}
configurations.myconf {
  transitive = false
  resolutionStrategy.failOnVersionConflict()
}
```

Forcing versions

- Forcing versions makes it possible to override default conflict resolution mechanism
- Forcing versions can be used to avoid bad versions or to stick with specific version

```
dependencies {
  //forcing version of a direct dependency
  compile("org.hibernate:hibernate:3.1") {
    force = true
  }
}

//forcing version at the level of configuration
//affects direct and transitive dependencies
configurations.compile {
  resolutionStrategy.force "org.hibernate:hibernate:3.1"
}
```

Dependency Resolution Strategies

```
configurations.all {
  resolutionStrategy.eachDependency { DependencyResolveDetails details ->
    if (details.requested.group == "org.gradle") {
      details.useVersion "1.4"
    }
  }

  resolutionStrategy.eachDependency { details ->
    if (details.requested.group == "org.acme.software"
        && details.requested.name == "cool-library"
        && details.requested.version == "1.2") {
      //prefer different version which contains some fixes
      details.useVersion "1.2.1"
    }
  }
}
```

Dependency Reports

Viewing the dependency tree:

```
gradle dependencies [--configuration «name»]
```

Focus on a particular dependency:

```
gradle dependencyInsight --dependency «name» --configuration «name»
```

- Defaults to compile
- Shows versions and selection *reason*

Lab

10-transitive-dependencies

Uploading

- Upload your artifacts to any Maven/Ivy repository
- pom/ivy file is generated
- Repository metadata (e.g. maven-metadata.xml) is generated
- "base" plugin adds "archives" configuration and applies task rules for uploading configurations
- "java" plugin automatically adds jar to the "archives" configuration artifacts

Uploading to Ivy Repositories

```
task myJar(type: Jar)

artifacts {
  archives myJar
}

uploadArchives {
  repositories {
    ivy {
      url "http://repo.mycompany.com"
      credentials {
        username "john"
        password "secret"
      }
    }
  }
}
```

Lab

11-ivy-uploading

Uploading to Maven Repositories

```
apply plugin: "maven"

uploadArchives {
  repositories {
    mavenDeployer {
      repository(url: "http://my.org/m2repo/")
    }
  }
}
```

- Provided by the `maven` plugin
- You can use all wagon protocols for uploading

Install to Local Maven Repo

Installs into ~/.m2/repository (reads Maven's `settings.xml`).

```
apply plugin: "maven"
```

The `install` task is added by the maven plugin.

```
> gradle install
```

Can be useful for locally sharing development versions.

```
repositories {
  mavenLocal()
}
```

Customizing the POM

```
uploadArchives {
  repositories {
    mavenDeployer {
      repository(url: "http://my.org/m2repo/")
      pom.project {
        description "A test project"
        licenses {
          license {
            name "Apache License, Version 2.0"
            url "http://.../LICENSE-2.0.txt"
          }
        }
      }
    }
  }
}
```

`pom.project {}` gives full access to the Maven class model.

Lab

12-maven-uploading

Extending Gradle

Gradle Extensibility

Different to Groovy's extensibility.

- Add "extra properties" to objects
- Add "extension" objects to existing objects

Allows built in domain types to be extended, including `Project`.

Makes the *build language* extensible.

Global Properties

```
def myDocsDestDir = "$buildDir/myDocs"

task myDocs {
  doLast {
    copy {
      from "someDir"
      into myDocsDestDir
    }
  }
}

task zip(type: Zip) {
  from myDocsDestDir
}
```

Easy to lose relationship between producers and consumers.

Extra Properties

```
task myDocs {
  ext.destDir = "$buildDir/myDocs"
  doLast {
    copy {
      from "someDir"
      into destDir
    }
  }
}
task zip(type: Zip) {
  from myDocs.destDir
}
```

- Applicable to most Gradle types
- Good OO design (e.g. encapsulation)
- Custom task type is a (more heavyweight) alternative

Extra Methods

```
task bar {
  ext {
    serviceUrl = ...
    //adding a 'domainGroup' method:
    domainGroup = {
      getGroup(serviceUrl)
    }
  }
}
task foo {
  fooProp = bar.domainGroup()
}
```

Just extra properties, where the property value is a Groovy closure.

Extensions

New objects can be attached to existing ones.

```
class MyExtension {
  String someProperty
}

extensions.create("myDSL", MyExtension)

myDSL {
  someProperty = "someValue"
}
```

Most types are extensible.

See `ExtensionAware` and `ExtensionContainer`.

Domain Object Container

Used for many domain objects (plugins, configs, tasks, ...)

```
def allJars = tasks.withType(Jar) //built-in filter
task myJar(type: Jar) //filter is 'live'

//custom filter:
def webTasks = tasks.matching { task ->
  task.name.startsWith("web")
}

//filter chaining:
def compJars = tasks.withType(Jar).matching { task ->
   task.name.startsWith("compile")
}

//dynamic dependsOn
task buildAllJars { dependsOn allJars }
```

Configuration Rules

Apply configuration to matching items now and in the future.

```
tasks.all {
  doFirst {
    println 'rule for all tasks, including those not yet created'
  }
}
tasks.withType(Jar) {
  destinationDir = "somePath"
  doLast { /* do something */ }
}

tasks.whenTaskAdded { task -> ... }
```

Configuration Rules Example

```
tasks.withType(Jar) {
  ext.ftp = false // add extra property
}

task jar1(type: Jar)

task jar2(type: Jar) {
  ftp = true
}

ext.ftpJars = tasks.withType(Jar).matching { it.ftp }

task jar3(type: Jar) {
  ftp = true
}

task showFtpJars {
  doLast { ftpJars.each { println it.name } }
}
```

Lab

13-extending-gradle

Task Inputs /Outputs

Task Inputs/Outputs

- One of Gradle's killer features
- You can describe:
  - Input/output files
  - Input/output directories
  - Input properties
- Gradle’s built-in tasks all describe their inputs/outputs

Input/Output Annotations

```
class MyTask extends DefaultTask {
  @InputFile File text
  @InputFiles FileCollection path
  @InputDirectory File templates
  @Input String mode
  @OutputFile File result
  @OutputDirectory File transformedTemplates
  boolean verbose // ignored

  @TaskAction
  generate() { ... }
}
```

Input/Output API

```
ant.import "build.xml"
someAntTarget {
  inputs.files "template.tm", new File("data.txt")
  inputs.dir "someDir"
  outputs.files "output.txt"
  outputs.dir "generatedFilesDir"
  outputs.upToDateWhen { task ->
    dbDataUpToDate(task.dbUrl)
  }
}
```

Incremental Build

- A task is UP-TO-DATE if:
  - Inputs haven’t changed
  - Outputs still present (untampered)
- Change detection
  - Input/output files are hashed
  - Content of input/output dirs is hashed
  - Values of input properties are serialized

More details

- file hashes are kept in projectDir/.gradle
- --rerun-tasks command line option bypasses up-to-date checks
- running the build with -i (--info) reveals more insight into up-to-date calculation
- 'UP-TO-DATE' decoration in the terminal is also printed for skipped tasks (for example, tasks that have no actions)
    - use --info to understand up-to-date result

Property Processing

- Input files/dirs are verified to exist
    - Disable with @Optional
- Output dirs are created before execution

Inferred Task Dependencies

- FileCollection/FileTree can be buildable
- Buildable input files/dirs allows inferring the dependencies

```
task generatedByMe { doLast { /*write into mydir*/ } }
def myFiles = files("$buildDir/mydir") {
  builtBy generatedByMe // could be many tasks
}

task copy(type: Copy) {
  from myFiles // implicit dependsOn
  into "someDir"
}

compileJava {
  classpath = myFiles // implicit dependsOn
}
```

Custom Tasks...

```
task generatedByMe { ... }
def myFiles = files("$buildDir/mydir") {
  builtBy generatedByMe
}

task task1 {
  //will below infer the necessary dependency?
  doLast { println myFiles.files }
}

task task2 {
  dependsOn myFiles
  doLast { println myFiles.files }
}

task task3 {
  inputs.files myFiles // implicit dependsOn + incremental build
  doLast { println myFiles.files }
}
```

Lab

14-task-input-output

Java Plugin

Java Plugins

- java-base
  - Provides additional Task types
  - Defines rules for conventions
  - Adds declarative elements to the DSL (e.g. SourceSet)
- java
  - Adds task instances to the project
  - Adds default values to task instances
  - Adds source sets for production and test code
  - Configures the dependency management for Java projects (adds scopes for compile, runtime, ...)

Source Sets

Models a *logical* unit of source code.

- Source files (e.g .java files)
- Resource files (e.g. properties files)
- Output class files
- Compile & runtime classpaths
- Associated tasks (e.g. compile)

A declarative element.

Java plugin adds `main` and `test`.

Source Set Defaults

When using the `java-base` plugin, all source sets have the defaults:

- Source: `src/«name»/«language»`
- Resources: `src/«name»/resources`
- Classes: `$buildDir/classes/«name»`
- Compile task: `compile«name»«language»`
  - e.g. `compileTestJava`
- Resource task: `process«name»Resources`
  - e.g. `processTestResources`
- Compile dependencies configuration: `«name»Compile` configuration
  - e.g. `testCompile`
- Runtime dependencies configuration: `«name»Runtime` configuration
  - e.g. `testRuntime`

“main” Source Set

Derived names for the “main” source set are different.

- **`compileJava`** -- not `compileMainJava`
- **`processResources`** -- not `processMainResources`
- **`compile`** -- not `mainCompile`

A common pattern in Gradle plugins.

Source Set Output

Source Sets have an `output` property, a buildable FileCollection for the built source.

- Class files
- Processed resources

```
task jar {
  from sourceSets.main.output
}
```

Used extensively by the `java` plugin to wire tasks together.

Working with Source Sets

```
sourceSets {
  main {
    java.srcDirs = ["src"] //overwrite dirs
    resources {
      srcDirs = ["src"]
    }
  }
  integTest {
    java.srcDirs "src/integTest" //add dirs
    output.classesDir = file("$buildDir/integ-classes")

    //FileCollections can be added together
    compileClasspath = sourceSets.main.output
        + configurations.integTestCompile
    runtimeClasspath = compileClasspath + output
  }
}
```

Querying Source Sets

```
// They all return FileTree
sourceSets.main.allJava
sourceSets.main.resources
sourceSets.main.allSource.matching { include ... }

// Returns a buildable FileCollection
sourceSets.main.output
```

Clean Task

- By default clean deletes the buildDir
- You can specify additional files to delete
- name: 'clean', type: Delete

```
clean {
  delete "fooDir", "bar.txt",
  fileTree("texts").matching { ... }
}
```

Javadoc Task

- Provides all the options of the Javadoc command
- name: 'javadoc', type: Javadoc
- input: sourceSets.main.java, sourceSets.main.compileClasspath

```
javadoc {
  maxMemory = "512M"
  include "org/gradle/api/**"
  title = "Gradle API $version"
}
```

Resources Tasks

- Usually configured via the source set
- Can use the powerful Copy API
- name: processResources, processTestResources
- type: Copy
- input: sourceSets.main(test).resources

Compile Tasks

- Usually configured via the source set
- Provides all the options of the Ant javac task
- name: compile, testCompile, type: JavaCompile
- input: sourceSets.main(test).java , sourceSets.main(test).compileClasspath

```
compileJava {
  options.fork = true
  options.forkOptions.with {
    memoryMaximumSize = "512M"
  }
}
```

Classes Tasks

- Aggregates compile related tasks
- name: classes, testClasses, type: DefaultTask
- dependsOn: compile|testCompile, processResources|processTestResources

Jar Task

- Content of the Jar: production classes
- name: 'jar', type: Jar
- input: sourceSets.main.output

```
jar {
   //you can add more content:
   from sourceSets.main.allJava
   from zipTree("lib/someJar.jar")
}
```

Test Task

- Support for JUnit and TestNG
- Parallel testing
- Custom fork frequency
- Test listeners
- Tests auto-detected in sourceSets.test.output
- name: 'test', type: Test
- input: sourceSets.test.output, sourceSets.test.runtimeClasspath

Test Task Example

```
test {
  jvmArgs "-Xmx512M"
  scanForTestClasses = false //disables auto-detection
  include "**/tests/special/**/*Test.class"
  exclude "**/Old*Test.class"
  forkEvery = 30
  maxParallelForks = guessMaxForks()
}

def guessMaxForks() {
  int processors = Runtime.runtime.availableProcessors()
  Math.max(2, processors.intdiv(2))
}
```

Test Task Listeners

```
test {
  beforeTest { desc ->
    // do something
  }
  afterTest { desc, result ->
    // do something
  }
  afterSuite { desc, result ->
    // do something
  }
}
```

Lab

15-testing

Build Lifecycle

Lifecycle Tasks

- Most important tasks to build users
- Concept and function, not a specific task *type*

Standard Java lifecycle tasks:

- **clean**
- **classes** - compile main source and resources
- **test** - unit tests
- **assemble** - make all “outputs”
- **check** - run *all* checks
- **build** - assemble & check

Lifecycle Tasks

Lifecycle tasks often have no actions, only dependencies…

```
task check {
  dependsOn test, codeQuality
}
```

The standard lifecycle can be easily extended just by adding new tasks and dependencies.

Lifecycle Tasks & Plugins

Plugins can integrate by adding dependencies to lifecycle tasks…

```
class IntegTestPlugin implements Plugin<Project> {
  void apply(Project project) {
    project.apply plugin: "java"
    // create integ test task
    project.check.dependsOn integTest
  }
}
```

Convention plugins should always consider the larger build lifecycle.

Multi-Project Builds

Multi-Project Builds

- Flexible directory layout
- Configuration injection
- Project dependencies
- Partial builds
- Customize build file names

Configuration Injection

```
subprojects {
  apply plugin: "java"
  dependencies {
    testCompile "junit:junit:4.7"
  }
  test {
    jvmArgs "-Xmx512M"
  }
}
```

Filtered configuration

```
configure(nonWebProjects()) {
  jar.manifest.attributes Implementor: "Gradle Inc."
}

def nonWebProjects() {
  subprojects.findAll { !it.name.startsWith("web") }
}
```

Task/Project Paths

- For projects and tasks there is a fully qualified path notation:
  - : (root project)
  - :clean (the clean task of the root project)
  - :foo (the foo project)
  - :foo:clean (the clean task of foo)

```
$ gradle :foo:classes
```

Configuring a Multi-Project Build

- settings.gradle (location defines the root)
- Most aspects of the multi-project build are customizable

```
include "foo", "bar"

//default: root dir name
rootProject.name = "main"

//default: 'api' dir
project(":foo").projectDir = file("/myLocation")

//default: 'build.gradle'
project(":bar").buildFileName = "bar.gradle"
```

Lab

16-multi-project-builds

Organizing Build Logic

Best Practices

- Use script plugins to decompose build scripts
  - Enhances comprehension and allows for reuse
  - Modularize according to domain (integ tests) or role (user/build admin)
- Encapsulate the imperative into plugins and custom tasks
- Enhance the API of the Gradle domain objects:
  - Encapsulate custom behavior
  - Integrate them with your own custom elements
    - `compile('junit:junit:4.10') { maven.optional = true }`
- Add your own declarative elements

Organizing Build Logic

- Your build logic can live in different locations:
  - In the build script
  - In another local/remote script (script plugin)
  - In the 'buildSrc' project
  - In some Jar
  - init.gradle

Build Script

- Build logic in the build script is fine
- Always try to separate the imperative from the declarative

```
//declarative:
task greeting(type: HelloTask) {
  greeting = "greetings from HelloTask"
}

//implementation details:
class HelloTask extends DefaultTask {
  String greeting = "hello from HelloTask"
  @TaskAction
  void printGreeting() {
    println greeting
    // do something complicated
  }
}
```

Script Plugin

- build.gradle
- gradle/
    - distributions.gradle
    - integTest.gradle

```
//root build.gradle:
apply from: "gradle/integTest.gradle"
```

buildSrc

- Drop any Java/Groovy class into:
  - /buildSrc/src/main/java(groovy)
- Gradle will automatically compile and test with any invocation
- Part of the build script class path

Jar

- Jars can be added to the build script class path
  - Your own or third-party plugins
  - Any other libraries (e.g. commons-math)

```
buildscript {
  repositories {
    mavenCentral()
  }
  dependencies {
    classpath "com.google.appengine:gradle-appengine-plugin:1.8.6"
    classpath files("lib/commons-math.jar")
  }
}
```

Lab

17-organizing-build-logic

Hooking into Gradle

Init Scripts

- Init scripts are run before the build starts:
    - Configure build script classpath (e.g. pull in your corporate plugins).
    - Configure common corporate repositories
    - Set up properties based on the current environment.
    - Define machine specific details, such as where JDKs are installed.
    - Register build listeners.
    - ...
- Also useful to enhance builds you don't want to touch.
- GRADLE_USER_HOME/init.gradle is automatically applied as an init script.
- You can specify any init script via the -I command line option.

```
$ gradle assemble -I ci-init.gradle
```

Sample Init Script

```
initscript {
  repositories {
    mavenCentral()
  }
  dependencies {
    classpath "org.apache.commons:commons-math:2.0"
  }
}
gradle.startParameter // do something with them
gradle.addBuildListener ...
```

Hooking into the Lifecycle

```
gradle.taskGraph.whenReady { taskGraph -> ... }
gradle.taskGraph.beforeTask { task -> ... }
gradle.taskGraph.afterTask { task -> ... }
gradle.beforeProject { project -> ... }
gradle.afterProject { project -> ... }
gradle.addBuildListener(BuildListener listener)

public interface BuildListener {
    void buildStarted(Gradle gradle);
    void settingsEvaluated(Settings settings);
    void projectsLoaded(Gradle gradle);
    void projectsEvaluated(Gradle gradle);
    void buildFinished(BuildResult result);
}
```

BuildAdapter

The BuildAdapter class provides a stubbed implementation of BuildListener.

```
public class MyListener extends BuildAdapter {
    void settingsEvaluated(Settings settings) {
        // do something interesting with the Settings object
    }
}
```

Lab

18-hooking-into-gradle

The Gradle Way

Declarativeness

- Build scripts specify **what** should happen.
- Gradle & plugins figure out the **how**.

Declarative where possible, imperative where necessary.

Flexibility

Gradle is not inherently prescriptive.

Flexibility is needed to meet the challenges of modern software delivery.

Not all projects are the same, and most real world projects are significantly non trivial.

Gradle provides mechanisms for managing complexity.

Gradle provides mechanisms for domain specific conventions and abstractions.

Build Language

Gradle is a build language engine, supporting domain modelling.

- Projects
- Custom Tasks
- Plugins
- Dependencies
- Configurations
- Source Sets
- Archives
- Artifacts

New Domains

```
android {
    defaultConfig {
        minSdkVersion 8
        versionCode 10
    }
    productFlavors {
        flavor1 {
            packageName "com.example.flavor1"
            versionCode 20
        }
        flavor2 {
            packageName "com.example.flavor2"
            minSdkVersion 14
        }
    }
}
```

The build language can be easily extended to describe any domain.

Ambitious Automation

Gradle supports ambitious, high quality automation.

High quality automation improves developer productivity and software quality.

Improved software quality makes developers, users, *everyone* happier.

Thank You!

- Thank you for attending!
- Questions?
- Feedback?
- Gradle Home
- Get more help!