Agile Software Development Practices

SOF2412 / COMP9412

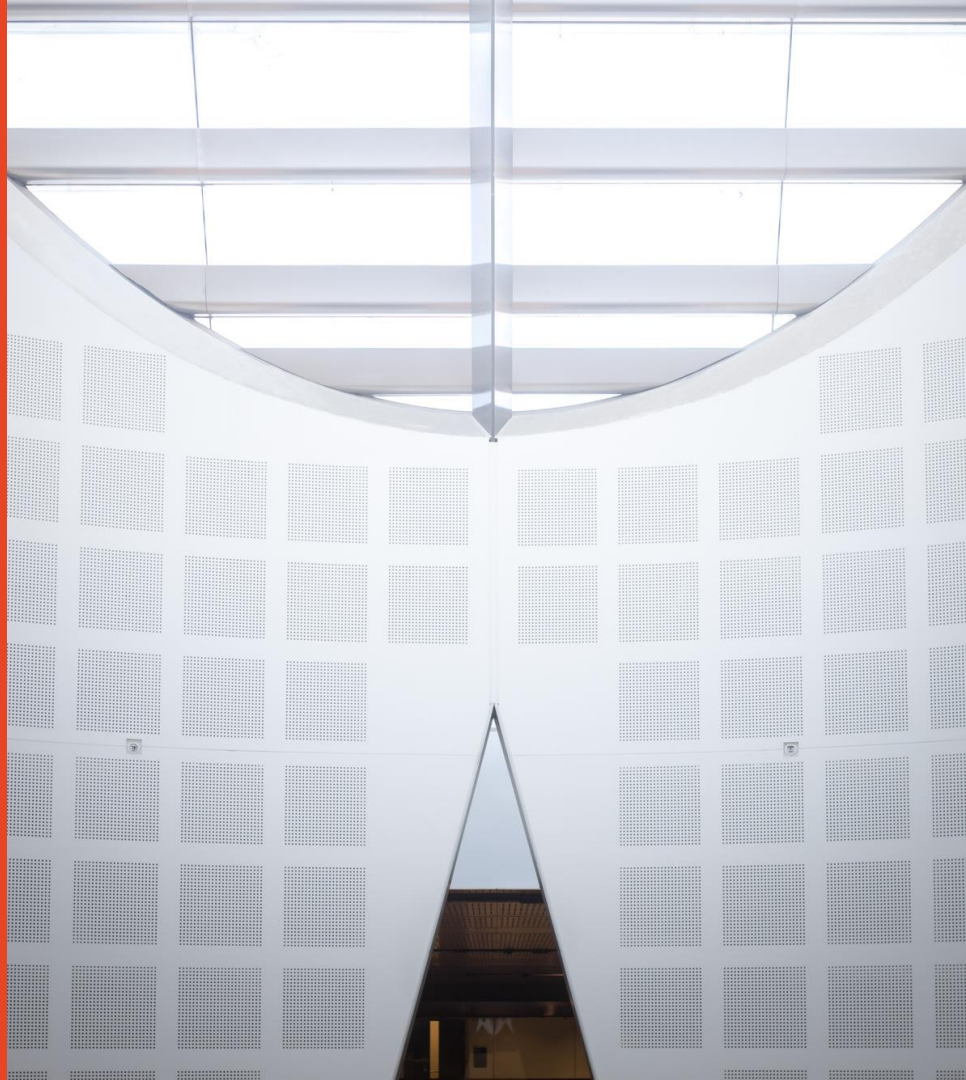
System Build Automation

Dr. Basem Suleiman

Presented by A/Prof. Bernhard Scholz

School of Information Technologies

THE UNIVERSITY OF SYDNEY



Copyright warning

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

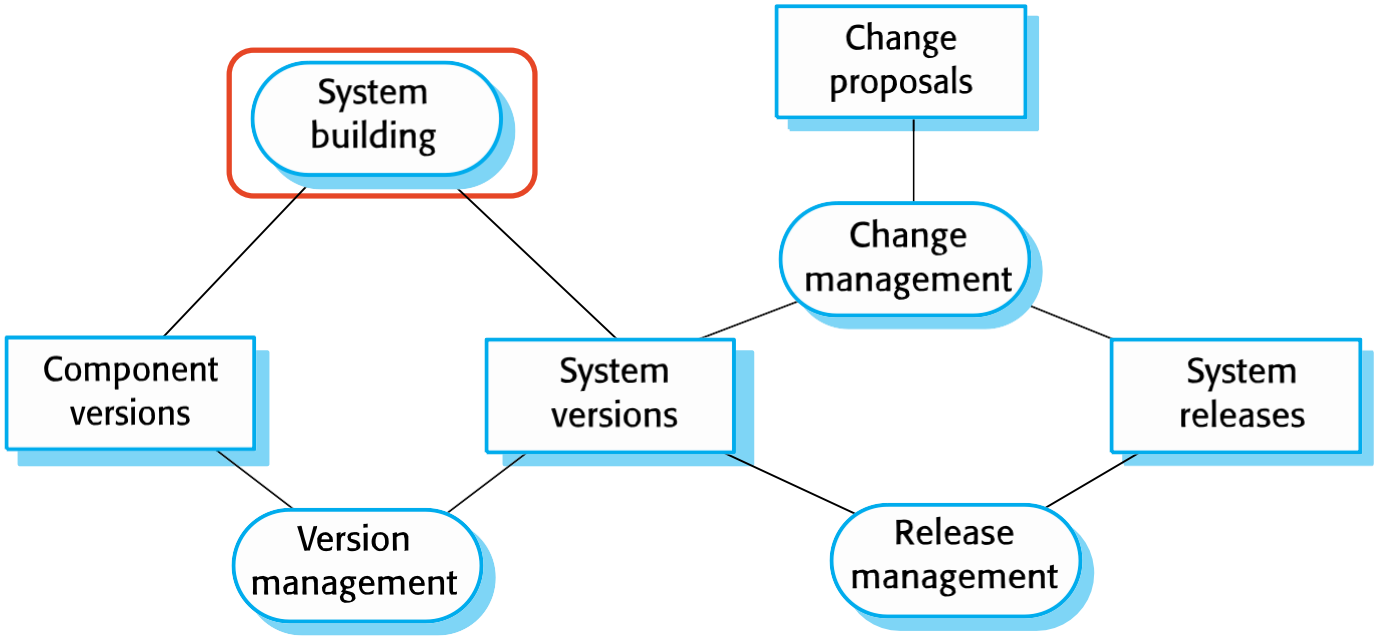
Agenda

- Software Configuration Management
  - System Building
  - Agile System Build
- Software Build Automation Tools
  - Ant
  - Maven
  - Gradle

Configuration Management (CM)

- Software systems are constantly changing during development and use
- Configuration management (CM) is concerned with the policies, processes and tools for managing changing software systems
- You need CM because it is easy to lose track of what changes and component versions have been incorporated into each system version.
- CM is essential for team projects to control changes made by different developers

Configuration Management Activities



Configuration Management Activities

- **System building**: the process of assembling program components, data and libraries, then compiling these to create an executable system.
- **Version management**: Keeping track of the multiple versions of system components and ensuring that changes made to components by different developers do not interfere with each other.
- **Change management**: keeping track of requests for changes to the software from customers and developers, working out the costs and impact of changes, and deciding the changes should be implemented.
- **Release management**: preparing software for external release and keeping track of the system versions that have been released for customer use.

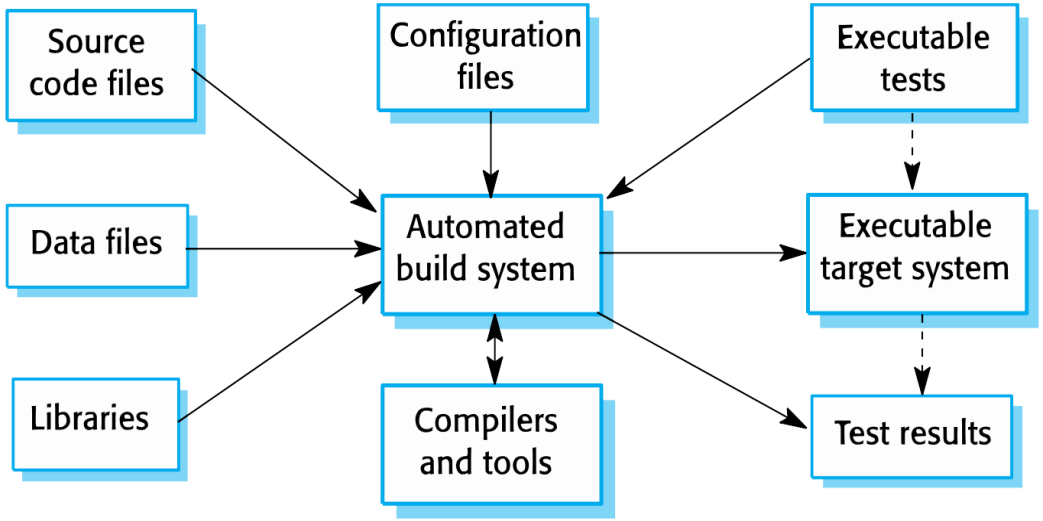
Agile Development in CM

- Agile development, where components and systems are changed several times per day, is impossible without using CM tools.

- The definitive versions of components are held in a shared project repository and developers copy these into their own workspace.

- They make changes to the code then use system building tools to create a new system on their own computer for testing. Once they are happy with the hanges made, they return the modified components to the project repository.

System Building



System Building

- System building is the process of creating a complete, executable system by compiling and linking the system components, external libraries, configuration files, etc.

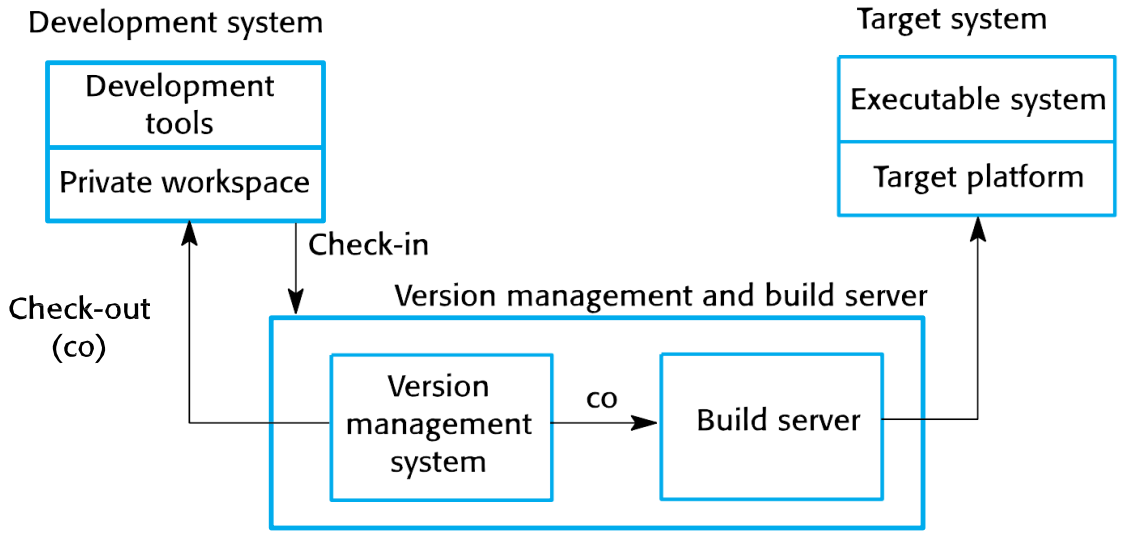
- System building tools and version management tools must communicate as the build process involves checking out component versions from the repo managed by the version management system.

- The configuration description used to identify a baseline is also used by the system building tool

System Integration and Building Tools

- Build script generation
- Version management system integration
- Minimal re-compilation
- Executable system creation
- Test automation
- Reporting
- Documentation generation

System Integration and Building Tools



System Building

- The development system, which includes development tools such as compilers, source code editors, etc.
  - Developers check out code from the version management system into a private workspace (repo.) before making changes to the system.
- The build server, used to build definitive, executable versions of the system
  - Developers check-in code to the version management system before it is built. The system build may rely on external libraries that are not in the version management system.
- The target environment, which is the platform on which the system executes.

Agile System Build

- Check out the mainline system from the version management system into the developer's private workspace

- Build the system and run automated tests to ensure that the built system passes all tests. If not, the build is broken and you should inform whoever checked in the last baseline system. They are responsible for repairing the problem.

- Make the changes to the system components

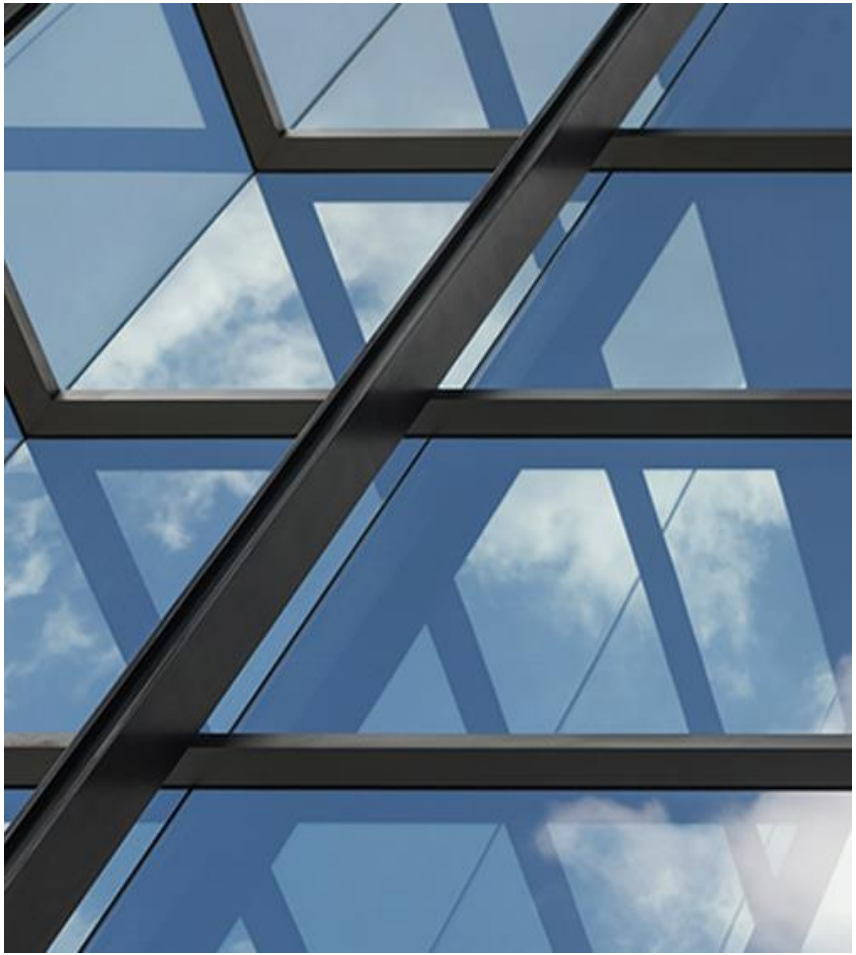
- Build the system in the private workspace and rerun system tests. If the tests fail, continue editing.

Agile System Build

- Once the system has passed its tests, check it into the build system but do not commit it as a new system baseline.
- Build the system on the build server and run the tests.
- You need to do this in case others have modified components since you checked out the system. If this is the case, check out the components that have failed and edit these so that tests pass on your private workspace.
- If the system passes its tests on the build system, then commit the changes you have made as a new baseline in the system mainline.

Software Build Automation Tools

Ant, Maven, Gradle



THE UNIVERSITY OF SYDNEY

Build Tools – Apache Ant

- Apache Ant is a Java-based software build tool for automating build processes
  - Requires Java platform and best suited for building Java projects
- It offers extreme flexibility to the user
  - Does not impose coding conventions
  - Does not impose any heavyweight dependency management framework (no special directory layout required to the java project)
- It uses XML to describe the code build process and its dependencies
  - By default *build.xml*

https://en.wikipedia.org/wiki/Apache_Ant

Apache ANT – Example

```xml
<?xml version="1.0"?>
<project name="Hello" default="compile">
    <target name="clean" description="remove intermediate files">
        <delete dir="classes"/>
    </target>
    <target name="clobber" depends="clean" description="remove all artifact files">
        <delete file="hello.jar"/>
    </target>
    <target name="compile" description="compile the Java source code to class files">
        <mkdir dir="classes"/>
        <javac srcdir="." destdir="classes"/>
    </target>
    <target name="jar" depends="compile" description="create a Jar file for the application">
        <jar destfile="hello.jar">
            <fileset dir="classes" includes="**/*.class"/>
            <manifest>
                <attribute name="Main-Class" value="HelloProgram"/>
            </manifest>
        </jar>
    </target>
</project>
```

https://en.wikipedia.org/wiki/Apache_Ant

Apache ANT – Drawbacks

- Too flexible
- Complexity (XML-based build files)
  - Need to specify a lot of things to make simple builds
- No standard structure/layout
  - Developers can create their own structure/layout of the project

Apache Maven

- A build automation tool used primarily for java projects
  - Also can be use d to build and manage software in C#, Ruby, Scala
  - XML-based description of the software being built, its **dependencies on other external modules and components**, the build order, directories, and required plug-ins
  - Popular IDEs support development with Maven; Eclipse, IntelliJ, JBuilder, NetBeans
- It uses conventions over configuration for the build procedure

- Built using a plugin-based architecture
  - Plugin for the .NET framework and native plugins for C/C++ are maintained

https://en.wikipedia.org/wiki/Apache_Maven
https://maven.apache.org/

Apache Maven – Minimal Example

- Maven projects are configured using Project Object Model (POM) stored in a pom.xml file

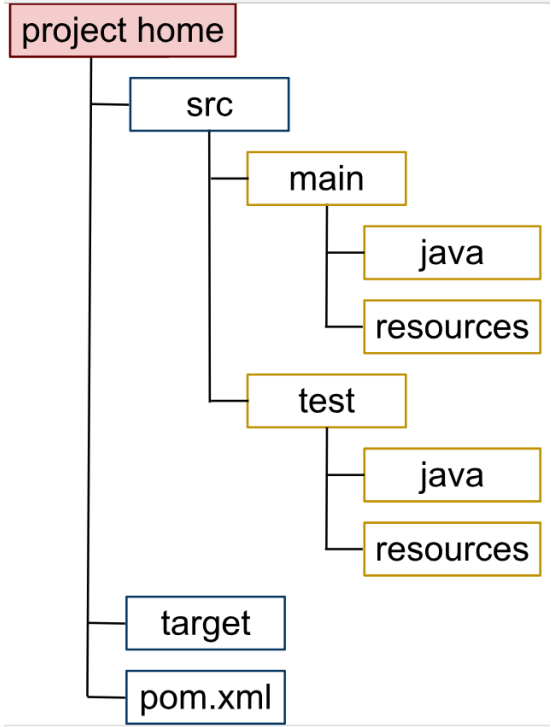
```xml
<project>
  <!-- model version is always 4.0.0 for Maven 2.x POMs -->
  <modelVersion>4.0.0</modelVersion>

  <!-- project coordinates, i.e. a group of values which
       uniquely identify this project -->

  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0</version>

  <!-- library dependencies -->

  <dependencies>
    <dependency>

      <!-- coordinates of the required library -->

      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>

      <!-- this dependency is only used for running and compiling tests -->

      <scope>test</scope>

    </dependency>
  </dependencies>
</project>
```

https://en.wikipedia.org/wiki/Apache_Maven

Apache Maven – Project Structure

- The directory structure of a normal Maven project has the directory entries shown in the figure at the right

- The command *mvn package* will
  - will compile all the Java files, run any tests, and package the deliverable code and resources into target/my-app-1.0.jar (assuming the artifactId is my-app and the version is 1.0.)



The Maven software tool auto-generated this directory structure for a Java project

https://commons.wikimedia.org/wiki/File:Maven_CoC.svg#/media/File:Maven_CoC.svg

Apache Maven – Central Repository

- Maven uses default Central Repository that maintains required software artefacts (libraries, plug-ins) to manage dependencies
- For example, project that is dependent on the Hibernate library needs to specify that in the pom.xml project file
  - Maven will check if the referenced dependency is already in the user's local repository
  - If it is, it will reference the dependency from the local repository
  - If it is not, Maven will dynamically download the dependency and the dependencies that Hibernate itself needs and store them in the user's local repository
- You can configure repositories other than the default (e.g., company-private repository)

Apache Maven – Drawbacks

- Again, XML-based files increase complexity (verbose)
- Too rigid; developers are required to understand follow the conventions

Gradle

- Build automation tool that builds upon the concepts of Apache Ant and Maven
  - Offers project build conventions, and allow developers to redefine those conventions
  - Project description is configured using Groovy-based Domain Specific Language (DSL)
  - The order in which tasks can run is determined using a directed acyclic graph (DAC)
  - Designed for multi-project builds
  - Incremental builds; it determines which parts of the build tree are up to date (any task dependent only on those parts does not need to be re-executed)
  - Flexible in the way it handles dependencies (transitive dependency management)

https://en.wikipedia.org/wiki/Gradle

Gradle – Groovy

- Gradle build files are Groovy scripts

- Groovy is a dynamic language of the JVM
  - Can be added as a plug-in
  - It allows developers to write general programming tasks in the build files
  - Relief developers from the lacking control flow in Ant or being forced into plug-in development in Maven to declare nonstandard tasks

https://en.wikipedia.org/wiki/Gradle

Gradle – DSL

- Gradle also presents a Domain-specific Language (DSL) tailored to the task of building code
  - Not general-purpose or programming language
  - Gradle DSL contains the language needed to describe how to build Java code and create a WAR file from the output
- Gradle DSL is extensible through plug-ins

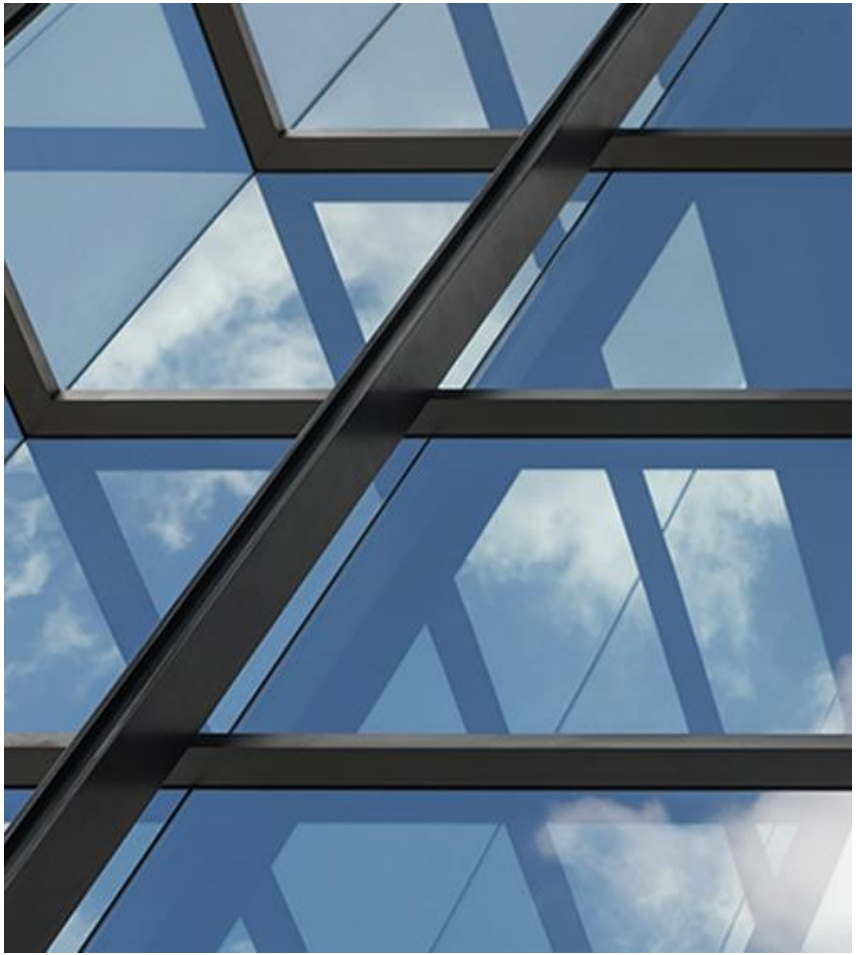
https://en.wikipedia.org/wiki/Gradle

Gradle – Extensible DSL

- Gradle DSL is extensible through plug-ins (if it doesn't have the language to describe what you want your build to do)
  - E.g., describe how to run database migration scripts or deploy code to a set of cloud-based QA servers

- Gradle plug-ins allow:
  - Adding new task definitions
  - Change the behavior of existing tasks
  - Add new objects
  - Create keywords to describe tasks that depart from the standard Gradle categories

Gradle Basics



THE UNIVERSITY OF SYDNEY

Gradle – Tasks

- **Task:** a single atomic piece of work for a build, such as compiling classes or generating Java documentation
- **Project:** a composition of several tasks that may represent the creation of a jar file, or a deploy of an application on the server
- Each task has a **name**, which can be used to refer to the task within its owning project, and a **fully qualified path**, which is unique across all tasks in all projects

https://en.wikipedia.org/wiki/Grale

Gradle – Task Actions

- A task is made up of sequence of **Action objects**
  - When a task is executed, each of the actions is executed (by calling *the Action.execute(T)*)
- Actions can be added to a task
  - *Task.doFirst()* or *Task.doLast()*
- Two Task action exceptions
  - Abort execution of the action and continue to the next actions of the task by throwing a ***StopActionException***
  - Abort execution of the task and continue to the next task by throwing a ***StopExecutionException***
  - Use these exceptions to have precondition actions which skip execution of the task, or part of the task, if not true

https://en.wikipedia.org/wiki/Grale

Gradle – Simplest Build File Example

Build.gradle

```
task helloWorld << {
        println 'hello, world'
}
```

Results of Hello World build file

$ gradle -q helloWorld
hello, world

One simple task, no dependencies

Build.gradle

```
task hello << {
print 'hello, '
}
task world(dependsOn: hello) << {
println 'world'
}
```

execute the second task, **world**

$ gradle -q world
hello, world

Two tasks with dependency

Gradle – Simplest Build File for Java Example (1)

Gradle.java

```java

package org.gradle.example.simple;

public class HelloWorld {
    public static void main(String args[]) {
        System.out.println("hello, world");
    }
}
```

build.gradle

```
apply plugin: 'java'
```

```
.
├── build.gradle
└── src
    └── main
        └── java
            └── org
                └── gradle
                    └── example
                        └── simple
                            └── HelloWorld.java
```

Project layout of HelloWorld.java

Simplest possible Gradle file for java

Gradle – Simplest Build File for Java Example (2)

This build file automatically introduces a number of tasks for us to run. Just run gradle build, and you'll see the output

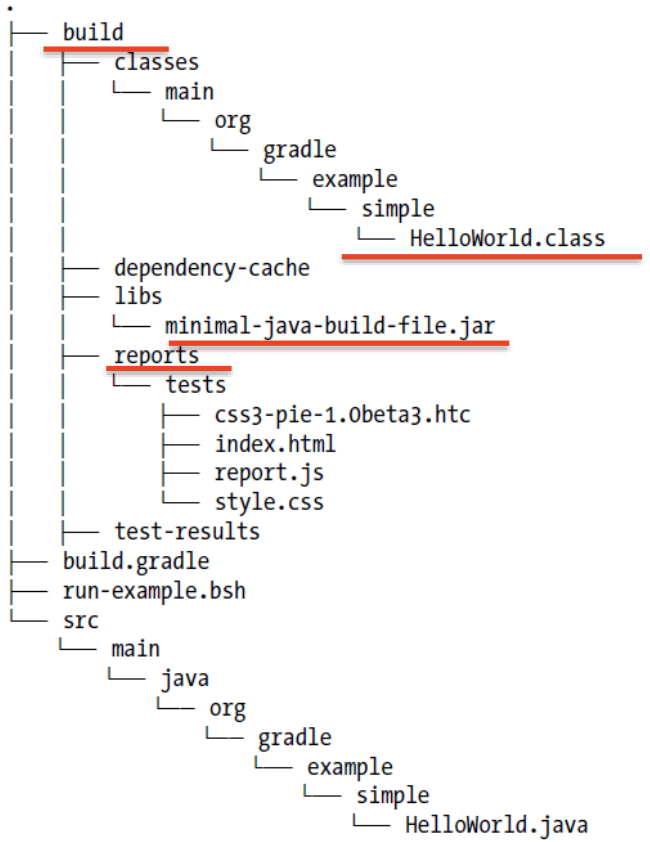
Notice:

- Class files generated and place in a directory
- Test report files (for unit test results)
- JAR built using the project directory

Run HelloWorld Java

$ java -cp build/classes/main/ org.gradle.example.simple.HelloWorld

hello, world



Project Layout of Hello World Java After Build

Gradle – Simplest Build File for Java Example (2)

This build file automatically introduces a number of tasks for us to run. Just run gradle build, and you'll see the output

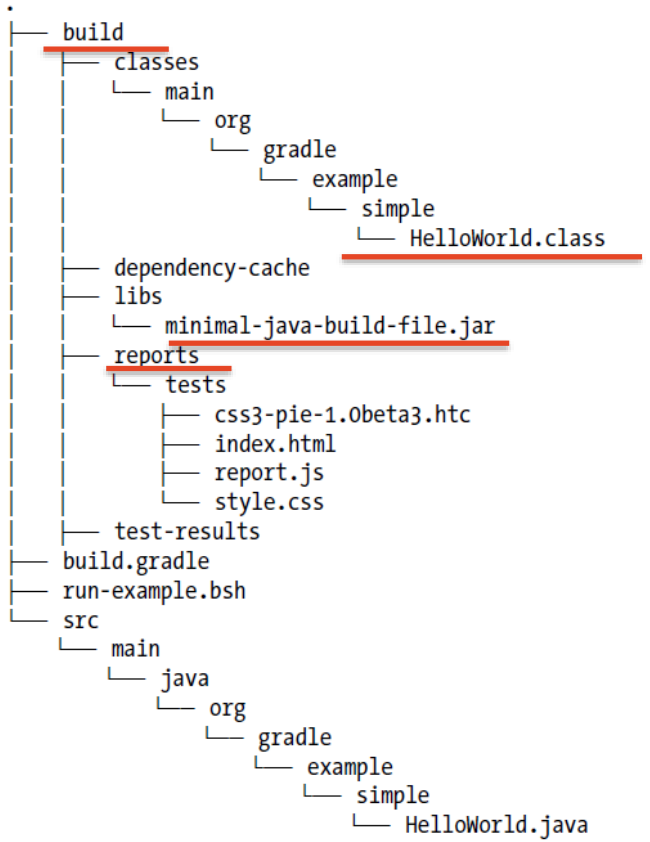
Notice:

- Class files generated and place in a directory
- Test report files (for unit test results)
- JAR built using the project directory

Run HelloWorld Java

$ java -cp build/classes/main/ org.gradle.example.simple.HelloWorld

hello, world



Project Layout of Hello World Java After Build

Gradle – Build Lifecycle

- Executing a build file in gradle will go through three phases:
- **Initialization:** gradle decides which projects are to participate in the build
- **Configuration:** task objects are assembled into an internal object model called Directed Acyclic Graph (DAG)
- **Execution:** build tasks are executed in the order required by their dependency relationship – this is important in multiproject build

Gradle – Task Configuration

- **Configuration block**: a place to setup variables and data structures needed by the task action when (and if) it runs later on the build
  - Provide distinction between configuration and action and turns build's tasks into a rich object model populated with information about the build

- **Closure:** used in Groovy to refer to a block of code between curly braces
  - Holding blocks of configuration and build actions
- The configuration block of a task is run during Gradle's *configuration* lifecycle phase, which runs before the *execution* phase, when task actions are executed.

build.gradle

```
1 ......
2 task initializeDatabase
3 initializeDatabase << { println 'connect to database' }
4 initializeDatabase << { println 'update database schema' }
5 initializeDatabase { println 'configuring database connection' }
```

Task

Task actions

Configuration block

Gradle – Tasks are Objects

- Every task declared in a project is represented internally as an **object**
  - Task's methods and properties
  - Gradle creates an internal object model of the build before executing it
  - Every declared task is of *DefaultTask* type. You can change the default task type
  - *DefaultTask* contains functionality required for them to interface with Gradle project model

Gradle – Methods of Default Task

Method	Description
dependsOn(task)	Adds a task as a dependency of the calling task. A depended-on task will always run before the task that depends on it
doFirst(closure)	Adds a block of executable code to the beginning of a task's action. During the execution phase, the action block of every relevant task is executed.
doLast(closure)	Appends behavior to the end of an action
onlyIf(closure)	Expresses a predicate which determines whether a task should be executed. The value of the predicate is the value returned by the closure.

Gradle – dependsOn() Examples

```
// Declare that world depends on hello
// Preserves any previously defined dependencies as well
task loadTestData {
  dependsOn createSchema
}

// An alternate way to express the same dependency
task loadTestData {
  dependsOn << createSchema
}

// Do the same using single quotes (which are usually optional)
task loadTestData {
  dependsOn 'createSchema'
}

// Explicitly call the method on the task object
task loadTestData
loadTestData.dependsOn createSchema

// A shortcut for declaring dependencies
task loadTestData(dependsOn: createSchema)
```

Different ways to call the dependsOn method

```
// Declare dependencies one at a time
task loadTestData {
  dependsOn << compileTestClasses
  dependsOn << createSchema
}

// Pass dependencies as a variable-length list
task world {
  dependsOn compileTestClasses, createSchema
}
```

A task can depend on more than one task – loadTestData depends on tasks createSchema and compileTestClasses

Gradle – doFirst() Examples

```
task setupDatabaseTests << {
  // This is the task's existing action
  println 'load test data'
}

setupDatabaseTests.doFirst {
  println 'create schema'
}
```

OR

```
task setupDatabaseTests << {
  println 'load test data'
}

setupDatabaseTests {
  doFirst {
    println 'create schema'
  }
}
```

Call the doFirst on the task object (top) and inside task's configuration block (bottom)

```
task setupDatabaseTests << {
  println 'load test data'
}

setupDatabaseTests.doFirst {
  println 'create database schema'
}

setupDatabaseTests.doFirst {
  println 'drop database schema'
}
```

Repeated calls to the doFirst method are additive. Each previous call's action code is retained, and the new closure is appended to the start of the list to be executed in order.

Gradle – onlyIf() Examples

```
task createSchema << {
  println 'create database schema'
}

task loadTestData(dependsOn: createSchema) << {
  println 'load test data'
}

loadTestData.onlyIf {
  System.properties['load.data'] == 'true'
}
```

Using onlyIf method to do simple system property tests

- onlyIf method can be used to switch individual tasks on and off using any logic you can express in Groovy code
- You can read files, call web services, check security credentials, or just about anything else

Gradle – Default Task's Properties

Method	Description
didWork	A boolean property indicating whether the task completed successfully
enabled	A boolean property indicating whether the task will execute.
path	A string property containing the fully qualified path of a task (levels; DEBUG, INFO, LIFECYCLE, WARN, QUIET, ERROR)
logger	A reference to the internal Gradle logger object
logging	The logging property gives us access to the log level
temporaryDir	Returns a File object pointing to a temporary directory belonging to this build file. It is generally available to a task needing a temporary place in to store intermediate results of any work, or to stage files for processing inside the task
description	a small piece of human-readable metadata to document the purpose of a task

Gradle – Dynamic Properties

- Properties (other than built-in ones) can be assigned to a task
- A task object functions can contain other arbitrary property names and values we want to assign to it (do not use built-in property names)

Scenario: copyFiles task should collect files from several sources and copy them into a staging directory, which the createArtifact task will later assemble into a deployment artifact.
The list of files may change depending on the parameters of the build, but the artifact must contain a manifest listing them, to satisfy some requirement of the deployed application

```
task copyFiles {
  // Find files from wherever, copy them
  // (then hardcode a list of files for illustration)
  fileManifest = [ 'data.csv', 'config.json' ]
}

task createArtifact(dependsOn: copyFiles) << {
  println "FILES IN MANIFEST: ${copyFiles.fileManifest}"
}
```

Gradle – Task Types (Copy)

- A copy task copies files from one place into another
- In its most basic form, it copies files from one directory into another, with optional restrictions on which file patterns are included or excluded

Example:
Create the destination directory if it doesn't already exist.
The copyFiles task will copy any files with the .xml, .properties, or .txt extensions
from the resources directory to the target directory

```
task copyFiles(type: Copy) {
from 'resources'
into 'target'
include '**/*.xml', '**/*.txt', '**/*.properties'
}
```

Note: that the from, into, and include methods are inherited from the Copy

Gradle – Task Types (Jar)

- A Jar task creates a Jar file from source files
- The Java plug-in creates a task of this type, called jar
- It packages the main source set and resources together with a trivial manifest into a Jar bearing the project's name in the build/libs directory
- The task is highly customizable.

```
apply plugin: 'java'

task customJar(type: Jar) {
  manifest {
    attributes firstKey: 'firstValue', secondKey: 'secondValue'
  }
  archiveName = 'hello.jar'
  destinationDir = file("${buildDir}/jars")
  from sourceSets.main.classes
}
```

Gradle – Task Types (JavaExec)

- A JavaExec task runs a Java class with a main() method
- Command-line Java can be a hassle, but this task tries to take the hassle away and integrate command-line Java invocations into your build.

```
apply plugin: 'java'

repositories {
  mavenCentral()
}

dependencies {
  runtime 'commons-codec:commons-codec:1.5'
}

task encode(type: JavaExec, dependsOn: classes) {
  main = 'org.gradle.example.commandline.MetaphoneEncoder'
  args = "The rain in Spain falls mainly in the plain".split().toList()
  classpath sourceSets.main.classesDir
  classpath configurations.runtime
}
```

Gradle – Custom Task Types

- Gradle's built-in tasks are useful, but it might not be sufficient for all scenarios

- Gradle allows defining custom task types in many ways, most commonly in
  - **The Build File**: the custom task is created in the build file and must extend the DefaultTask class or one of its descendants

  - **The Source Tree**: the custom task is sophisticated (has significant custom logic) and has its own class hierarchy, might rely on external API and need automated testing
    - When the custom task logic outgrows the build file, it can be migrated to the **buildSrc** directory at the project root
    - This directory is automatically compiled and added to the build classpath

Gradle – Custom Task Types (Build File)

- Suppose your build file needs to issue arbitrary queries against a MySQL database

```
task createDatabase(type: MySqlTask*) {
  sql = 'CREATE DATABASE IF NOT EXISTS example'
}

task createUser(type: MySqlTask*, dependsOn: createDatabase) {
  sql = "GRANT ALL PRIVILEGES ON example.*
        TO exampleuser@localhost IDENTIFIED BY 'passw0rd'"
}

task createTable(type: MySqlTask*, dependsOn: createUser) {
  username = 'exampleuser'
  password = 'passw0rd'
  database = 'example'
  sql = 'CREATE TABLE IF NOT EXISTS users
        (id BIGINT PRIMARY KEY, username VARCHAR(100))'
}
```

* Actual build tasks inherits MySQLType's properties and actions

```
class MySqlTask* extends DefaultTask {
  def hostname = 'localhost'
  def port = 3306
  def sql
  def database
  def username = 'root'
  def password = 'password'

  @TaskAction
  def runQuery() {
    def cmd
    if(database) {
      cmd = "mysql -u ${username} -p${password} -h ${hostname}
      -P ${port} ${database} -e "
    }
    else {
      cmd = "mysql -u ${username} -p${password} -h ${hostname} -P ${port} -e "
    }
    project.exec {
      commandLine = cmd.split().toList() + sql
    }
  }
}
```

Task's properties (Groovy idiom)

Task method will run when the task runs

Gradle – Custom Task Types (Source Tree)

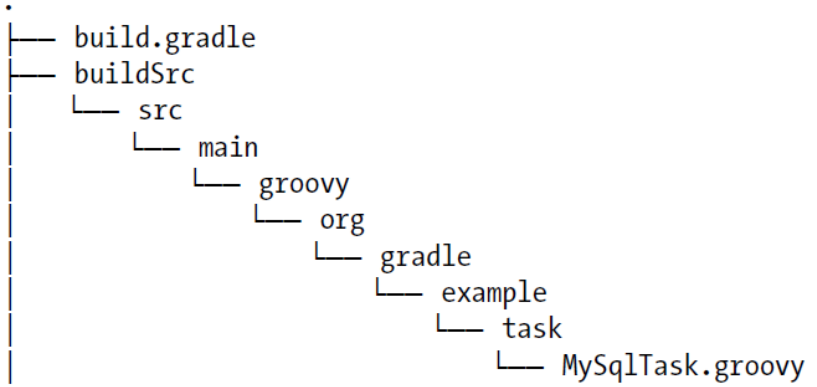
```
import org.gradle.api.DefaultTask
import org.gradle.api.tasks.TaskAction

class MySqlTask extends DefaultTask {
  def hostname = 'localhost'
  def port = 3306
  def sql
  def database
  def username = 'root'
  def password = 'password'

  @TaskAction
  def runQuery() {
    def cmd
    if(database) {
      cmd = "mysql -u ${username} -p${password} -h ${hostname}
            -P ${port} ${database} -e "
    }
    else {
      cmd = "mysql -u ${username} -p${password} -h ${hostname} -P ${port} -e "
    }
    project.exec {
      commandLine = cmd.split().toList() + sql
    }
  }
}
```

Note this the task definition in the buildSrc directory is very similar to the code included in the build script in the previous example. However, we now have a robust platform for elaborating on that simple task behavior, growing an object model, writing tests, and doing everything else we normally do when developing software.

Gradle – Custom Task Types (Source Tree Code Structure)

Where to put custom Gradle build code?

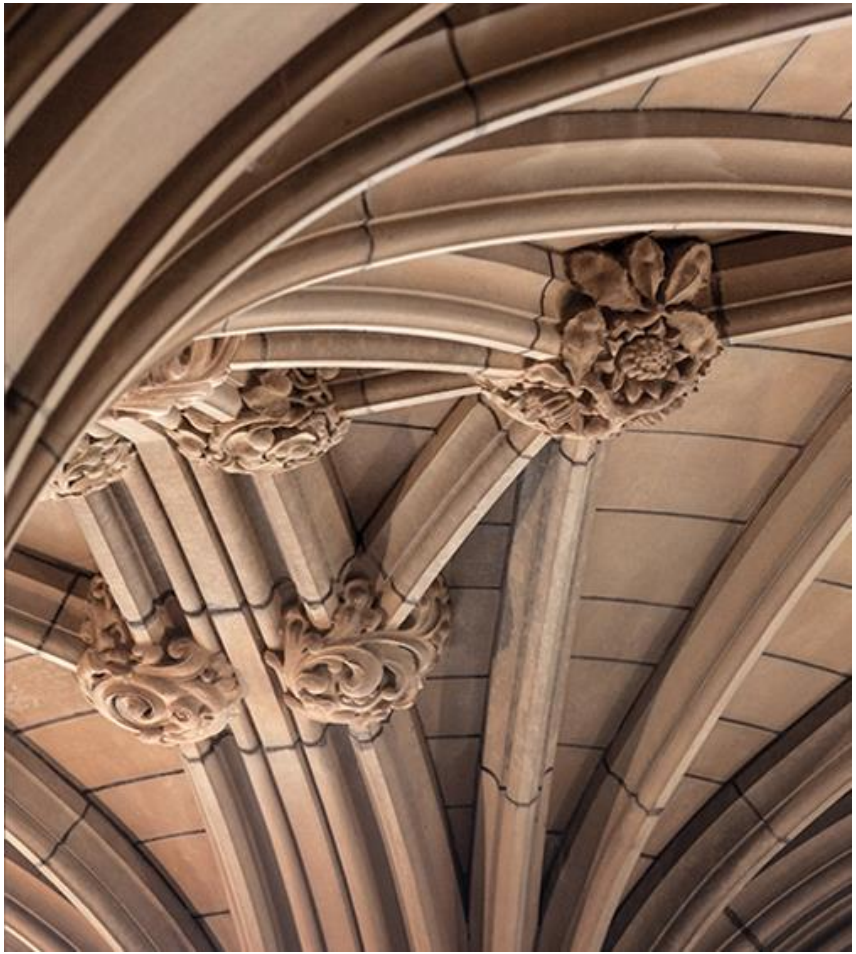
- In the **build script itself**: in a task action block
- In the **buildSrc directory**: example in previous slide
- In a **separate build script file**: to be imported into the main build script
- In a **custom plug-in**: written in Java or Groovy. Programming Gradle with custom plug-ins will be the topic of a separate volume



Example. The structure of a Gradle project with custom code in the buildSrc directory

Gradle – Plug-ins

Java Plug-in



THE UNIVERSITY OF SYDNEY

Gradle – Java Plug-in

- Gradle is extensible through plug-ins (e.g., Java plug-in)
- A plug-in is a extension to Gradle which configures projects (by adding some pre-configured tasks which together achieve some goals)
- **Java plug-in** adds some tasks to your project which will compile and unit test your Java source code, and bundle into a JAR
  - Convention-based; default values for many aspects of the project are pre-defined (e.g., location of Java source files)
    - Specify as *apply plugin : 'java'* in the **build.gradle**
  - Also you can customize projects if you do not follow the convention

https://docs.gradle.org/current/userguide/java_plugin.html

Gradle – Java Plug-in (Project Structure)

- Gradel expects to find production source code under ***src/main/java*** *and test source code under* ***src/test/java***
- Files under ***src/main/resources*** will be included in the JAR as resources
- Files under ***src/test/resources*** will be included in the classpath used to run tests
- All output files are created under the build directory, with the JAR file will end up in the ***build/libs*** directory

https://docs.gradle.org/current/userguide/java_plugin.html

Gradle – Java Plug-in (Project Build)

- Java plug-in will add a few tasks (the project may need some tasks though)
- Run *gradle tasks* to list the tasks of a project
- Gradle will compile and create a JAR file containing main classes and resources – run *gradle build*

```
> gradle build
> Task :compileJava
> Task :processResources
> Task :classes
> Task :jar
> Task :assemble
> Task :compileTestJava
> Task :processTestResources
> Task :testClasses
> Task :test
> Task :check
> Task :build

BUILD SUCCESSFUL in 0s
6 actionable tasks: 6 executed
```

Example of output of gradle build

https://docs.gradle.org/current/userguide/java_plugin.html

Gradle – Java Plug-in (Project Build)

- Some other useful tasks
- **clean**
  - Deletes the build directory, removing all built files.
- **assemble**
  - Compiles and jars your code, but does not run the unit tests. Other plugins add more artifacts to this task. For example, if you use the War plugin, this task will also build the WAR file for your project.
- **check**
  - Compiles and tests your code. Other plugins add more checks to this task. For example, if you use the checkstyle plugin, this task will also run Checkstyle against your source code.

https://docs.gradle.org/current/userguide/java_plugin.html

Gradle – Java Plug-in (Dependencies)

- To reference external JAR files that the project is dependent on, tell gradle where to find it
  - JAR files are located in a repository (contains artefacts/dependencies needed for a project)
  - Different repositories types are supported in gradle (see Gradle Repository Types)
  - Example (using Central Maven Repository)

build.gradle

```
1 ....
2 repositories {
3     mavenCentral()
4 }
5 dependencies {
6     compile group: 'commons-collections', name: 'commons-collections', version: '3.2.2'
7     testCompile group: 'junit', name: 'junit', version: '4.+'
8 }
```

production classes have a compile-time dependency on commons collections

Test classes have a compile-time dependency on junit

https://docs.gradle.org/current/userguide/java_plugin.html

Gradle – Java Plug-in (Project Customization)

- The Java plug-in adds many properties with default values to a projects
- It is possible to customize default values to suit project needs
- Use *gradle properties* to list properties of a project
- Example; in the below customizing the Java version number, and attributes to the JAR manifest

build.gradle

```
1 ...
2 version = '1.0'
3 jar {
4     manifest {
5         attributes 'Implementation-Title': 'Gradle Quickstart',
6                    'Implementation-Version': version
7     }
8 }
```

https://docs.gradle.org/current/userguide/java_plugin.html

Gradle – Java Plug-in (Publish JAR file)

- Artefacts such as JAR files can be published to repositories (local, remote, or multiple locations)
- To publish a JAR file tell gradle where you want to do so
  - Use gradle *uploadArchives* to publish a JAR file

build.gradle

```
...
uploadArchives {
    repositories {
       flatDir {
           dirs 'repos'
       }
    }
}
```

Publish a JAR file to a local repository

https://docs.gradle.org/current/userguide/java_plugin.html

Gradle – Complete Build file for Java

build.gradle

```
apply plugin: 'java'
apply plugin: 'eclipse'

version = '1.0'
jar {
    manifest {
        attributes 'Implementation-Title': 'Gradle Quickstart',
                   'Implementation-Version': version
    }
}

repositories {
    mavenCentral()
}

dependencies {
    compile group: 'commons-collections', name: 'commons-collections', version: '3.2.2'
    testCompile group: 'junit', name: 'junit', version: '4.+'
}

test {
    systemProperties 'property': 'value'
}

uploadArchives {
    repositories {
       flatDir {
           dirs 'repos'
       }
    }
}
```

Eclipse plug-in to create the Eclipse-specific descriptor files, like .project

https://docs.gradle.org/current/userguide/java_plugin.html

Gradle – Defining Multi-project Java Build

- Imagine a project called multiproject with the following layout:
  - Project API shipped to the client to provide a java client for your XML webservice
  - Project webservice which returns XML
  - Project shared contains code used by both API and webservices
  - Project services/shared has code that depend on the shared project

```
multiproject/
  api/
  services/webservice/
  shared/
  services/shared/
```

- To define multi-project build, create *settings.gradle file* (in the source tree root directory) and specify which projects to include in the build

settings.gradle

```
include "shared", "api", "services:webservice", "services:shared"
```

https://docs.gradle.org/current/userguide/java_plugin.html

Gradle – Configuring Multi-project Java Build

- ***Configuration Injection***: a technique to specify common configuration in the root project. The root project is like a container and the subprojects method iterates over the container's elements (projects) and injects the specified configuration

build.gradle

```
subprojects {
    apply plugin: 'java'
    apply plugin: 'eclipse-wtp'

    repositories {
       mavenCentral()
    }

    dependencies {
        testCompile 'junit:junit:4.12'
    }

    version = '1.0'

    jar {
        manifest.attributes provider: 'gradle'
    }
}
```

Apply java plug-in to each subproject – all tasks and configuration properties are available in each project

Run gradle build from the root project directory to compile, test and JAR all the projects

Build will not expect to find source files in the root project (only in the subprojects) as the plug-ins are only applied to the subprojects

https://docs.gradle.org/current/userguide/java_plugin.html

Gradle – Multi-project Build (Dependency)

- Let's say the JAR file of one project is used to compile another project – how to specify such dependency?
  - Example: in the API build file is dependent on the shared project

api/build.gradle

```
dependencies {
    compile project(':shared')
}
```

Due to the specified dependency, gradle will ensure that project shared always gets built before the project api

https://docs.gradle.org/current/userguide/java_plugin.html

References

- Ian Sommerville 2016. Software Engineering: Global Edition (3rd edition). Pearson, Englad
- Tim Berglund and Matthew McCullough. 2011. Building and Testing with Gradle (1st ed.). O'Reilly Media, Inc.