

Gradle 교육 문서

Gradle 활용 및 응용 편

문서번호 :

VER 1.0

1	개요	5
2	Gradle 이란?	5
2.1	특징	5
2.2	Maven 과 무엇이 다른가 !	6
2.3	단점	7
3	테스트 환경 및 설치	7
3.1	테스트환경	7
3.2	테스트환경	7
4	프로젝트 생성	8
4.1	터미널에서의 프로젝트 생성	8
4.2	STS 에서의 Gradle 의 적용	8
5	Gradle 을 위한 기본 Groovy 언어	10
5.1	변수선언	10
6	build.gradle 스크립트 작성	11
6.1	plugin 설정	11
6.2	plugin 설정	12
6.3	plugin 설정	12
6.4	의존성 구성(dependency configurations)	13
6.5	의존성 구성(dependency configurations)	19
7	build.gradle 스크립트 작성	22
7.1	테스트 실행	23
7.2	System properties	23
7.2	테스트 감지	24

7.3	JUnit.....	24
7.4	TestNG.....	24
7.5	테스트의 분리	25
7.6	TestLogging	25
8	Gradle Life Cycle	26
9	부록 Profile 흉내내기	27
10	부록 Gradle Multi Project.....	27
1.	최상위 프로젝트의 이름	27
2.	멀티 프로젝트의 기본	28
3.	교차 프로젝트 구성 Cross Project Configuration.....	28
4.	서브 프로젝트 구성.....	30
5.	멀티 프로젝트 빌드 실행 규칙.....	34
6.	절대 경로로 태스크 실행하기	35
7.	프로젝트와 태스크의 경로.....	35
8.	의존성 - 어느 의존성을 선택?.....	36
9.	프로젝트 lib 의 존성.....	41
10.	분리된(decoupled) 프로젝트	43
11.	멀티 프로젝트 빌드와 테스트	43
12.	프라퍼티와 메소드 상속	44
13.	멀티 프로젝트 단위 테스트간의 의존성	44

제.개정내역

버전	승인일자	개요	작성자	승인자
1.0	2014.04.07	제정	백성진	

1 개요

- 본 문서에서는 Gradle 교육을 진행하면서 준수해야 할 전반적인 개발환경에 대한 표준을 식별하고 교육 내용을 기술한다

2 Gradle 이란?

- Gradle 은 진화된 빌드툴로 빌드, 테스트, 배포, 개발 등을 자동화 할 수 있다.
- Ant 의 유연성과 효과적인 빌드툴인 Maven 의 편리성을 조합하여 많은 오픈소스 프로젝트의 빌드 시스템으로 빠르게 채택 되고 있다.
- Ant 의 가장 큰 장점은 개발자가 자유롭게 빌드 단위(target)를 지정하고 빌드 단위간의 의존관계를 자유롭게 설정할 수 있다는 것이다. 하지만 자유도가 높다는 것은 잘 활용할 경우 좋은 도구가 될 수 있지만 그렇지 않을 경우 애물단지로 전락할 가능성이 있다.
- Maven 의 가장 큰 장점은 Convention Over Configuration 전략에 따라 프로젝트 빌드 과정에 대한 많은 부분이 이미 관례로 저해져 있다는 것이다. 따라서 Maven 기반 프로젝트를 경험한 개발자는 Maven 을 기반으로 하고 있는 새로운 프로젝트에서도 쉽게 적용할 수 있다는 것이다. 하지만 관례가 항상 좋은 것은 아니며, 특수한 상황이 발생하는 경우에는 맞지 않는 경우도 종종 발생한다.
- Gradle 은 Ant 의 자유도와 Maven 의 관례의 장점을 모두 흡수했다. 그리고 Ant 와 Maven 빌드 툴 모두 가지고 있었던 가장 큰 단점인 XML 에 대한 이슈도 Groovy 언어를 사용해 해결하고 있다.

2.1 특징

- ◆ Groovy 기반의 DSL(Domain Specific Language) 채용
- ◆ 의존성 기반 프로그래밍을 위한 언어
- ◆ 빌드의 구조화
- ◆ API 제공
- ◆ Multi Project 빌드
- ◆ 의존성 관리의 다양한 방법 제공
- ◆ migration 의 편리성
- ◆ build script 는 xml 이 아닌 Groovy 로 작성
- ◆ 오픈소스로 제공

2.2 Maven 과 무엇이 다른가 !

- ◆ 프로젝트 구성과 빌드는 근본적으로 “구성”이라는 정적인 요소와 “빌드”라는 동적인 요소의 집합이다. 이를 Maven은 정적인 데이터를 저장하는 XML로 만들어서 동적인 행위에 대한 정의를 어렵게 만들었다.
- ◆ Maven의 가장 큰 문제이며 이로 인한 복잡한 프로젝트에서 설정이 거의 불가능한 상황이 자주 발생한다.
- ◆ Gradle은 DSL로 설정 정보를 구성하고, 그 자체가 groovy 스크립트 언어이므로 동적인 작업은 그냥 groovy 코드로 즉석에서 작성하면 된다.
- ◆ Maven은 상속 구조를 사용해 멀티 모듈을 구현한다. Gradle은 구성 주입(Configuration Injection)을 사용한다.
- ◆ Maven에서 특정 설정을 몇몇 모듈에서만 공통으로 사용하려면 불필요하게 부모 프로젝트를 생성하여 설정하고 그것을 자식들이 상속하게 해야 한다. 게다가 다른 모든게 같더라도 약간이라도 설정이 다른 프로젝트가 하나라도 있다면 그 프로젝트는 상속을 할 수 없고, 거의 모든 설정을 중복해서 해당 프로젝트에 넣어줘야 한다.
- ◆ Gradle은 공통 설정을 조건에 따라 특정 프로젝트에만 주입 가능하다. 불필요한 프로젝트는 필요없다.
- ◆ 프로젝트에 상대적인 파일 경로로 작업을 할 때 Gradle은 “rootProject.file()”로 쉽게 구성 가능하다.
- ◆ Maven은 자신만의 플러그인을 만들기가 힘들다. 하지만 Gradle은 “build.gradle” 혹은 “buildSrc”를 통해 자신만의 플러그인과 태스크를 매우 쉽게 정의할 수 있다.
- ◆ Gradle은 Ant태스크를 바로 가져다가 사용할 수 있기 때문에 수많은 Java Ant태스크들을 이미 내장하고 있는 것이나 다름없다.
- ◆ Gradle은 태스크간의 작동 순서 정의가 매우 쉽다. Maven은 정적인 특성 때문에 특정 태스크를 반복 수행하거나 하는 등의 작업이 힘들고, 다른 구문에 태스크를 끼어 넣는 것도 직관적이지 못하다.
- ◆ Gradle은 Maven플러그인으로 있으나, Gradle 혹은 못 플러그인이 없을 경우 그냥 외부 프로그램을 실행해버리거나 groovy로 Maven 플러그인의 java 코드를 호출해서 실행하면 된다.

2.3 단점

- ◆ 의존성에서 "provided"를 기본으로 제공하지 않고 있다. 하지만 "configurations"를 직접 구성하는 방법이 있다. **의존성 구성(dependency configurations)** 참조.
- ◆ Maven 보다 프로젝트 컴파일 / 빌드 속도가 느리다.
- ◆ 이행적 의존성(transitive dependency) 충돌로 인해 자신도 모르게 지정한 것 보다 높은 버전의 라이브러리를 받아오는 현상이 생긴다. 이것은 문제라기 보다는 Gradle 의 의도인데, 이것을 이해하지 못하면 앞서 말한 현상이 생긴다. **오류! 참조 원본을 찾을 수 없습니다.** 참조.
- ◆ IDE(Integrated Development Environment)의 지원이 다소 미흡하다.

3 테스트 환경 및 설치


3.1 테스트환경

- ◆ Spring Tool Suite(STS) 3.5.0-Release & MacOS OSX 10.9.2 & Win 7
- ◆ Gradle 1.11
- ◆ JDK & JRE 8


3.2 테스트환경

- ◆ 마켓 플레이스에서 gradle 검색 후 아래의 이미지 버전을 설치하면 된다. (이미지는 이미 설치 된 버전을 캡처 한 것)

Gradle Integration for Eclipse 3.5.0.RELEASE

 The Eclipse-Integration-Gradle project brings you developer tooling for Gradle into Eclipse. It comes with Spring UAA (User Agent Analysis), an optional component... [more info](#)

by Pivotal, EPL
[gradle](#)

★ 15  Installs: **41.0K** (3,566 last month) [Update](#) [Uninstall](#)

- ◆ STS 마켓 플레이스에서 gradle 검색 시 같이 나오는 "Minimalist Gradle Editor"는 Syntax highlight 기능을 제공한다.

Minimalist Gradle Editor 0.12



Minimalist Gradle Editor for build.gradle files with highlight for keywords, strings and matching brackets. It also takes some additional keywords from android... [more info](#)

by Nodeclipse/Enide, GPL

[gradle](#) [editor](#) [highlight](#) [build](#) [android](#)

★ 1

Installs: 2.12K (441 last month)

Update Uninstall

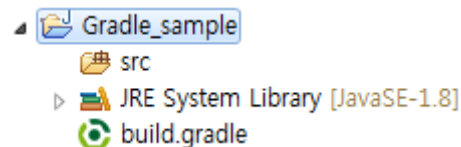
4 프로젝트 생성

4.1 터미널에서의 프로젝트 생성

- ◆ 프로젝트를 생성을 원하는 폴더를 생성한다.
- ◆ build.gradle 파일을 생성하고 오류! 참조 원본을 찾을 수 없습니다. 같이 스크립트를 작성한다.
- ◆ gradle 의 ⅔ 는 build.gradle 작성에 달려있다. 작성법을 유심히 보고 사용할 것을 권장한다.

4.2 STS 에서의 Gradle 의 적용

- ◆ 기존 프로젝트에서의 build.gradle 파일 생성



- ◆ build.gradle 파일에 아래와 같은 빌드스크립트(Groovy) 작성

- 빌드스크립트 작성 방식은 오류! 참조 원본을 찾을 수 없습니다. 참고

```
//사용 plug-in 설정
apply plugin: 'java'
apply plugin: 'war'
apply plugin: 'eclipse-wtp'
apply plugin: 'maven'

//java version
sourceCompatibility = 1.8

//개발한 어플리케이션 버전
version = '1.0'

//Jar manifest에 몇가지 속성 추가. 미사용시 Java 보안 업데이트 에서 차단이 될수 있음.
jar {
    manifest {
        attributes 'Implementation-Title': 'Gradle Quickstart', 'Implementation-Version': version
    }
}

//사용할 repo 선언
```



```

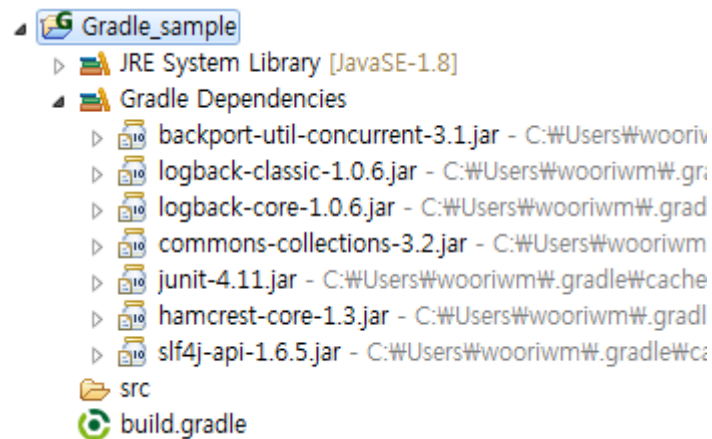
repositories {
    mavenCentral()
}

//dependencies를 추가하기 위해 compile group과 같은 몇몇의 필수요소를 선언한다.
dependencies {
    compile group: 'commons-collections', name: 'commons-collections', version: '3.2'
    compile 'backport-util-concurrent:backport-util-concurrent:3.1'
    compile 'ch.qos.logback:logback-classic:1.0.6'
    testCompile group: 'junit', name: 'junit', version: '4.+
'
}

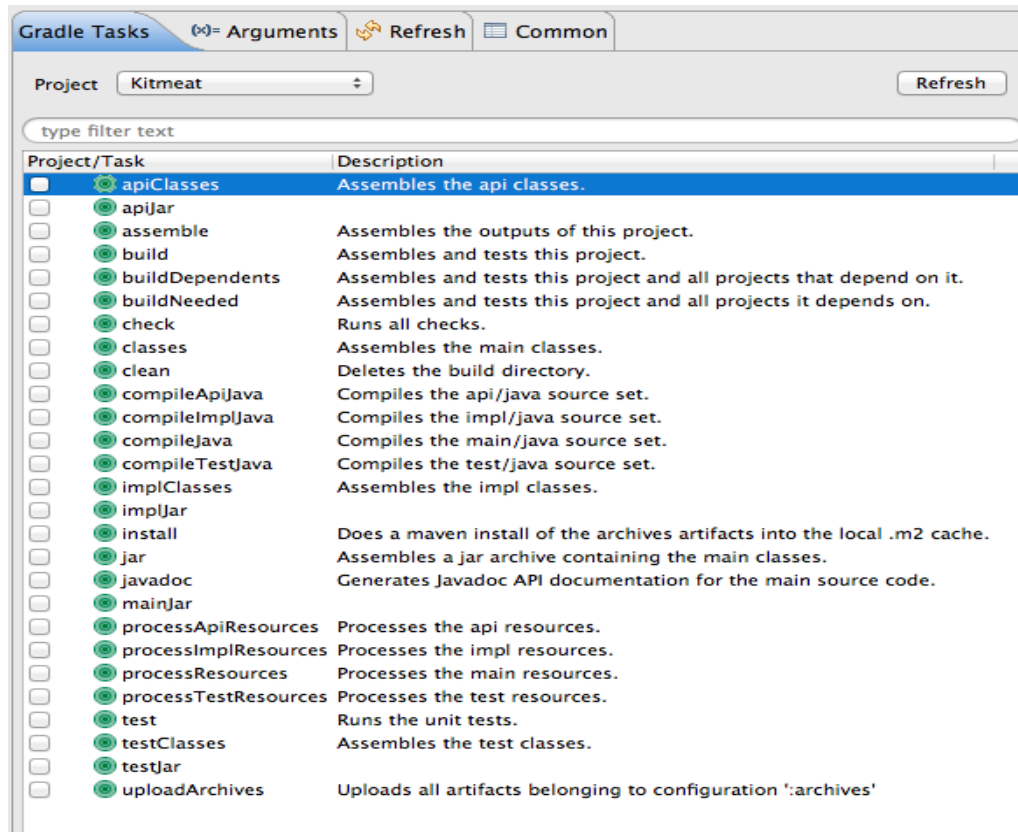
test {
    jvmArgs = ["-ea" , "-Xmx256m" ]
    logging.captureStandardOutput(LogLevel.INFO)
    testReport = false
}

```

■ 프로젝트 -> Configure -> Convert to Gradle Project 로 변환



■ Run As -> Gradle Build -> 실행하고자 하는 Task 선택 후 -> Run (ex 로 Build 를 실행)



5 Gradle을 위한 기본 Groovy 언어

- Gradle 빌드 스크립트를 제대로 이해하려면 Groovy 언어를 알면 한결 용이하다. Groovy 언어 전체를 학습하려면 비용이 많이 들기에 빌드에 필요한 것 위주로 정리해본다.

5.1 변수선언

◆ 변수 선언

- 로컬 변수: `def` 변수명 으로 선언. 해당 스크립트 로컬에서만 접근 가능하다
- `Ext` 변수 : 프로젝트 전체와 서브 프로젝트에서도 접근 가능하다.
- `{ }` 사용.

```
ext {
    springVersion = "3.4.0.RELEASE"
    emailNotification = "erimankr@gmail.com"
}
```

Gradle 에서 가장 많이 볼수 있는 표현법이다.

이 표현법은 `ext` 변수로 `springVersion` 이라는 이름으로 객체를 할당한다.

◆ Closure

```
dependencies {
    assert delegate == project.dependencies
    compile('junit:junit:4.11')
    delegate.compile('junit:junit:4.11')
}
```

- Gradle을 처음 접할 때, Closure 잘 모르면 황당하다. 문서 곳곳에서는 compile() 메소드를 사용하지만 어느곳에도 compile() 메소드는 존재하지 않는다.
- Gradle의 상당 부분은 closure로 구현되어 있으므로 개념을 확실히 잡아야 한다.
- 클로저는 함수 객체(function objects) 또는 익명 함수(anonymous function)의 영역안에 묶여 버리는 추상적인 묶음이라고 표현 할 수 있다
- 그루비의 클로저는 코드 블록 혹은 메서드 포인터와 같다. 클로저는 정의 후 나중에 실행할 코드 조각을 말한다
- 정확한 가이드는 <http://groovy.codehaus.org/Closures+-+Formal+Definition> 를 참고한다.

◆ List 와 Map Literal

```
//List
test.includes = ['org/gradle/api/**', 'org/gradle/internal/**']

List<String> list = new ArrayList<String>()
list.add('org/gradle/api/**')
list.add('org/gradle/internal/**')
test.includes = list

//Map literal
apply plugin: 'java'

Map<String, String> map = new HashMap<String, String>()
map.put('plugin', 'java')
apply(map)
```

◆ Left Shift Operator(<<)에 대한 이해

```
task hello << {
    println 'Hello world!'
}
```

- doLast method와 같은 목적으로 사용할 수 있다

```
task first {
    doFirst {
        println 'Running first'
    }
}

task second {
    doLast { Task task ->
        println "Running ${task.name}"
    }
}
```

```

    }
}

task third << { taskObject ->
    println 'Running' + taskObject.name
}

```

- "<<"는 doLast와 같은 의미이다. doFirst/doLast는 여러 개 선언될 수 있으며, doFirst가 선언된 순서로 먼저 실행되고, 그 뒤에 doLast가 선언된 순서대로 실행된다.

6 build.gradle 스크립트 작성

6.1 plugin 설정

- ◆ java project 를 빌드하는 과정에는 source file compile, 단위 테스트, 그리고 jar 파일 생성이라고 한다면 이 일련의 단계들을 gradle 의 **"java plugin"**이 task 로 포함하고 있어 모든 과정들이 내부적으로 수행된다.
- ◆ plugin 에 관한 더 많은 정보는 <http://www.gradle.org/plugins> 에서 확인 할 수 있다.

```

apply plugin: 'java'
apply plugin: 'war'

```

6.2 plugin 설정

- ◆ 위에서 사용한 플러그인들은 프로젝트에 많은 속성을 추가하는데, 이러한 속성들은 충분히 기본 값을 가지고 있다. 소스에서 사용되는 자바.
- ◆ 또한 JAR manifest 에 몇 가지 속성을 추가 할 수 있다. 다음의 사이트에서 manifest 속성을 확인 할 수 있다.

<http://www.gradle.org/docs/current/javadoc/org/gradle/api/java/archives/Manifest.html>

```

//java versio
sourceCompatibility = 1.8

springVersion = '3.4.0.RELEASE'
logBackVersion = '1.0.6'

//Jar manifest에 몇가지 속성 추가. 미사용시 Java 보안 업데이트 에서 차단이 될수 있음.
jar {
    manifest {
        attributes 'Implementation-Title': 'Gradle Quickstart', 'Implementation-Version': version
    }
}

```

6.3 plugin 설정

- ◆ java 프로젝트는 외부 JAR 파일들에 의존할 경우가 많아, 프로젝트에 jar 파일들을 추가하기 위해서는 repository 에 존재해야 한다. Maven repository 에서 사용하듯이 repository 는 프로젝트의 dependencies 를 가져올 때 또는 artifacts 의 배포에 사용 될 수 있다
- ◆ repositories 정의는 default 값이 없으므로 꼭 정의가 필요하다. 정의 방법은 아래와 같다.
- ◆ 아래의 이미지처럼 "mavenCentral()"을 사용 하면 기본 Maven repositories 를 명시한다.

```
repositories {  
    mavenCentral()  
}
```

- ◆ 또는 url 을 통하여 maven repository 를 remote 해서 사용할 수 있다.

```
repositories {  
    maven {  
        url "http://nexus.wooriwm.com/nexus/"  
    }  
}
```

- ◆ ivy repository 를 remote 하여 사용 할 수도 있으며.

```
repositories {  
    ivy {  
        url "http://nexus.wooriwm.com/repo/"  
    }  
}
```

- ◆ 로컬 저장소를 참조하여 사용 할 수도 있다

```
repositories {  
    ivy {  
        url "../local-repo"  
    }  
}
```

- ◆ maven{}을 여러번 사용하는 것이 가능하며, 사용자 정의 repository 에 접속 계정정보를 추가 할 경우이다

```
repositories {  
    maven {  
        credentials {  
            username 'user'  
            password 'password'  
        }  
        url "http://nexus.wooriwm.com/nexus/"  
    }  
}
```

6.4 의존성 구성(dependency configurations)

- ◆ Java 와 의존성 관리

- 원칙적으로 자바에는 의존성 관리 기법이 존재하지 않기 때문에 Maven, Ivy 와 같은 비표준 솔루션이 만들어지게 되었다
- Maven 은 완전한 빌드 시스템이며, Ivy 는 의존성 관리만 한다.
- Maven 과 Ivy 모두 특정 jar 에 대한 의존성 정보를 기술하는 XML 파일 기술자(descriptor)를 통해 의존성을 관리한다
- Gradle 의존성 분석 엔진은 pom(Maven)과 ivy 기술자를 모두 지원한다

◆ 의존성 구성

- Java 에서 의존성은 configurations 으로 그룹화 된다. 구성의 각 그룹은 classpath 를 의미한다.
- 많은 Gradle 플러그인들은 의존성 구성을 미리 정의해 두고 있다. 또한 사용자가 스스로 자신만의 구성을 추가할 수 있다. (빌드시에는 필요 없지만 배포는 같이 되어야 하는 추가 JDBC 드라이브 같은 것들이 해당)
- 프로젝트 구성은 configurations 객체로 관리한다. configurations 객체에 클로저를 전달하면 이 클래스의 API 가 호출된다
- Java 에서는 기본적으로 네 가지 configuration 이 존재한다

configuration	discription
compile	프로젝트를 컴파일할 때 필요한 의존성 라이브러리들을 추가한다.
runtime	프로젝트를 실행할 때 필요한 의존성 라이브러리들을 추가한다. 기본적으로 컴파일 타임을 모두 포함한다.
testCompile	테스트 소스 프로젝트에서 컴파일 시 필요한 의존성을 추가한다. 기본적으로 컴파일 된 클래스들과 컴파일 의존성을 포함한다.
testRuntime	테스트가 실행 될 때 필요한 의존성을 추가한다. 기본적으로 컴파일, 런타임과 테스트 컴파일의 의존성도 포함한다.

◆ 구성하기

■ Configurations 의 항목은

<http://www.gradle.org/docs/current/dsl/org.gradle.api.artifacts.Configuration.html> 에 기술 된 객체이다.

```
configurations {
    compile {
        description = 'compile classpath'
        transitive = true
    }

    runtime {
        extendsFrom compile
    }
}

configurations.compile {
    description = 'compile classpath'
}
```

■ 외부 모듈 의존성을 추가하는 방식은 두 가지로 **맵 지정방식**과 **문자열 지정방식**이 있다

```
dependencies {
    compile group: 'org.hibernate', name: 'hibernate-core', version:
    '3.6.7.Final'
}

dependencies {
    compile 'org.hibernate:hibernate-core:3.6.7.Final'
}
```

■ <http://www.gradle.org/docs/current/javadoc/org/gradle/api/artifacts/ExternalModuleDependency.html>

에서 더 많은 property 와 method 를 볼 수 있다

- 의존성이 선언되면 그에 맞는 기술자 파일(pom.xml, ivy.xml)을 찾고, 그에 따라 해당 모듈의 artifact jar 파일과 그 의존하는 파일들을 다운로드 한다.
- 만약 파일이 존재하지 않으면 바로 적합한 파일명을 구성하여 다운로드 한다. (존재하지 않는 버전을 명시 하였을 경우는 deprecated 됨)
- Maven 에서는 모듈이 하나의 artifact 만 존재하지만 Gradle 또는 Ivy 는 하나의 모듈이 여러 개의 artifact 를 가질 수 있다. 각 artifact 는 서로 다른 의존성을 가질 수 있다

◆ 분류자 Classifier

- Maven 에는 분류자(classifier)가 존재한다. 분류자는 @ 확장자 지정자와 함께 사용할 수 있다.

```
dependencies {
    compile group:
    "org.hibernate.test.classifiers:service:1.0:jdk15@jar"
    otherConf group: 'org.gradle.test.classifiers', name: 'service',
```

```
version: '1.0', classifier: 'jdk15'
}
```

◆ 특정 구성의 외부 의존성 목록 보기

- 다음 task 를 만들고 터미널에서 "gradle -q listJars"로 실행한다. 혹은 task 없이 "gradle dependencies"만 해도 볼 수 있다.

```
task listJars << {
    configurations.compile.each { File file -> println file.name }
}
```

◆ 클라이언트 모듈 의존성

- 클라이언트 모듈 의존성은 빌드 스크립트에서 직접 이행적 의존성을 선언할 수 있게 해준다. 즉, pom.xml 과 같은 모듈 기술자를 대체하는 기법이다.
- 아래 설정에서는 현재 프로젝트가 groovy 에 의존하지만, groovy 자체의 의존성은 무시하고 빌드 파일에서 직접 지정한 의존성을 따르도록 한다.

```
dependencies {
    runtime module("org.codehaus.groovy:groovy-all:1.8.7") {
        dependency("common-cli:commons-cli:1.0") {
            transitive = false
        }
        module(group: 'org.apache.ant', name: 'ant', version
'1.8.4') {
            dependencies "org.apache.ant:ant-
launcher:1.8.4@jar", "org.apache.ant:ant-junit:1.8.4"
        }
    }
}
```

- <http://www.gradle.org/docs/current/javadoc/org/gradle/api/artifacts/ClientModule.html> 참조.

◆ 프로젝트 의존성

- 멀티 프로젝트에서 다른 프로젝트에 대한 의존성을 설정할 수 있다

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

- 이는 "shared"라는 프로젝트에 의존성을 설정 한 것이다

- <http://www.gradle.org/docs/current/javadoc/org/gradle/api/artifacts/ProjectDependency.html> 참조

◆ 파일 의존성

- 파일 의존성을 사용하면 jar 파일을 repository 에 넣지 고도 의존성에 추가하는 것이

가능하다

- FileCollection(아래의 주소 참조)을 인자값으로 넘기면 된다.

```
dependencies {
    runtime files('libs/somthing.jar', 'libs/nothing.jar')
    runtime fileTree(dir: 'libs', include: '*.jar')
}
```

- <http://www.gradle.org/docs/current/javadoc/org/gradle/api/file/FileCollection.html> 참조.

- 파일이 원래 존재하는 것이 아니라 task 를 통해 생성되는 경우, 파일 의존성에 해당하는 파일들을 어느 task 에서 생성하는지도 명시할 수 있다

```
dependencies {
    compile files("$buildDir/classes") {
        builtBy 'compleie'
    }
}

task compile << {
    println 'compiling classes'
}

task list(dependsOn: configurations.compile) << {
    println "classpath = ${configurations.compile.collect {File file -> file.name}}"
}
```

- 이와 같이 "builtBy"부분의 "compile"로 지정하여 compile task 에 의해 클래스 파일들이 생성된다

◆ Gradle API 의존성

- Gradle task 나 plugin 을 만들 경우에 현재 Gradle API 에 대해 의존성을 지정할 수 있다.

- [http://www.gradle.org/docs/current/dsl/org.gradle.api.artifacts.dsl.DependencyHandler.html#org.gradle.api.artifacts.dsl.DependencyHandler:gradleApi\(\)](http://www.gradle.org/docs/current/dsl/org.gradle.api.artifacts.dsl.DependencyHandler.html#org.gradle.api.artifacts.dsl.DependencyHandler:gradleApi()) 참조

◆ 이행성 의존성 제외하기

- 이행적 의존성 중에서는 일부는 제외하도록 설정할 수 있다.

1. 특정 구성(configuration)에서 이행적 의존성을 제거하면 의존성 구성을 분석하거나

해당 구성을 상속할 때 그 의존성은 제외된다

2. 모든 구성에서 제외시킬 때는 "all*"로 Groovy의 spread-dot 연산자를 사용한다.
3. 제외할 의존성 지정시에는 의존성의 이름만 지정(module: '이름')하거나 그룹 이름(group: '그룹이름')만 지정하거나 혹은 둘 다 함께 지정할 수 있다.
4. 모든 이행적 의존성이 제외 가능한 것은 아니다. 없어도 되는 의존성인지 주의 깊게 살펴보고 지정해야 한다.
5. 의존성 제외가 필요한 경우
 - 의존성 제외가 아닌 다른 방식으로 해결 가능한지 항상 고려한다
 - 라이선스 때문에 해당 모듈을 빼야한다.
 - 어떠한 원격 리포지토리에서도 받아 올 수 없는 모듈이다
 - 실행 시점에는 필요 없는 모듈이다
 - 의존성에 지정된 버전이 다른 버전과 충돌한다. 이때는 버전 충돌 부분으로 해결하도록 한다
6. 대부분의 경우 의존성 제외는 구성 단위로 해야 한다. 그래야 더 명시적이다
7. 의존성 단위 제외의 경우 구성의 다른 의존성에서 제외했던 모듈을 다시 의존할 경우 무용지물이 된다.
8. <http://www.gradle.org/docs/current/javadoc/org/gradle/api/artifacts/ModuleDependency.html> 을 참고한다.
9. 구성 단위로 제외할 시 아래와 같은 형식으로 정의한다.

```
configurations {
    compile.exclude module: 'commons' // compile configurations에서
    특정 모듈 제외
    all*.exclude group: 'org.gradle.test.excludes', module:
    'reports' // 모든 configuration에서 특정 모듈 제외
}
```

10. 의존성 단위로 제외할 시 아래와 같은 형식으로 정의 한다.

```
dependencies {
    compile("org.gradle.test.excludes:api:1.0") {
        exclude module: 'shared' //특정 의존성에 대해서만 모듈 제외.
        exclude group: 'grdoupId', module: 'artifactId'
    }
}
```

11. 혹은 이런 식으로 제외 할 수도 있다.

```
[configurations.runtime, configurations.default]*.exclude(module:
'common-logging')
```

◆ 선택적 속성들

- Gradle 의 의존성은 여러가지 구성(configurations)을 가질 수 있다.
- 지정하지 않으면 기본 구성을 사용한다.
- Maven 에는 기본 구성밖에 없다.
- Ivy 에는 의존성에 여러 구성을 둘 수 있다.
- Gradle 의 서브 프로젝트에 대한 의존성을 지정할 때는 다음과 같이한다.

```
dependencies {
    compile project(path: ':api', configuration: 'spi')
}
```

◆ 선택적 속성들

```
configurations {
    all*.exclude group: 'xml-apis', module: 'xmlParserAPIs'
}

configurations {
    all.collect { configuration ->
        configuration.exclude group: 'xml-apis', module:
'xmlParserAPIs'
    }
}
```

◆ 선택적 속성들

```
[compileJava, compileTestJava]*.options*.encoding = 'UTF-8'
```

6.5 의존성 구성(dependency configurations)

◆ jar 이름에 버전 붙이기

- 이행적 의존성 중에서는 일부는 제외하도록 설정할 수 있다.

◆ 이행적 의존성 관리 사용

- 이행적 의존성 관리를 사용하지 않으면 최상위 의존성을 삭제할 경우 그것이 의존하는 다른 라이브러리가 무엇인지 몰라서 불필요한 jar 가 계속 쌓이게 된다.

- Gradle 은 Maven/Ivy 가 아니더라도 일반 jar 파일에 대한 의존성도 지원한다.

◆ 버전 충돌

- 동일한 jar 의 서로 다른 버전 충돌은 정확히 찾아내어 해결해야 한다.
- 이행적 의존성 관리를 하지 않으면 버전 충돌을 알아내기 힘들다.
- 서로 다른 의존성은 서로 다른 버전의 다른 라이브러리에 의존하기 마련이고, 이 경우 버전 충돌이 일어난다.
- Gradle 이 제공하는 충돌 방지 전략 :
 - 최신 우선 : 가장 최신 의존성이 기본적으로 사용된다.
 - 빠른 실패 : 버전 충돌이 일어나면 빠른 시간안에 실패한다. 이렇게 되면 개발자 스스로 충돌을 제어 할 수 있게 된다.

◆ 버전 충돌 해결 방법

- 충돌이 발생하는 라이브러리를 최상위 의존성으로 버전을 명시하여 강제(forced)지정한다.
- 아무 의존성(이행적이든 아니든)을 강제로 지정한다.

◆ 동적 버전(Dynamic Version)결정과 변하는 모듈(Changing Module)

- 때로는 특정 의존 라이브러리에 대해 항상 최신 버전을 사용하거나, 혹은 특정 version(2.x)최신을 사용하고 싶을 경우가 있다. 동적 버전을 통해 사용 가능하다.
- 특정 version 대를 사용하고 싶을 경우는 버전 명시하는 곳에 4.+와 같이 쓰도록 한다.
- 사용 가능한 최신 버전을 쓰고 싶을 경우는 latest.integration 을, release 된 버전의 최신을 쓸 경우 latest.release 를 쓰도록 한다.
- 동적 버전은 실제 버전이 변경되고, 변하는 모듈의 버전은 그대로이지만 그 내용물(jar)이 계속해서 변경 될 수 있다.
- 기본적으로 동작 버전과 변하는 모듈은 24 시간 캐시된다. 설정을 통해 바꿀 수 있다.
- 특정 라이브러리의 변경을 매번 검사해야 한다면 changing = true 옵션을 추가한다.
- -SNAPSHOT 버전은 기본으로 changing=true 설정이 된다. 단, Maven Repository 일

때만 그렇다.(Maven 자체의 기본정책인듯 하다). 그러므로 Ivy Repository 는
SNAPSHOT 이라도 `changing = true` 설정이 필요하다.

```
dependencies {  
    compile 'junit:junit:4,+'  
}  
  
dependencies {  
    compile 'junit:junit:latest.integration'  
    compile 'junit:junit:latest.release'  
}  
  
dependencies {  
    compile ('junit:junit:latest.4.+') { changing = true }  
}
```

◆ ResolutionStrategy

- <http://www.gradle.org/docs/current/dsl/org.gradle.api.artifacts.ResolutionStrategy.html> 참조.
- 조사 결과 `failOnVersionConflict()`를 설정하고 의존성을 관리하는 것이 좋다고 한다. 그 이유는 이행성에 의해 버전이 변하는 것을 방지할 수 있기 때문이다.
- `failOnVersionConflict()` -> 동일 모듈에 대한 버전 충돌시 즉시 오류 발생하고 실패
- `force 'something:1.x.x', 'something:2.x.x'` -> 특정 모듈의 버전을 강제 지정(최상위건 이행적 의존서이건 상관없이 지정)
- `forceModules = ['something:3.x.x']` -> 이미 강제 지정된 모듈 버전을 대체함
- `cacheDynamicVersionFor 10, 'minutes'` -> 동적 버전 탐색을 10 분 캐시함.
- `cacheChangingModulesFor 0, 'seconds'` -> 변하는 모듈(changing Module)을 캐시하지 않음.

◆ 스크립트 예시

- 먼저 아래의 내용과 같이 `bulid.gradle` 스크립트를 작성한다

```
apply plugin: 'java'  
apply plugin: 'war'  
apply plugin: 'eclipse' // eclipse에 import 시키기위함  
apply plugin: 'eclipse-wtp' // web project로 import하기 위함  
  
sourceCompatibility = 1.8 // jdk version  
version = '1.0' // personality version  
  
springVersion = '3.2.7.RELEASE'  
logBackVersion = '1.1.1'
```

```

// jar 파일의 manifest에 attributes들을 추가
jar {
    manifest {
        attributes 'Implementation-Title': 'springmvc-hellogradle', 'Implementation-Version': version
    }
}

// mavenCentral의 repo를 사용
// 사용할 repo 선언
repositories {
    mavenCentral()
}

// 총 네가지의 lib과 세가지의 테스트 컴파일 라이브러리를 추가
dependencies {
    compile "org.springframework:spring-webmvc:$springVersion"
    compile "cglib:cglib-nodep:3.1"
    compile "ch.qos.logback-classic:1.1.1"
    compile "org.aspectj:aspectjweaver:1.7.4"

    providedCompile 'javax.servlet:javax.servlet-api:3.0.1'

    testCompile "org.springframework:spring-test:$springVersion"
    'junit:junit:4.+'
    'org.mockito:mockito-core:1.9.5'
}

// logback(slf4j)를 사용하기 때문에 모든 의존성에서 commons-logging는 제외
[configurations.runtime, configurations.default]*.exclude( module: "commons-logging" )

// JAVA 컴파일시 인코딩 설정
[compileJava, compileTestJava]*.options*.encoding = 'UTF-8'

// 테스트시 실행할 property설정
test {
    jvmArgs = ["-ea" , "-Xmx256m" ]
    logging.captureStandardOutput(LogLevel.INFO)
    testReport = false
}

// 웹프로젝트를 위한 폴더 생성
// src/main/java, src/test/java
// src/main/resource, src/test/resource
// src/main/webApp, src/main/webapp/WEB-INF
task initProject(description: "initialize project") << {
    createDir = {
        println "create source directory: $it"
        it.mkdirs()
    }
    sourceSets*.java.srcDirs*.each createDir
    sourceSets*.resources.srcDirs*.each createDir
    [webAppDir, new File(webAppDir, "/WEB-INF" ), new File(webAppDir, "/WEB-INF/spring" ), new File(webAppDir, "/WEB-INF/views" ), new File(webAppDir, "/META-INF" )].each createDir
}

```

- 터미널에서 "gradle check" 명령어 혹은 eclipse 를 통해 빌드 스크립트가 정상적으로 작성 된 것 인지를 확인한다. 아래의 이미지와 비슷하게 나온다면 정상적으로 작성 된 것이다.

```
<terminated> Gradle_sample check [Gradle Build] Gradle Build on Gradle_sample
[sts] -----
[sts] Starting Gradle build for the following tasks:
[sts]     check
[sts] -----
Creating properties on demand (a.k.a. dynamic properties) has been deprecated and is scheduled to be removed in Gradle 2.0. Please read http://gradle.org/docs/current/dsl/org.gradle.api.plugins.ExtraPropertiesExtension.html
Deprecated dynamic property: "springVersion" on "root project 'Gradle_sample'", value: "3.2.7.RELEASE".
Deprecated dynamic property: "logbackVersion" on "root project 'Gradle_sample'", value: "1.1.1".
The Test.testReport property has been deprecated and is scheduled to be removed in Gradle 2.0. Please use the Test.getReports().getHtml().setEnabled() property instead.
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:compileTestJava UP-TO-DATE
:processTestResources UP-TO-DATE
:testClasses UP-TO-DATE
:test UP-TO-DATE
:check UP-TO-DATE
BUILD SUCCESSFUL

Total time: 0.305 secs
[sts] -----
[sts] Build finished successfully!
[sts] Time taken: 0 min, 0 sec
[sts] -----
```

- "gradle initProject" 명령어를 통해 소스 디렉토리를 생성한다.
- "gradle eclipse" 명령어를 통해 이클립스에 import 할 수 있도록 빌드한다.
- 빌드 성공 후, eclipse 에서 import 하여 사용한다.

7 build.gradle 스크립트 작성

- 테스트 소스셋에서 모든 단위 테스트를 자동으로 감지하여 실행한다.
- 테스트 수행이 끝나면 보고서를 생성한다.
- JUnit, TestNG를 지원한다.
- test 관련 참조 문서 <http://www.gradle.org/docs/current/dsl/org.gradle.api.tasks.testing.Test.html>

7.1 테스트 실행

- ◆ 테스트는 독립 JVM 에서 격리상태로 실행된다.
- ◆ test.debug 프로퍼티를 true 로 설정하면 디버그모드로 실행되며 5005 포트로 디버깅 할 수 있다.
- ◆ 병렬 테스트
 - 여러 테스트 프로세서를 동시에 실행 할 수 있다.
 - "maxParallelForks" 프로퍼티로 테스트 프로세스 갯수를 설정할 수 있다. 기본값은

1 이다.(병렬 테스트 안함)

- 테스트 프로세스는 `org.gradle.test.worker` 시스템 프로퍼티를 설정한다
- ◆ “forkEvery” 프로퍼티로 몇 개의 테스트를 수행한 뒤에 프로세스를 재 시작 할지를 정할 수 있다. 단위 테스트가 JVM Heap 을 너무 많이 소모할 경우 이 값을 작게 준다. 기본은 재 시작 안 함
- ◆ “ignoreFailures” 프로퍼티는 테스트 실패 시 행위를 정의한다. 기본값은 false 이며 테스트가 실패하면 즉시 멈춘다. true 일 경우 테스트가 실패해도 멈추지 않고 다음으로 넘어간다.
- ◆ “testLogging” 프로퍼티는 테스트의 로깅 레벨을 설정한다. 기본적으로 모든 실패한 테스트에 대한 요약 메시지를 보여준다. [TestLoggingContainer](#) 참조.
- ◆ 표준 출력/에러를 화면에 표시하려면 “testLogging.showStandardStreams = true” 설정 필요

7.2 System properties

- ◆ 시스템 프로퍼티는 “gradle -D 프로퍼티이름 = 값” 형태로 지정한다.
- ◆ “taskName.single=testNamePattern” 형태로 지정하면 “testNamePattern”에 일치하는 테스트만 실행된다
- ◆ “taskName”은 멀티프로젝트 패스 형태(sub1:sub2:test)로 기술하거나 그냥 태스크 이름만 기술해도 된다
- ◆ “testNamePattern”은 “*/testNamePattern*.class” 형태로 기술한다.
- ◆ 패턴은 각 서브 프로젝트에 적용된다. 특정 서브 프로젝트에서 패턴에 매칭되는 테스트가 없으면 예외가 발생한다. 이 경우 패턴에 서브 프로젝트를 명시 할 수 있다.

```
gradle -Dtest.single=ThisUniquelyNamedTest test
gradle -Dtest.single=a/b/ test
gradle -DintegTest.single=*IntegrationTest integTest
gradle -Dtest.single=:proj1:test:Customer build
gradle -DintegTest.single=c/d/ :proj1:integTest
```

7.2 테스트 감지

- ◆ Test task 는 컴파일 된 클래스를 분석하여 테스트 클래스를 감지한다. 기본적으로 모든

*.class 파일을 분석한다.

- ◆ 추상 클래스는 실행 안한다.
- ◆ 상위 클래스까지 모두 분석한다.
- ◆ "scanForTestClasses"를 "false"로 하면 자동감지를 수행하지 않는다. 이 경우 명시적으로 포함/제외 시킨 클래스만 실행한다.
- ◆ "scanForTestClasses=false"이면서 포함/제외 클래스를 명시하지 않으면 기본적으로 "**/*Tests.class"와 "**/*Test.class"를 실행하고, "**/*Abstract*.class"는 제외한다.

7.3 JUnit

- ◆ JUnit 3, 4 의 테스트 클래스.
- ◆ TestCase, GroovyTestCase 상속
- ◆ "@RunWith" 어노테이션 적용
- ◆ "@Test" 어노테이션을 가진 메소드가 있는 클래스

7.4 TestNG

- ◆ "@Test" 어노테이션을 가진 메소드가 있는 클래스

7.5 테스트의 분리

- ◆ "*Test"와 "*IntegrationTest"를 분리해서 실행하고자 하는 경우가 있을 수 있다.
- ◆ [Gradle goodness - Running Single Test 참조](#)
- ◆ "include"와 "exclude"를 사용하고 [Test](#)를 상속하는 또 다른 태스크를 만들어 지정한다.

```
test {
    exclude '**/*IntegrationTest.class'
}

task integrationTest(Type: Test, dependsOn: testClasses) {
    description = 'Integration test'
    group = 'verification'

    include '**/*IntegrationTest.class'
    testReportDir file("${buildDir}/reports/integration-test")
}

task.withType(Test) {
    // Test 들의 공통 설정
    useJUnit()
    maxHeapSize '2048m'
```

```
jvmArg '-XX:MaxPermSize=256m'

testLoggin {
    events 'started', 'passed'
}
}
```

7.6 TestLogging

◆ [TestLogginContainer](#) 참고

```
test {
    testLogging {
        events "failed"
        exceptionFormat "short"
        showStandardStreams true

        debug {
            events "started", "skipped", "failed"
            exceptionFormat "full"
        }
    }
}
```

8 Gradle Life Cycle

◆ Gradle 빌드 툴은 기본적인 라이프사이클을 제공하지 않는다. 이 점이 메이븐 빌드 툴과의 가장 큰 차이점이다. 메이븐 빌드 툴은 빌드 설정 파일이 상속 개념이다. 따라서 부모(parent) 설정 파일에서 제공하는 기본 라이프사이클을 그대로 상속받는 구조이다. 객체 지향 프로그래밍에서의 상속과 똑같은 개념이다. 상속이 좋은 점도 많지만 부모의 속성과 행위를 규정함으로써 강제하는 부분이 많다. 따라서 객체 지향 프로그래밍에서도 상속보다는 구성(composition)을 사용하는 유연한 설계를 할 수 있다고 안내하고 있다.

◆ Gradle 빌드 툴은 메이븐이 가지는 빌드 스크립트의 상속 문제를 해결하기 위해 구성을

통한 확장을 선택했다. 이 구성을 통한 확장의 핵심에 plugin이 있다. Gradle의 기본 설정에는 빌드 라이프사이클이 존재하지 않는다. 하지만 "apply plugin: 'java'" 와 같이 java plugin을 사용하도록 설정하는 순간 자바 프로젝트에 대한 빌드 라이프사이클이 추가된다.

◆ 자바 프로젝트를 빌드하는 기본 과정을 유추해보면 다음과 같다.

8.1.1 production java 코드(src/main/java 소스 코드)를 컴파일 한다.

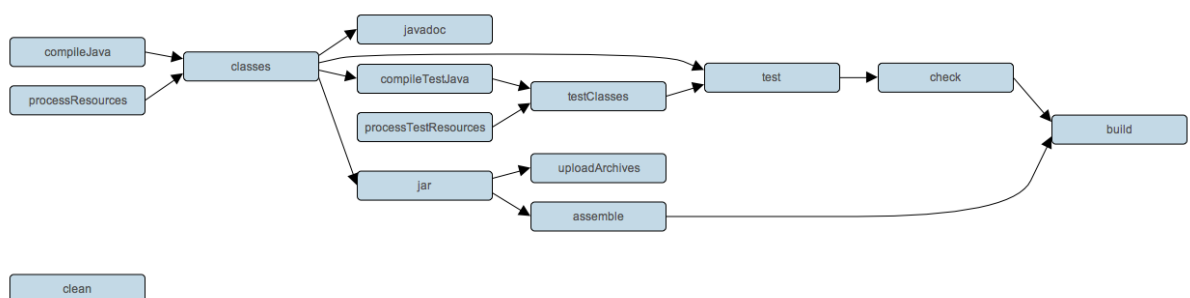
8.1.2 production resources(src/main/resources 자원)을 컴파일 output 디렉토리에 복사한다.

8.1.3 test java 코드(src/test/java 소스 코드)를 컴파일 한다.

8.1.4 test resources(src/test/resources 자원)을 test 코드 컴파일 output 디렉토리에 복사한다.

8.1.5 test 소스 코드를 실행해 테스트를 진행한다.

◆ 자바 프로젝트를 컴파일하고 테스트하는 과정을 살펴보면 위와 같다. Gradle 빌드 스크립트에 java plugin을 추가하면 자바 프로젝트를 빌드하기 위한 여러 개의 task가 추가된다. 여러 개의 task는 서로 간의 의존관계를 맺으면서 자바 프로젝트에 대한 빌드 라이프사이클을 추가한다. java plugin task간의 의존관계를 살펴보면 다음과 같다



9 부록 Profile 흉내내기

-Pprofile=값 형태의 옵션으로 Maven의 Profile을 흉내낼 수 있다.

```

final String DEFAULT_PROFILE = 'development'
allprojects {
    if (!project.hasProperty('profile') || !profile) {
        ext.profile = DEFAULT_PROFILE
    }
}
  
```

```
// 리소스에 각 프로파일별 리소스 디렉토리 추가
sourceSets {
    main {
        resources {
            srcDir "src/main/resources-${profile}"
        }
    }
}
}
```

- 이제 `gradle -Pprofile=production` 형태로 호출하면 모든 프로젝트에서 `profile` 속성에 `production` 이라는 값이 지정된 상태가 된다. 이에 따라 가변적인 행동을 정의해 준다.
- `-Pprofile` 을 생략하면 기본인 `DEFAULT_PROFILE` 상수의 값 `development` 로 작동하게 된다.

10 부록 Gradle Multi Project

1. 최상위 프로젝트의 이름

`settings.gradle` 파일에서 다음과 같이 최상위 프로젝트 이름을 지정한다. 이는 해당 프로젝트 디렉토리 이름과 무관하게 설정된다.

```
rootProject.name = '프로젝트이름'
```

2. 멀티 프로젝트의 기본

- ◆ 최상위 프로젝트에 `setting.gradle`이 필요하다. 여기서 하위 프로젝트를 `include`해준다.

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

- ◆ 최상위프로젝트의 `build.gradle`에 모든 서브 프로젝트에 공통된 설정을 넣는다.

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

    repositories {
        mavenCentral()
    }

    dependencies {
        testCompile 'junit:junit:4.8.2'
    }

    version = '1.0'

    jar {
```

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

- ◆ 모든 서브 프로젝트에 java, eclipse-wtp 플러그인이 적용되고, 지정된 리포지토리와 의존성이 무조건 추가된다

2.1 서브 프로젝트간 의존성

```
// 어느 서브 프로젝트의 build.gradle
dependencies {
    compile project(':shared')
}
// shared 서브 프로젝트에 의존하고 있다.
```

3. 교차 프로젝트 구성 Cross Project Configuration

3.1 공통 행위 정의하기

- ◆ 다음과 같은 구조의 프로젝트가 있다고 하자. water가 부모 프로젝트이다.

```
water/
  build.gradle
  settings.gradle
  bluewhale/
```

- ◆ settings.gradle

```
include 'bluewhale'
```

- ◆ 서브 프로젝트의 빌드 파일은 없어도 상관없다. 부모 프로젝트에서 서브 프로젝트의 행위를 정의하는 것도 가능하다.

```
Closure cl = { task -> println "I'm $task.project.name" }
task hello << cl
project(':bluewhale') {
    task hello << cl
}
```

- ◆ 위를 gradle -q hello로 실행하면

```
> gradle -q hello
I'm water
I'm bluewhale
```

- ◆ Gradle에서는 어떠한 빌드 스크립트에서라도 멀티 프로젝트의 아무 프로젝트에나 접근할 수 있다. Project API에는 `project()`라는 메소드가 있으며, 이는 프로젝트의 경로를 인자로 받아서 해당 경로의 Project 객체를 리턴한다. 이러한 방법을 교차 프로젝트 구성 cross project configuration이라고 부른다.

- ◆ krill 서브 디렉토리를 만들어서 krill 서브 프로젝트를 선언한다. settings.gradle

```
include 'bluewhale', 'krill'
```

- ◆ 모든 프로젝트에 적용되는 태스크를 선언한다.

```
allprojects {  
    task hello << { task -> println "I'm $task.project.name" }  
}
```

- ◆ 실행하면

```
> gradle -q hello  
I'm water  
I'm bluewhale  
I'm krill
```

- ◆ Project API의 `allprojects` 프라퍼티는 현재 프로젝트와 그것의 모든 서브 프로젝트를 리턴한다. `allprojects`에 클로저를 인자로 주면 클로저의 구문이 `allprojects`의 프로젝트들로 위임된다. `allprojects.each`로 이터레이션을 도는 것도 가능하다.

- ◆ Gradle은 기본적으로 구성 주입(configuration injection)을 사용한다

- ◆ 또한 다른 빌드 툴 처럼 프로젝트 상속 구조도 가능하다

4. 서브 프로젝트 구성

`Project.subprojects`로 서브 프로젝트들만 접근하는 것도 가능하다. `allprojects`는 부모 프로젝트까지 포함한 것이다.

4.1 공통 행위 정의

- ◆ 서브 프로젝트에만 적용되는 공통 행위 정의하기

```
allprojects {  
    task hello << {task -> println "I'm $task.project.name" }  
}  
// 아래는 서브프로젝트에만 적용된다.  
subprojects {  
    hello << {println "- I depend on water"}  
}
```

- ◆ 실행하면

```
> gradle -q hello
I'm water
I'm bluewhale
- I depend on water
I'm krill
- I depend on water
```

4.2 특정 서브 프로젝트에만 행위 추가

- ◆ 일반적으로는 서브 프로젝트의 빌드 파일에 해당 프로젝트에 국한된 행위를 기술한다. 하지만 특정 프로젝트에 국한된 행위를 부모 프로젝트 빌드 파일에 정의할 수도 있다.

```
allprojects {
    task hello << {task -> println "I'm $task.project.name" }
}
subprojects {
    hello << {println "- I depend on water"}
}
project(':bluewhale').hello << {
    println "- I'm the largest animal that has ever lived on this planet."
}
```

- ◆ 실행하면.

```
> gradle -q hello
I'm water
I'm bluewhale
- I depend on water
- I'm the largest animal that has ever lived on this planet.
I'm krill
- I depend on water
```

- ◆ 서브 프로젝트의 디렉토리 최상단에 build.gradle을 두고 거기에 행위를 추가할 수 있다.

```
water/
  build.gradle
  settings.gradle
bluewhale/
  build.gradle
krill/
  build.gradle
```

- ◆ bluewhale/build.gradle

```
hello.doLast { println "- I'm the largest animal that has ever lived on this planet." }
```

- ◆ krill/build.gradle

```
hello.doLast {  
    println "- The weight of my species in summer is twice as heavy as all  
human beings."  
}
```

◆ build.gradle

```
allprojects {  
    task hello << {task -> println "I'm $task.project.name" }  
}  
subprojects {  
    hello << {println "- I depend on water"}  
}
```

◆ 실행하면

```
> gradle -q hello  
I'm water  
I'm bluewhale  
- I depend on water  
- I'm the largest animal that has ever lived on this planet.  
I'm krill  
- I depend on water  
- The weight of my species in summer is twice as heavy as all human beings.
```

4.3 프로젝트 필터링

tropicalFish라는 프로젝트를 추가하고 water 프로젝트 빌드 파일에 행위를 더 추가해보자.

4.4 이름으로 필터링

◆ 변경된 프로젝트 레이아웃

```
water/  
  build.gradle  
  settings.gradle  
  bluewhale/  
    build.gradle  
  krill/  
    build.gradle  
  tropicalFish/
```

◆ setting.gradle

```
include 'bluewhale', 'krill', 'tropicalFish'
```

◆ build.gradle

```
allprojects {
```



```

    task hello << {task -> println "I'm $task.project.name" }
}
subprojects {
    hello << {println "- I depend on water"}
}
// 이름이 tropicalFish가 아닌 프로젝트만 찾아서 설정
configure(subprojects.findAll { it.name != 'torpicalFish' }) {
    hello << {println '- I love to spend time in the arctic waters.'}
}

```

◆ 실행하면

```

I'm water
I'm bluewhale
- I depend on water
- I love to spend time in the arctic waters.
- I'm the largest animal that has ever lived on this planet.
I'm krill
- I depend on water
- I love to spend time in the arctic waters.
- The weight of my species in summer is twice as heavy as all human beings.
I'm tropicalFish
- I depend on water

```

- [Project.configure\(\)](#) 메소드는 리스트를 인자로 받아서 리스트 안의 프로젝트에 구성을 추가한다.

4.4 프라퍼티로 필터링하기

ext 프라퍼티를 통해 필터링이 가능하다.

◆ 프로젝트 레이아웃

```

water/
  build.gradle
  settings.gradle
bluewhale/
  build.gradle
krill/
  build.gradle
tropicalFish/
  build.gradle

```

◆ bluewhale/build.gradle

```

ext.arctic = true
hello.doLast { println "- I'm the largest animal that has ever lived on this
planet." }

```

◆ krill/build.gradle

```
ext.arctic = true
hello.doLast {
    println "- The weight of my species in summer is twice as heavy as all
human beings."
}
```

◆ tropicalFish/build.gradle

```
ext.arctic = false
```

◆ build.gradle

```
allprojects {
    task hello << { task -> println "I'm $task.project.name" }
}

subprojects {
    hello {
        doLast { println "- I depend on water" }
        afterEvaluate { Project project ->
            if (project.arctic) {
                doLast {
                    println '- I love to spend time in the arctic waters.'
                }
            }
        }
    }
}
```

◆ 실행하면

```
> gradle -q hello
I'm water
I'm bluewhale
- I depend on water
- I'm the largest animal that has ever lived on this planet.
- I love to spend time in the arctic waters.
I'm krill
- I depend on water
- The weight of my species in summer is twice as heavy as all human beings.
- I love to spend time in the arctic waters.
I'm tropicalFish
- I depend on water
```

- ◆ afterEvaluate는 서브 프로젝트의 빌드 스크립트를 모두 수행한 뒤에 인자로 넘어온 클로저를 실행하라는 의미이다. arctic 프라퍼티가 서브 프로젝트 빌드 스크립트에 선언 돼 있기 때문이다.

5. 멀티 프로젝트 빌드 실행 규칙

- 최상위 프로젝트에서 hello 태스크를 실행하면 최상위와 그 아래 모든 서브 프로젝트의 hello 태스크가 실행 된다.
- bluewhale 디렉토리로 이동해서 hello 태스크를 실행하면 bluewhale 프로젝트의 태스크만 실행된다.
- Gradle 의 태스크 실행
 - 현재 디렉토리에서 시작하여 계층구조를 탐색하여 hello 라는 이름의 태스크를 찾고 실행한다.
 - Gradle 은 항상 모든 프로젝트를 평가하고, 존재하는 모든 태스크 객체를 생성한다
 - 그리고서 태스크의 이름과 현재 디렉토리를 기반으로 실행해야할 태스크를 결정한다.
 - Gradle 의 교차 프로젝트 구성 때문에 모든 프로젝트는 어떠한 태스크를 실행할 때는 그 전에 먼저 평가 되어 한다.

◆ Bluewhale/build.gradle

```
ext.arctic = true
hello << { println "- I'm the largest animal that has ever lived on this planet." }

task distanceToIceberg << {
    println '20 nautical miles'
}
```

◆ krill/build.gradle

```
ext.arctic = true
hello << { println "- The weight of my species in summer is twice as heavy as all human beings." }

task distanceToIceberg << {
    println '5 nautical miles'
}
```

◆ 최상위 프로젝트에서 실행하면

```
> gradle distanceToIceberg
:bluewhale:distanceToIceberg
20 nautical miles
:krill:distanceToIceberg
5 nautical miles

BUILD SUCCESSFUL

Total time: 1 secs
```

- ◆ 최상위 water 프로젝트에서 실행한다. water와 tropicalFish는 distanceToIceberg 태스크가 없지만 상관없다. 왜냐면 계층 구조를 따라 내려가면서 해당 명칭의 태스크를 실행한다라는 규칙 때문이다.

6. 절대 경로로 태스크 실행하기

◆ tropicalFish에서 실행한 `gradle -q :hello :krill:hello hello`

```
> gradle -q :hello :krill:hello hello
I'm water
I'm krill
- I depend on water
- The weight of my species in summer is twice as heavy as all human beings.
- I love to spend time in the arctic waters.
I'm tropicalFish
- I depend on water
```

◆ water의 `:hello`, krill의 `hello`, tropicalFish의 `hello` 순서로 실행된다.

7. 프로젝트와 태스크의 경로

◆ 프로젝트 경로의 패턴은 다음과 같다

- 항상 콜론(:)으로 시작한다. 이는 최상위 프로젝트를 의미한다.
- 최상위 프로젝트만 이름 없이 사용된다.
- `:bluewhale` 은 파일 시스템상에서 `water/bluewhale`을 뜻한다

◆ 태스크의 경로는 프로젝트 경로에 태스크 이름을 붙인 것이다

- `:bluewhale:hello`는 bluewhale 프로젝트의 `hello` 태스크
- 프로젝트 안에서는 태스크 이름만 사용하면 해당 프로젝트의 태스크로 간주한다. 상대 경로로 해석하기 때문이다.

8. 의존성 – 어느 의존성을 선택?

의존성과 실행 순서에 대해서 확인 해보자.

8.1 의존성과 실행 순서

◆ 프로젝트 레이아웃

```
messages/
  settings.gradle
  consumer/
    build.gradle
  producer/
    build.gradle
```

◆ Settings.gradle

```
include 'consumer', 'producer'
```

◆ consumer/build.gradle

```
task action << {
    println("Consuming message: " +
        (rootProject.hasProperty('producerMessage') ?
        rootProject.producerMessage : 'null'))
}
```

◆ producer/build.gradle

```
task action << {
    println "Producing message:"
    rootProject.producerMessage = 'Watch the order of execution.'
}
```

◆ 실행하면

```
> gradle -q action
Consuming message: null
Producing message:
```

◆ 이것은 작동하지 않는다. 왜냐면 명시적으로 정의하지 않으면 Gradle은 **알파벳 순서에 따라** 태스크를 실행하기 때문이다.

◆ 따라서 :consumer:action이 :producer:action 보다 먼저 실행된다.

◆ producer 프로젝트를 aProducer로 바꾸면 원하는 대로 작동한다

◆ aProducer로 바뀐 상태에서 consumer 디렉토리에서 action 태스크를 실행하면 규칙에 따라 :aProducer:action은 실행이 안되므로null이 찍힌다

8.2 태스크 실행 의존성 선언하기

◆ consumer/build.gradle

```
task action(dependsOn: ':producer:action') << {
    println("Consuming message: " +
        (rootProject.hasProperty('producerMessage') ?
        rootProject.producerMessage : 'null'))
}
```

◆ 최상위와 consumer 디렉토리 어디에서든 실행하면

```
> gradle -q action
Producing message:
```

Consuming message: Watch the order of execution.

- ◆ :consumer:action이 :producer:action에 실행시 의존성을 걸고 있기 때문에 항상 :producer:action이 먼저 실행된다

8.3 교차 프로젝트 태스크 의존성의 특징

- ◆ 의존성을 지정할 때 태스크 이름은 아무 상관이 없다

8.4 구성 시(Configuration Time) 의존성 설정하기

- ◆ 태스크에 의존성을 거는 것이 아니라 프로젝트 구성에 의존해야 할 경우가 있다

- ◆ consumer/build.gradle

```
message = rootProject.hasProperty('producerMessage') ?
rootProject.producerMessage : 'null'

task consume << {
    println("Consuming message: " + message)
}
```

- ◆ producer/build.gradle

```
rootProject.producerMessage = 'Watch the order of evaluation.'
```

- ◆ 실행하면

```
> gradle -q consume
Consuming message: null
```

- ◆ 기본 빌드 파일 평가 순서가 알파벳 순서이기 때문에 consumer가 producer보다 먼저 평가된다

- ◆ 해결하려면 consumer/build.gradle

```
evaluationDependsOn(':producer')

message = rootProject.hasProperty('producerMessage') ?
rootProject.producerMessage : 'null'

task consume << {
    println("Consuming message: " + message)
}
```

◆ 실행하면

```
> gradle -q consume
Consuming message: Watch the order of evaluation.
```

◆ 기본 빌드 파일 평가 순서가 **알파벳 순서**이기 때문에 consumer가 producer보다 먼저 평가된다

◆ 해결하려면 consumer/build.gradle

```
evaluationDependsOn(':producer')

message = rootProject.hasProperty('producerMessage') ?
rootProject.producerMessage : 'null'

task consume << {
    println("Consuming message: " + message)
}
```

◆ 실행하면

```
> gradle -q consume
Consuming message: Watch the order of evaluation.
```

◆ evaluationDependsOn은 producer를 consumer보다 먼저 평가하게 만든다

◆ 사실 위의 경우는 억지스럽다. 사실은 그냥 rootProject.producerMessage 값을 바로 읽게 만들기만 해도 된다.

◆ consumer/build.gradle

```
task consume << {
    println("Consuming message: " +
        (rootProject.hasProperty('producerMessage') ?
        rootProject.producerMessage : 'null'))
}
```

◆ 구성시 의존성은 실행시 의존성과는 매우 다르다

◆ 구성시 의존성은 프로젝트간에 맺는 것이고, 실행시 의존성은 태스크간의 의존성으로 결정된다

◆ 다른 다른 점은 서브 프로젝트에서 빌드 명령을 내려도 항상 모든 프로젝트의 구성을 수행한다는 점이다

◆ 기본 구성 순서는 위에서 아래로 내려간다

◆ 기본 구성 순서를 아래에서 위로 방향으로 바꾸려면 부모 프로젝트가 자신의 자식 프로젝트에 의존한다는 뜻이 되는데 이 때는evaluationDependsOnChildren() 메소드를 사용한다.

- ◆ 동일 단계의 프로젝트간 구성 순서는 알파벳 순서에 따른다. 가장 일반적인 경우로 모든 서브 프로젝트가 Java 플러그인을 사용하는 것 처럼 공통의 라이프사이클을 공유하는 때가 있다
- ◆ dependsOn을 사용해 서로 다른 두 프로젝트의 실행시 의존성을 지정할 경우 이 메소드는 기본적으로 두 프로젝트간에 구성 의존성을 생성하는 것이다. 따라서 이 때는 구성 의존성을 명시적으로 지정하지 않아도 된다

8.5 실전 예제

두 개의 웹 애플리케이션 서브 프로젝트를 가진 최상위 프로젝트가 웹 애플리케이션 배포본을 생성하는 예를 본다. 예제에서는 단 하나의 교차 프로젝트 구성을 사용한다.

◆ 프로젝트 레이아웃

```
webDist/
  settings.gradle
  build.gradle
  date/
    src/main/java/
      org/gradle/sample/
        DateServlet.java
  hello/
    src/main/java/
      org/gradle/sample/
        HelloServlet.java
```

◆ 프로젝트 레이아웃

```
webDist/
  settings.gradle
  build.gradle
  date/
    src/main/java/
      org/gradle/sample/
        DateServlet.java
  hello/
    src/main/java/
      org/gradle/sample/
        HelloServlet.java
```

◆ settings.gradle

```
include 'date', 'hello'
```

◆ build.gradle

```
allprojects {
    apply plugin: 'java'
    group 'org.gradle.sample'
    version = '1.0'
}
```



```

subprojects {
    apply plugin: 'war'
    repositories {
        mavenCentral()
    }

    dependencies {
        compile "javax.servlet:servlet-api:2.5"
    }
}

task explodedDist(dependsOn: assemble) << {
    File explodedDist = mkdir("$buildDir/explodedDist")

    subprojects.each { project ->
        project.tasks.withType(Jar).each { archiveTask ->
            copy {
                from archiveTask.archivePath
                into explodedDist
            }
        }
    }
}

```

- ◆ 최상위 프로젝트에서 `gradle -q explodedDist`를 실행하면 `"$buildDir/explodedDist"`에 `hello-1.0.jar`와 `date-1.0.jar`가 생성된다
- ◆ `date`와 `hello` 프로젝트는 `webDist` 프로젝트의 구성시 의존성을 가진 상태이다. 그리고 빌드 로직도 `webDist`에서 주입되었다
- ◆ 하지만 실행시 의존성은 `webDist`가 `date`와 `hello`의 빌드된 아티팩트에 의존한다
- ◆ 세번째 의존성으로 `webDist`가 자식인 `date`와 `hello`에 구성시 의존성도 있는데, 이는 `archivePath`를 알아야만 하기 때문이다. 하지만 태스크를 실행하는 시점에 요청한다. 따라서 순환 의존성은 아니다
- ◆ `withType()` 메소드. 컬렉션에서 특정 타입인 것만 골라서 새로운 컬렉션으로 만든다

9. 프로젝트 lib 의 존성

한 프로젝트가 다른 프로젝트의 컴파일 결과와 그 의존하는 라이브러리들 모두에 의존하는 경우가 발생한다. 이 때 프로젝트간 의존성을 설정한다.

- ◆ 프로젝트 레이아웃

```

java/
settings.gradle

```

```

build.gradle
api/
  src/main/java/
    org/gradle/sample/
      api/
        Person.java
      apiImpl/
        PersonImpl.java
  services/personService/
    src/
      main/java/
        org/gradle/sample/services/
          PersonService.java
      test/java/
        org/gradle/sample/services/
          PersonServiceTest.java
  shared/
    src/main/java/
      org/gradle/sample/shared/
        Helper.java

```

- ◆ shared, api, personService 프로젝트가 있다. personService는 다른 두 프로젝트에 의존하고, api는 shared에 의존한다

- ◆ settinga.gradle

```
include 'api', 'shared', 'services:personService'
```

- ◆ build.gradle

```

subprojects {
    apply plugin: 'java'
    group = 'org.gradle.sample'
    version = '1.0'
    repositories {
        mavenCentral()
    }
    dependencies {
        testCompile "junit:junit:4.8.2"
    }
}

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

project(':services:personService') {

```

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

- ◆ lib 의존성은 실행시 의존성의 특별한 형태이다. 의존성이 걸리게 되면 다른 프로젝트가 먼저 빌드하여 jar를 생성하고 그것을 현재 프로젝트의 클래스패스에 추가한다
- ◆ 따라서 api 디렉토리에서 gradle compile을 실행하면 shared가 먼저 빌드 되고 그 뒤에 api가 빌드 된다. 프로젝트 의존성은 부분적인 멀티 프로젝트 빌드를 가능케 한다
- ◆ Ivy 방식의 매우 상세한 의존성 설정도 가능하다
- ◆ build.gradle

```
subprojects {
    apply plugin: 'java'
    group = 'org.gradle.sample'
    version = '1.0'
}

project(':api') {
    configurations {
        spi
    }
    dependencies {
        compile project(':shared')
    }
    task spiJar(type: Jar) {
        baseName = 'api-spi'
        dependsOn classes
        from sourceSets.main.output
        include('org/gradle/sample/api/**')
    }
    artifacts {
        spi spiJar
    }
}

project(':services:personService') {
    dependencies {
        compile project(':shared')
        compile project(path: ':api', configuration: 'spi')
        testCompile "junit:junit:4.8.2", project(':api')
    }
}
```

- ◆ Java 플러그인은 기본적으로 프로젝트당 모든 클래스를 포함한 하나의 jar를 생성한다. 위 예제에서는 api 프로젝트의 인터페이스만 포함하는 추가적인 jar를 생성하였다

9.1 의존하는 프로젝트의 빌드 금지하기

- ◆ 때로는 부분 빌드를 할 때 의존하고 있는 프로젝트를 빌드하지 않기를 바랄 때도 있다. -

a 옵션으로 gradle을 실행하면 된다.

10. 분리된(decoupled) 프로젝트

- ◆ 두 프로젝트간의 프로젝트 모델에 접근하지 않는 것을 서로 분리된(decoupled) 프로젝트라고 부른다
- ◆ 분리된 프로젝트는 프로젝트 의존성이나 태스크 의존성으로만 연결되어 있다
- ◆ 그 외의 어떠한 형태의 프로젝트간 소통행위(다른 프로젝트의 값을 읽거나 수정하는 등)은 두 프로젝트를 엮인(coupled) 프로젝트로 만든다
- ◆ 엮인 프로젝트가 되는 가장 일반적인 상황은 교차 프로젝트 설정에서 구성 주입을 사용할 경우이다
- ◆ allprojects 혹은 subprojects 키워드를 사용한 순간 프로젝트들은 엮인 것이다.

11. 멀티 프로젝트 빌드와 테스트

- ◆ Java 플러그인의 build 태스크를 사용하여 컴파일, 테스트, 코드 스타일 검사(CodeQuality 플러그인 사용시)등을 할 수 있다.
- ◆ 다중 프로젝트에서 여러 범위의 프로젝트에 대해 빌드를 할 경우가 있는데 이 때 buildNeeded 와 buildDependents 태스크를 사용한다.
- ◆ "프로젝트 lib 의존성"의 프로젝트로 테스트 해본다
- ◆ gradle :api:build : api와 api가 의존하는 모든 프로젝트에 대해 컴파일과 jar를 수행하고 api 프로젝트의 build를 수행한다
- ◆ gradle -a :api:build : api 프로젝트의 build 만 수행한다.
- ◆ gradle :api:buildNeeded : api와 api가 의존하는 모든 프로젝트의 build를 수행한다
- ◆ gradle :api:buildDependents : api와 api에 의존하는 모든 프로젝트에 대해 build를 수행한다
- ◆ gradle build : 모든 프로젝트에 대해 build한다

12. 프라퍼티와 메소드 상속

- ◆ 프로젝트에 정의된 프라퍼티와 메소드는 모든 서브 프로젝트로 상속된다
- ◆ 이 때문에 gradle 태스크이름 -P프라퍼티이름=값으로 실행할 경우 모든 project 객체에서 해당 프라퍼티를 사용할 수 있게 된다

13. 멀티 프로젝트 단위 테스트간의 의존성

개인적으로 아래 방법보다는 공통 단위 테스트용 프로젝트를 만들고(예: xxx-test-support) 해당 프로젝트에 각종 테스트용 의존성과 테스트용 유틸리티 클래스를 일반 코드로 작성한 뒤에 다른 프로젝트들이 `testCompile project(':xxx-test-support')` 형태로 의존성을 추가하는 것이 더 일관성 있고 깔끔한 방법으로 보인다