



# Gradle: Réalisez vos builds avec Gradle

---

David THIBAU – 2023

[david.thibau@gmail.com](mailto:david.thibau@gmail.com)



# Agenda

---

- Introduction
  - Objectifs d'un outil de build
  - Gradle vs Ant/Maven
  - Installation, commandes
  - Un bref tour de Groovy
- Concepts cœur
  - Composants d'un projet
  - Manipulation des tâches
  - Hooks du cycle de vie
  - Plugins, le plugin init, le wrapper
- Java et C++
  - Gestion de Dépendances
  - Plugins pour Java
  - Plugins pour C++
- Multi-projets
  - Multi-projets
- Livraison continue
  - Tests
  - Analyse de code
  - Jenkins



# Introduction

---

## **Objectifs d'un outil de build**

Gradle vs Ant/Maven

Installation, Commandes

Un bref tour de Groovy



# Pré-requis d'un build

---

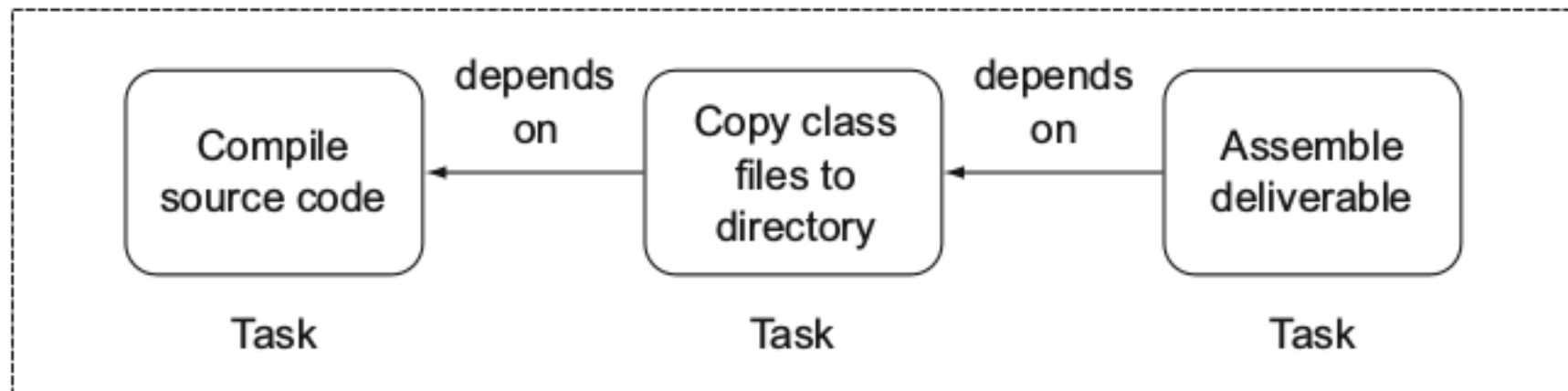
- ♦ **Proscrire les interventions manuelles** sujettes à erreur et chronophage
- ♦ Créer des builds **reproductibles** : Pour tout le monde qui exécute le build
- ♦ **Portable** : Ne doit pas nécessiter un OS ou un IDE particulier, il doit être exécutable en ligne de commande
- ♦ **Sûr** : Confiance dans son exécution



# Graphe de build

Un build est une séquence ordonnée de tâches unitaires.

Les tâches ont des dépendances entre elles qui peuvent être modélisées via un **graphe acyclique dirigé** :





# Composants d'un outil de build

---

**Le fichier de build** : Contient la configuration requises pour le build, les dépendances externes, les instructions pour exécuter un objectif sous forme de tâches inter-dépendantes

**Une tâche unitaire** prend une entrée effectue des traitements et produit une sortie.  
Une tâche dépendante prend comme entrée la sortie d'une autre tâche

**Moteur de build** : Le moteur traite le fichier de build et construit sa représentation interne. Des outils permettent d'accéder à ce modèle via une API

**Gestionnaire de dépendances** : Traite les déclarations de dépendances et les résout par rapport à un dépôt d'artefact contenant des méta-données permettant de trouver les dépendances transitives



# Introduction

---

Objectifs d'un outil de build  
**Gradle vs Ant/Maven**  
Installation, Commandes  
Un bref tour de Groovy



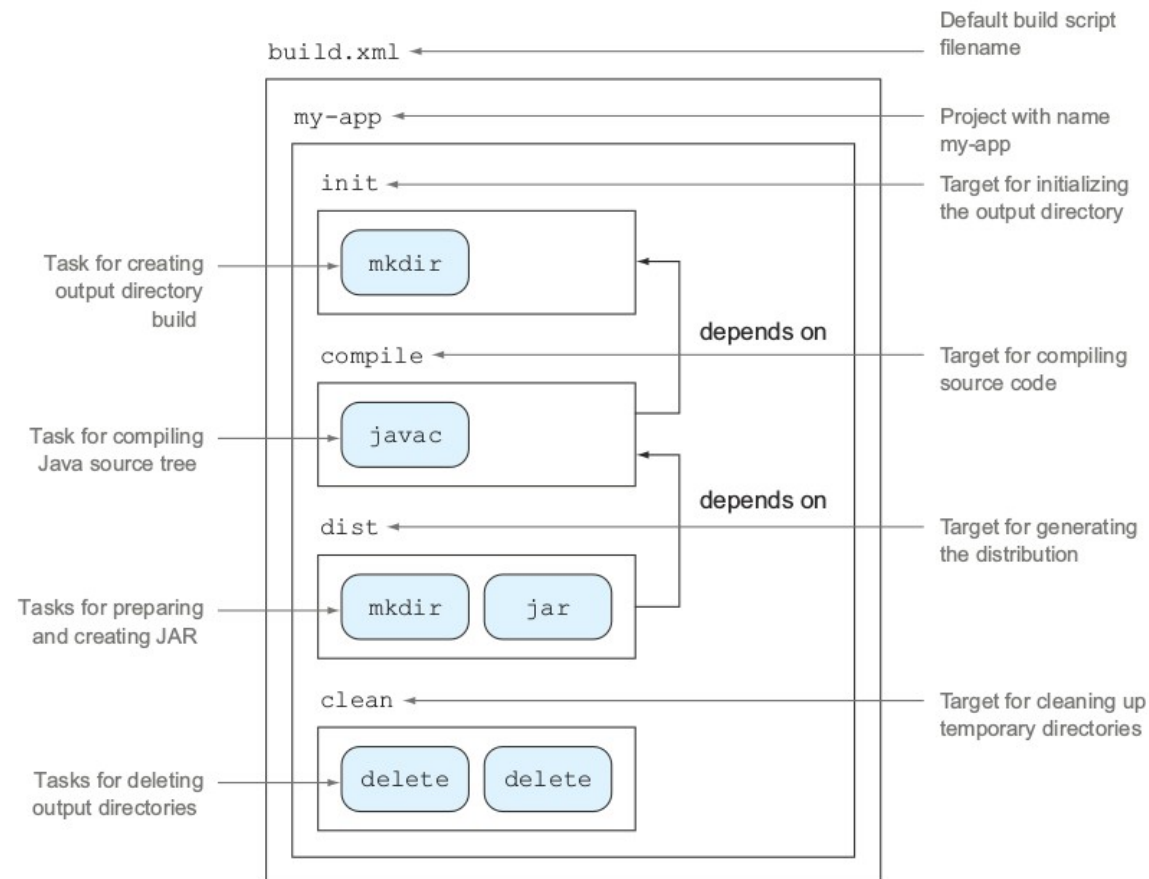
# Ant

---

- ♦ **Apache Ant (Another Neat Tool)** : L'objectif principal est de d'automatiser les tâches typiques nécessaires à un projet Java (Compilation, Tests unitaires, Packaging Jar, Javadoc, ...)
- ♦ Fournit de nombreuses tâches prédéfinies et peut être étendu avec de nouvelles tâches écrites en Java
- ♦ Ne fournit pas de gestionnaire de dépendances mais s'intègre facilement avec Ivy
- ♦ Fichier de build en XML (*build.xml*)



# Structure *build.xml*





# Example

---

```
<project name="my-app" default="dist" basedir=". ">
  <property name="src" location="src"/>
  <property name="build" location="build"/>
  <property name="dist" location="dist"/>
  <property name="version" value="1.0"/>

  <target name="init">
    <mkdir dir="${build}"/>
  </target>

  <target name="compile" depends="init" description="compile the source">
    <javac srcdir="${src}" destdir="${build}"/>
  </target>

  <target name="dist" depends="compile" description="generate the distribution">
    <mkdir dir="${dist}"/>
    <jar jarfile="${dist}/my-app-${version}.jar" basedir="${build}"/>
  </target>

  <target name="clean" description="clean up">
    <delete dir="${build}"/>
    <delete dir="${dist}"/>
  </target>
</project>
```



# Inconvénients

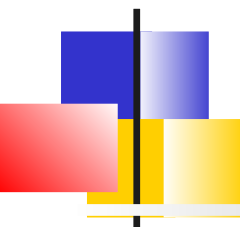
---

- Usage de XML => Fichier de build long
- Logique de build complexe conduit à des scripts de builds longs et difficilement maintenable (*if-then-else* avec un langage de markup!!).
- *Ant* ne donne pas de recommandation pour la mise en place d'un projet => chaque projet est différent, bcp de copier/coller entre projets. Chaque nouveau développeur doit comprendre le build du projet.
- N'expose pas d'API permettant de connaître le modèle interne ni de métriques sur l'exécution.
- *Ant* sans *Ivy* nécessite une gestion manuelle des dépendances
- *Ant* avec *Ivy* nécessite un effort d'intégration supplémentaire

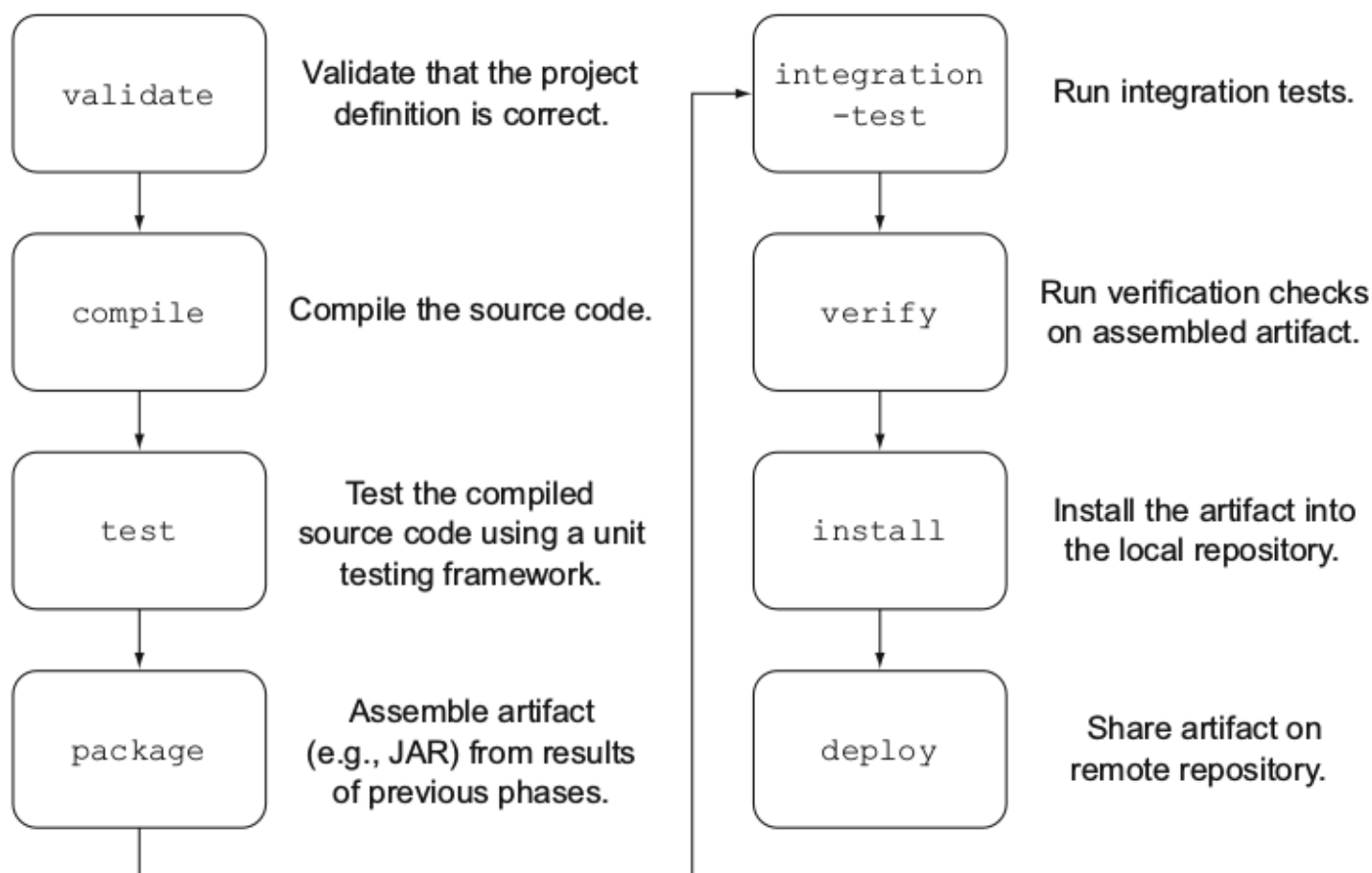


# Maven

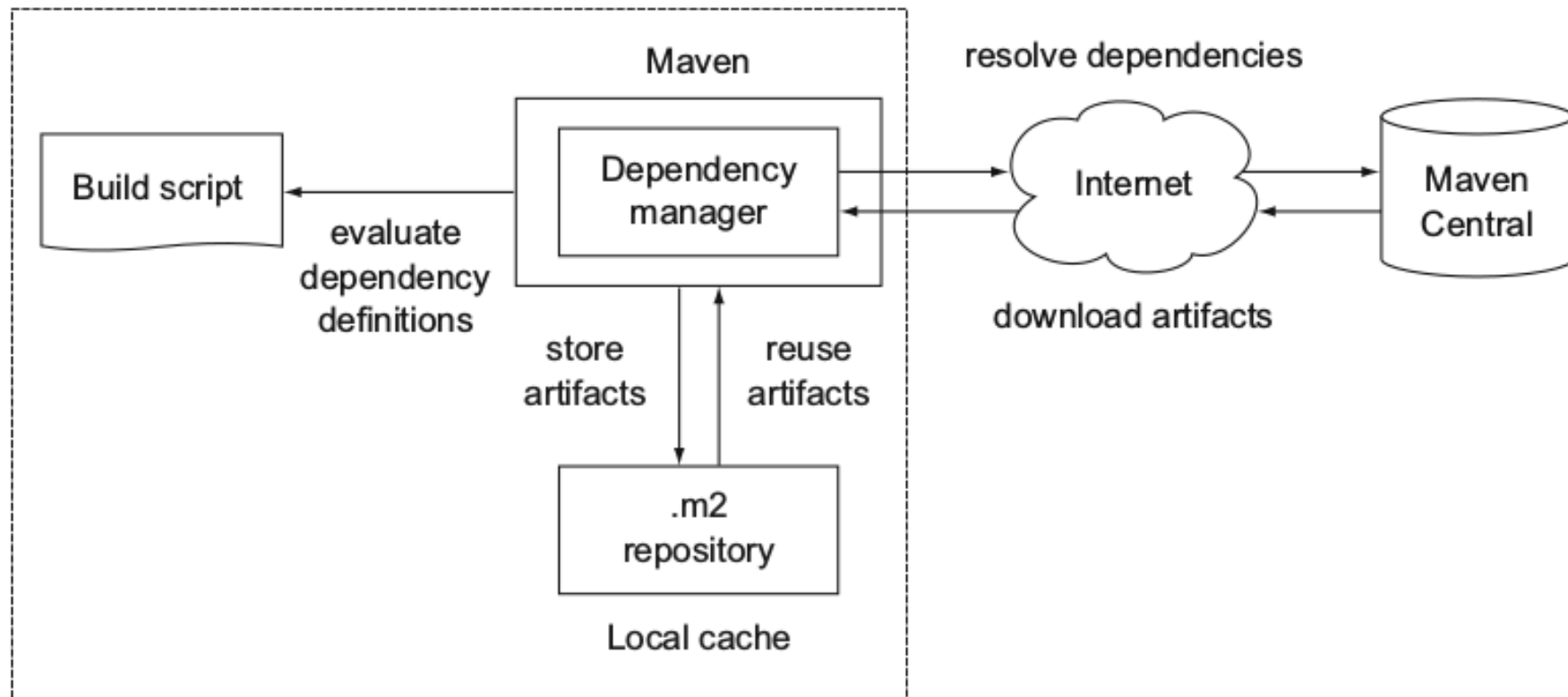
- Maven applique l'idée « **Convention plutôt que configuration** »  
=> un projet complet si il adhère aux conventions nécessite peu de lignes de XML
- Maven permet de générer de la **documentation**  
Site web avec Javadoc, licences, développeurs
- Les fonctionnalités cœur extensibles via des **plugins**.  
Les plugins sont nombreux, développer son propre plugin n'est pas anodin.
- Maven définit le cycle de vie d'un build comme une suite ordonnée de **phases**.  
Chaque phase correspond à l'exécution de plugins prédéfinis ou custom
- Maven gère les **dépendances** et propose des **dépôts d'artefacts**
- Il gère également les projets **multi-modules**



# Phases Maven



# Dépendances et dépôts





# Exemple *pom.xml*

---

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <packaging>jar</packaging>
  <version>1.0</version>

  <dependencies>
    <dependency>
      <groupId>org.apache.commons</groupId>
      <artifactId>commons-lang3</artifactId>
      <version>3.1</version>
      <scope>compile</scope>
    </dependency>
  </dependencies>
</project>
```



# Inconvénients

---

- Comme avec Ant, le format XML rend le fichier *pom.xml* long et + difficile à lire et appréhender
- La structure par défaut et le cycle de vie imposé sont souvent trop restrictifs et peuvent ne pas correspondre aux besoins d'un projet
- Le développement de plugins est assez lourd :  
Connaître les Mojos, fournir une description en XML et annotations Java
- Maven est limité à Java.
- Une dépendance téléchargée dans le dépôt local ne stocke pas sa provenance. Cela peut être un souci de portabilité





# Objectifs de Gradle

---

- Un langage de build expressif, déclaratif et maintenable
- Des arborescences et cycle de vie projet standardisés, mais offrant une flexibilité complète
- Possibilité de facilement ajouter de la logique personnalisée
- Support pour les multi-projets
- Support pour la gestion de dépendances.
- Facilité d'intégration et de migration des outils existants (Ant/Maven).
- Builds performants et scalable
- Ouverture hors Java



# Contexte

---

Les projets actuels utilisent des **pires logicielles variées** faisant usage de différents langages de programmation, de framework de test ou d'analyse.

Avec l'arrivée des méthodes agiles, le build doit permettre **l'intégration, la release et le déploiement en continu.**



# Code plutôt que XML

---

En gardant une approche configuration par convention, *Gradle* permet d'implémenter son build de façon déclarative en utilisant un DSL.

- De la logique personnalisée peut alors être codée en *Groovy* ou *Kotlin*

*Gradle* fournit également sa propre implémentation pour la gestion des dépendances et les projets multi-modules ; tout en essayant le plus possible d'être compatible avec Ivy et Maven

*Gradle* aide à la migration en supportant directement les tâches Ant



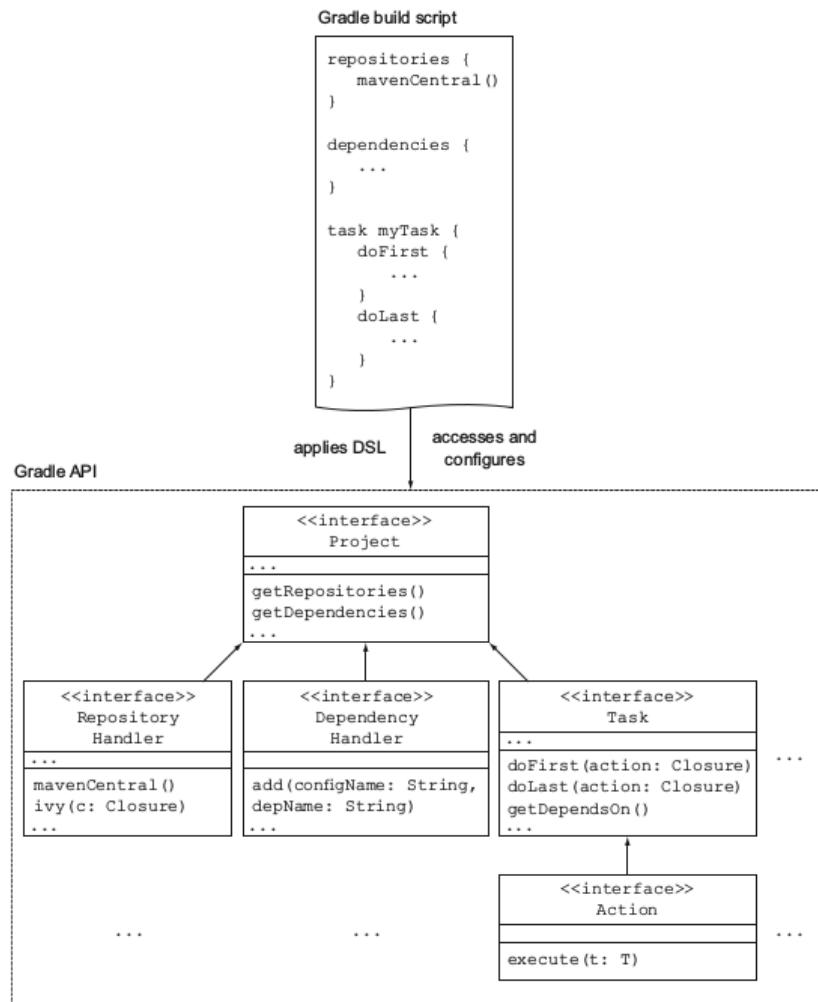
# Build is script

---

Un fichier de build (*build.gradle*) est un script exécuté par le moteur Gradle.

Le script représente un modèle objet (Projet, Tasks, ...) qui peut être directement manipulé par du code spécifique

# Correspondance fichier de build et modèle interne





# *Build is code*

---

Chaque élément d'un script *Gradle* correspond à une classe Java.

- *Groovy/Kotlin* permettent juste de rendre plus compact l'écriture de code avec la programmation fonctionnelle
- Gradle expose également des hooks dans les phases de son cycle de vie, permettant de configurer, monitorer, influencer l'exécution



# Conventions ... flexibles

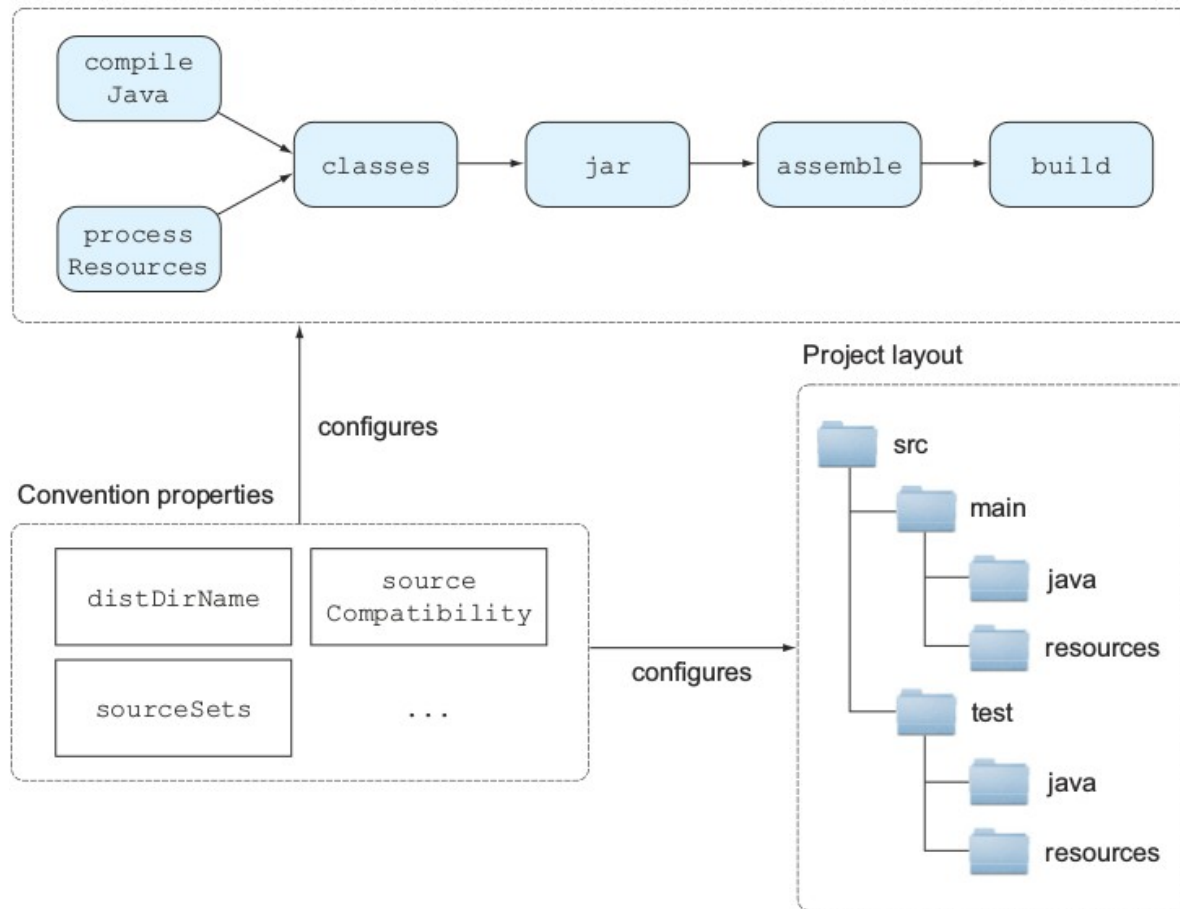
---

*Gradle* fournit des recommandations et des valeurs par défaut sensées pour les projets.

- Les tâches par défaut suffisent dans pas mal de cas.
- Mais, *Gradle* permet également de facilement remettre en cause ses conventions

*Gradle* fournit des archétypes pour les projets *Java, Scala, Groovy, Kotlin, C++, ....*

# Java







# Scalable

---

*Gradle*, pour des soucis de scalabilité et de performance

- supporte les builds incrémentaux en gérant les entrées sorties des différentes tâches du build
- Supporte l'exécution // des tests
- Utilise un processus démon, pour optimiser les performances des build



# OpenSource

---

*Gradle* est sous licence Apache License 2.0

1ère release Avril 2008

Forum : <https://discuss.gradle.org/>

Offre commerciale et support :  
*Gradleware*



# Introduction

---

Objectifs d'un outil de build  
Gradle vs Ant/Maven  
**Installation, Commandes**  
Un bref tour de Groovy



# Moyens

---

Gradle peut être installé sur Linux, MacOS ou Windows

- via des gestionnaires de paquet comme SDKMAN, Homebrew, ou Scoop
- Ou manuellement

*Cependant, il est rare que les développeurs d'un projet aient besoin d'installer Gradle.*

**Gradle Wrapper** permet des installations automatiques et de travailler avec différentes versions de Gradle. (Voir + loin)



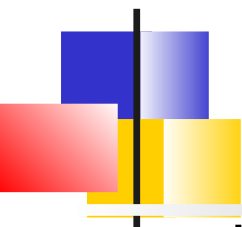
# Pré-requis

---

JDK 8+

Java est dans le Path ou indiqué dans  
*JAVA\_HOME*

*Groovy* est livré avec la distribution de  
Gradle



# Gestionnaire de paquets

---

Linux : SDK permet l'installation de plusieurs versions

```
sdk install gradle 4.6
```

MacOS : Homebrew et MacPort

```
brew install gradle
```

```
sudo port install gradle
```

Windows :

```
scoop install gradle
```

```
choco install gradle
```



# Installation manuelle

---

Téléchargement de la distribution  
(binaires ou sources)

Dézipper

Ajouter `${GRADLE_HOME}/bin` dans le  
*PATH*

=> Vérification :  
**gradle -v**



# Premier script

---

La commande **gradle** recherche le fichier nommé par défaut **build.gradle** dans son répertoire

Le fichier définit des **tâches**

Exemple *HelloWorld* :

```
task helloWorld {  
    doLast {  
        println 'Hello world!'  
    }  
}  
---  
gradle -q helloWorld
```





# Autre exemple

---

```
// L'opérateur << est un raccourci pour doLast
```

```
task startSession << {  
    chant()  
}
```

```
// Utilisation d'une tâche Ant
```

```
def chant() {  
    ant.echo(message: 'Repeat after me...')  
}
```

```
// Définition dynamique de tâches
```

```
3.times {  
    task "yayGradle${it}" << {  
        println 'Gradle rocks'  
    }  
}
```

```
// Dépendances entre tâches
```

```
yayGradle0.dependsOn startSession  
yayGradle2.dependsOn yayGradle1, yayGradle0  
task groupTherapy(dependsOn: yayGradle2)
```



# Commande en ligne

---

***gradle tasks*** : Liste les tâches disponibles d'un projet groupées par groupe de tâches

***gradle properties*** : Les propriétés du projet disponibles

Abréviation en *camelCase* : *gradle gT*

Option ***-x*** pour exclure une tâche



# Options

---

Principales options :

- ?, -h, --help**: Affiche toutes les options disponibles
- b, --build-file**: Le fichier de build, par défaut *build.gradle*
- offline**: Mode offline pour tester si le cache local a tout ce qu'il faut
- D, --system-prop**: Propriété JVM
- P, --project-prop** : Propriété du projet
- i, --info**: Logger Gradle à INFO
- s, --stacktrace**: Affichage de la stack trace en cas d'erreur
- q, --quiet**: Quiet mode

Les options peuvent être combinées, exemple -is



# Démon Gradle

---

Un démon démarre automatiquement après la première commande *gradle*.

- Il est utilisé par les builds suivants afin d'optimiser les temps de build.
- Il s'arrête automatiquement après 3h d'inactivité



# Introduction

---

Objectifs d'un outil de build  
Gradle vs Ant/Maven  
Installation, Commandes  
**Un bref tour de Groovy**



# Sources

---

Les fichiers sources de Groovy sont soit :

- Des définitions de classe
- Des scripts qui pourront alors être démarrés par une commande en ligne.



# Déclaration de classes

---

La déclaration de classe est similaire à Java, les méthodes sont *public* par défaut

```
class Book {  
    private String title  
    Book (String theTitle) {  
        title = theTitle  
    }  
    String getTitle(){  
        return title  
    }  
}
```



# Scripts

---

Les **scripts** sont des fichiers texte (ayant par convention l'extension *\*.groovy*)

Il peuvent être exécutés par :

> `groovy myfile.groovy`

- Les scripts contiennent des instructions qui ne sont pas encapsulées par une déclaration de classe.
- Ils peuvent contenir des définitions de méthodes
- Un script est parsé intégralement avant son exécution





# Exemple de script

---

**# Book.groovy est dans le classpath**

```
Book gina = new Book('Groovy in Action')  
assert gina.getTitle() == 'Groovy in Action'
```

**# Appel de méthode avant sa déclaration**

```
assert getTitleBackwards(gina) == 'noitcA ni yvoorG'  
String getTitleBackwards(book) {  
    String title = book.getTitle()  
    return title.reverse()  
}
```



# *GroovyBeans*

---

Un **GroovyBean** est un JavaBean défini en Groovy.

Groovy facilite le travail avec des beans :

- Il génère les accesseurs
- Il permet un accès simplifié aux propriétés
- Il simplifie l'enregistrement des gestionnaires d'événements associés

```
class BookBean {  
    String title  
}  
def groovyBook = new BookBean()  
// Appel du setter  
groovyBook.title = 'Groovy conquers the world'
```



# Annotations

---

Groovy permet d'utiliser et de définir des **annotations** comme Java

- Exemple : *@Immutable* qui est sensiblement équivalent à *final*

Les annotations *Groovy* instruisent le compilateur d'effectuer une transformation AST (*abstract syntax tree*),

Par exemple : ajout/suppression de méthodes, modification de la structure de code.

Il est possible de définir ses propres transformations (*meta-programming* à la compilation)



# String

---

En Groovy, les littéraux String peuvent utiliser les simples ou double-quotes.

La version double-quotes appelée Gstring permet l'utilisation de placeholders résolues à l'exécution.

- Exprimés par ***`${expression}`*** ou tout simplement ***`$reference`***.

```
def nick = 'ReGina'  
def book = 'Groovy in Action, 2nd ed.'  
assert "${nick.toUpperCase()} is $book" == 'REGINA is Groovy in  
Action, 2nd ed.'
```



# Littéraux

---

Groovy permet différentes options pour les littéraux :

- Les **simple quotes** pour les *String* Java
- Les **double-quotes** sont équivalentes au simple quote sauf si le texte contient un **\$** (placeholder). Dans ce cas, il s'agit d'une *Gstring*.
- Les **triples-quotes** (**simple** ou **double**) permettent du texte sur plusieurs lignes . Elles peuvent être de simples *String* ou des *GString*
- Si le texte démarre par un **/** ou un **\$/** (légères différences), Il n'est pas nécessaire d'échapper les backslash et cela facilite l'écriture de *regex*



# Examples

---

Start/end characters	Example	Placeholder resolved?	Backslash escapes?
Single quote	'hello Dierk'	No	Yes
Double quote	"hello \$name"	Yes	Yes
Triple single quote (' ' ')	'''===== Total: \$0.02 ====='''	No	Yes
Triple double quote (" " ")	"""first \$line second \$line third \$line"""	Yes	Yes
Forward slash	/x(\d*)y/	Yes	Occasionally
Dollar slash	\$/x(\d*)y/\$	Yes	Occasionally



# Méthodes additionnelles des String

---

```
print 'Hello Groovy!'
String greeting = 'Hello Groovy!'
// Utilisation de range
assert greeting[6..11] == 'Groovy'
// Opérateur -
assert 'Hi' + greeting - 'Hello' == 'Hi Groovy!'
// Comptage de caractère
assert greeting.count('o') == 3
// Ajout de caractère à droite ou gauche, centrage
assert 'x'.padLeft(3) == ' x'
assert 'x'.padRight(3, '_') == 'x__'
assert 'x'.center(3) == ' x '
// Opérateur *
assert 'x' * 3 == 'xxx'
```



# Tout Objet

---

Dans *Groovy*, tout est objet. Il n'y a pas de type primitif comme en Java

- Tout nombre, booléen est converti en une objet Java.
- Les notations des types primitifs sont cependant toujours supportées :

Groovy permet de spécifier explicitement le type d'une variable **ou** de l'omettre

- Le mot-clé **def** est utilisé pour indiquer qu'aucun type n'est spécifié.

```
def x = 1
def y = 2
assert x + y == 3
assert x.plus(y) == 3
assert x instanceof Integer
```





# Type safe

---

Même si le type de variable peut être implicite, Groovy est ***type safe*** :  
Un objet d'un type particulier ne peut pas être assigné à un autre type si il n'y a pas de conversion définie.

=> On peut donc omettre les marqueurs de type mais Groovy effectuera les vérifications de type à l'exécution.



# Surcharge d'opérateurs

---

*Groovy* base ses opérateurs sur des appels de méthode.

- Ainsi, les opérateurs peuvent être surchargés et s'appliquer à différents types
- Les opérateurs peuvent alors être utilisés sur nos propres classes du moment qu'elles implémentent la méthode associée (Encore une fois, pas besoin d'implémenter une interface comme en Java !).



# Examples

Operator	Name	Method	Works with
<code>a + b</code>	Plus	<code>a.plus(b)</code>	Number, String, StringBuffer, Collection, Map, Date, Duration
<code>a - b</code>	Minus	<code>a.minus(b)</code>	Number, String, List, Set, Date, Duration
<code>a * b</code>	Star	<code>a.multiply(b)</code>	Number, String, Collection
<code>a / b</code>	Divide	<code>a.div(b)</code>	Number
<code>a % b</code>	Modulo	<code>a.mod(b)</code>	Integral number
<code>a++</code> <code>++a</code>	Postincrement Preincrement	<code>def v = a; a = a.next(); v</code> <code>a = a.next(); a</code>	Iterator, Number, String, Date, Range
<code>a--</code> <code>--a</code>	Postdecrement Predecrement	<code>def v = a; a = a.previous(); v</code> <code>a = a.previous(); a</code>	Iterator, Number, String, Date, Range
<code>-a</code>	Unary minus	<code>a.unaryMinus()</code>	Number, ArrayList



# Méthodes et paramètres

---

- Les qualifieurs Java peuvent être utilisés
- Déclarer un type de retour est optionnel (utilisation de *def*)
- L'instruction *return* est optionnelle même si on a déclaré un type de retour. La dernière ligne est retournée.
- La visibilité par défaut est *public*
- La déclaration explicite des types des paramètres est optionnelle
- Lorsque la déclaration n'est pas précisée, le type *Object* est utilisé
- Les paramètres peuvent avoir une valeur par défaut
- Les appels de méthodes peuvent respecter l'ordre de déclaration des paramètres ou utiliser des paramètres nommés avec une *Map*



# Exemples

---

```
class Summer {  
  def sumWithDefaults(a, b, c=0){ // Valeur par défaut de c  
    return a + b + c  
  }  
  def sumWithList(List args){  
    // Closure avec Initialisation de sum à 0  
    return args.inject(0){sum,i -> sum += i}  
  }  
  def sumWithOptionals(a, b, Object[] optionals){  
    return a + b + sumWithList(optionals.toList())  
  }  
}  
  
def summer = new Summer()  
assert 2 == summer.sumWithDefaults(1,1) // Appel sans paramètre c  
assert 3 == summer.sumWithDefaults(1,1,1)  
assert 3 == summer.sumWithList([1,1,1])  
assert 2 == summer.sumWithOptionals(1,1) // Dernier paramètre non renseigné  
assert 3 == summer.sumWithOptionals(1,1,1)
```



# Parenthèses

Les parenthèses peuvent également être omises pour les expressions de haut-niveau :

```
println "Hello"  
method a, b  
<=>  
println("Hello")  
method(a, b)
```

Dans le cas d'une closure :

```
list.each( { println it } )  
list.each(){ println it }  
list.each { println it }
```

Attention les parenthèses sont obligatoires lors d'appels imbriqués :

```
def foo(n) { n }  
def bar() { 1 }  
println foo 1 // Ne marche pas  
def m = bar   // Ne marche pas
```



# *Map* et paramètres nommés

---

Le passage des paramètres lors des appels de méthodes peut se faire en utilisant une *Map* permettant de nommer les paramètres

```
class Summer {  
  def sumNamed(Map args){  
    ['a','b','c'].each{ println args.get(it,0)}  
    return args.a + args.b + args.c  
  }  
}  
  
def summer = new Summer()  
assert 2 == summer.sumNamed(a:1, b:1)
```



# Closures

---

Les ***closures*** Groovy permettent la programmation fonctionnelle.

Un bloc d'instructions peut être passé en paramètre à une méthode.

Un objet de type *Closure* travaille en sous-main

```
[1, 2, 3].each { entry -> println entry }
```

**// Variable implicite *it***

```
[1, 2, 3].each { println it }
```





# Déclaration

---

## 3 façons de déclarer une *Closure*

- Déclaration simple
- Affectation à une variable
- Référence à une méthode

La syntaxe utilisée par les 2 premières façons est :

```
{ [closureParameters ] -> statements }
```



# Déclarations

---

Si la *Closure* n'a qu'un seul paramètre sa déclaration est optionnelle, et la variable ***it*** peut être utilisée.

```
log = ''  
(1..10).each{ log += it }  
// Version longue  
  
log = ''  
(1..10).each({ counter -> log += counter })
```

```
// Déclaration par affectation  
def printer = { line -> println line }
```

```
// Référence à la méthode sizeUpTo(it)  
SizeFilter filter6 = new SizeFilter(limit:6)  
Closure sizeUpTo6 = filter6.&sizeUpTo
```



# Objet délégué

---

La particularité des Closure est qu'il est possible de définir son objet **délégué** : (la résolution de this dans le bloc de la Closure)

Cette technique est très utilisé par Gradle



# Collections

---

*Groovy* facilite l'utilisation des collections

- Il ajoute la collection *Range* représentant un intervalle :  
`def a = 0..10`
- Les collections peuvent être déclarés par des littéraux.
- Ils ont des opérateurs spécialisés et de nombreuses améliorations par rapport au JDK

La notation utilisée est différente de Java mais la sémantique reste la même



# Exemples

---

```
// roman est une List (ArrayList)
def roman = ['', 'I', 'II', 'III', 'IV', 'V', 'VI', 'VII']
// On accède à un élément comme un tableau Java
assert roman[4] == 'IV'
// Il n'y a pas d'ArrayIndexOutOfBoundsException
roman[8] = 'VIII'
assert roman.size() == 9
// Les maps peuvent être facilement instanciées (LinkedHashMap par défaut)
def http = [
  100 : 'CONTINUE',
  200 : 'OK',
  400 : 'BAD REQUEST'
]
// Accès aux éléments avec la notation Array Java
assert http[200] == 'OK'
// Méthode put simplifiée
http[500] = 'INTERNAL SERVER ERROR'
assert http.size() == 4
```



# Boucle sur les Range

---

**// Boucle for in**

```
def log = ''  
for (element in 5..9){ log += element }  
assert log == '56789'  
log = ''  
for (element in 9..5){ log += element }  
assert log == '98765'
```

**// Méthode each**

```
log = ''  
(9..<5).each { element -> log += element }  
assert log == '9876'
```



# Les listes

---

**// Intersection**

```
myList = ['a', 'b', 'c']  
assert ['x','a','z'].grep(myList) == ['a']
```

**// for in sur une liste**

```
def list = [0, 1, 2, 3]  
for (j in list) {  
    assert j == list[j]  
}
```

**// each avec une closure**

```
list.each() { item ->  
    assert item == list[item]  
}
```



# Manipuler les Map

---

```
def myMap = [a:1, b:2, c:3]
// Récupérer un élément existant
assert myMap['a'] == 1
assert myMap.a == 1
assert myMap.get('a') == 1
assert myMap.get('a',0) == 1
// Essayer de récupérer un élément manquant
assert myMap['d'] == null
assert myMap.d == null
assert myMap.get('d') == null
// Valeur par défaut
assert myMap.get('d',0) == 0
assert myMap.d == 0
// Mise à jour
myMap['d'] = 1
assert myMap.d == 1
myMap.d = 2
assert myMap.d == 2
```





# Méthodes *any* et *every*

---

GDK ajoute les méthodes ***any*** et ***every*** qui retournent un *Boolean* indiquant si une ou toute les entrées de la Map satisfont une *closure* donnée.

```
assert myMap.any {entry -> entry.value > 2 }  
assert myMap.every {entry -> entry.key< 'd'}
```



# Boucles sur les Map

---

Il est possible d'itérer sur les entrées ou sur les clés et valeurs séparément

```
def myMap = [a:1, b:2, c:3]
// Boucle sur les entrees
def store = ''
myMap.each { entry ->
store += entry.key
store += entry.value
}
assert store == 'a1b2c3'
// Boucle sur les clés et valeurs
store = ''
myMap.each { key, value -> store += key ; store += value}
assert store == 'a1b2c3'
// Boucle sur les clés
store = ''
for (key in myMap.keySet()) {store += key}
assert store == 'abc'
// Boucle sur les valeurs
store = ''
for (value in myMap.values()) {store += value}
assert store == '123'
```



# *regex*

---

Groovy s'appuie sur Java pour les expressions régulières et ajoute 3 opérateurs :

- find , *=~*
- match , *==~*
- pattern , *~string*

L'écriture de *regex* est facilitée via le marqueur de String */*

```
assert "\\d" == /\d/
```



# Exemples

## boucle sur les occurrence

---

```
def myFairStringy = 'The rain in Spain stays mainly in the plain!'
def wordEnding = /\w*ain/
def rhyme = /\b$wordEnding\b/
def found = ''
myFairStringy.eachMatch(rhyme) { match ->
    found += match + ' '
}
assert found == 'rain Spain plain '
found = ''
(myFairStringy =~ rhyme).each { match ->
    found += match + ' '
}
assert found == 'rain Spain plain '
```



# Concepts cœur

---

## **Composants d'un build**

Manipulation des tâches

Hook du cycle de build

Plugins



# Composants d'un build

---

Chaque build *Gradle* est constitué de **projets**, de **tâches** et de **propriétés**.

- Chaque build contient au moins 1 projet qui contient une ou plusieurs tâches.
- Les projets et les tâches exposent des propriétés qui peuvent être utilisées pour contrôler le build
- Les projets et les tâches correspondent à des classes de l'API Gradle



# Projet

---

Lors du démarrage d'un build, *Gradle* instancie une classe ***org.gradle.api.Project*** en fonction du script *build.gradle* et rend la variable ***project*** disponible

C'est la variable implicite

```
// Utilisation de la variable implicite  
setDescription("My Beautiful Project")  
println "Description of project $name: " + description
```



# Initialisation du build cas général

---

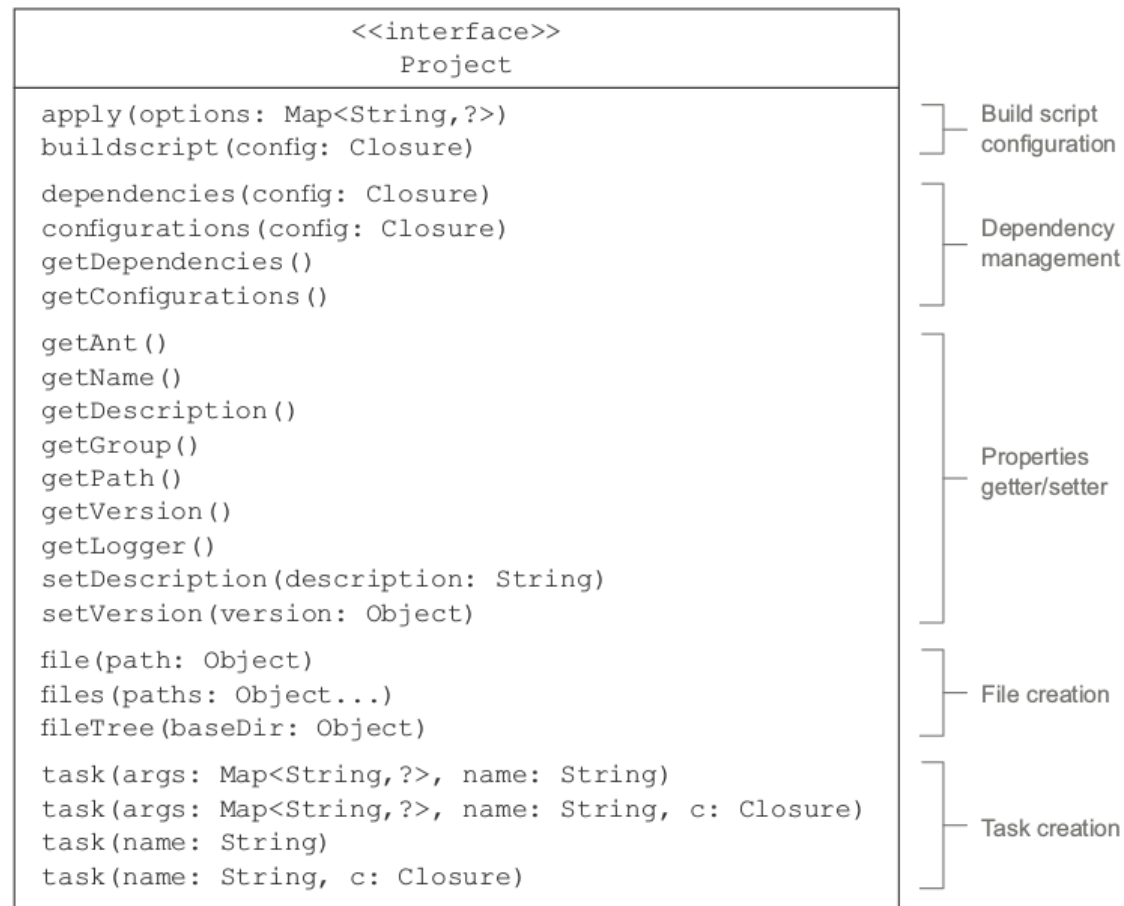
Durant l'initialisation, Gradle assemble un objet *Project* pour chaque fichier "*build.gradle*" participant au build comme suit :

- Crée une instance de la classe ***Settings***
- Évalue le script "***settings.gradle***" si il est présent afin de configurer l'instance *Settings*
- Utilise l'objet *Settings* pour créer la hiérarchie des instances *Project*
- Finalement, évalue chaque projet en exécutant son fichier "***build.gradle***".





# Interface *Project*





# Tâches

---

Une **tâche** représente une partie atomique d'un build comme la compilation ou la génération de la javadoc.

Chaque tâche appartient à un projet.

La création d'une tâche s'effectue généralement par la méthode **task** disponible avec différentes signatures :

```
task myTask
task myTask { configure closure }
task myTask(type: SomeType)
task myTask(type: SomeType) { configure closure }
```



# Groupe/Description

---

Une tâche a :

- Un nom
- Éventuellement un groupe et une description

Les tâches sans groupe ne sont pas  
affichées lors de  
***gradle tasks***



# Actions

---

Une tâche est constituée d'une séquence d'objets **Action**.

Lorsque la tâche est exécutée, chaque action est exécutée séquentiellement par l'appel de *Action.execute(T)*.

Des actions peuvent être ajoutées en appelant les méthodes ***doFirst(Action)*** ou ***doLast(Action)***.



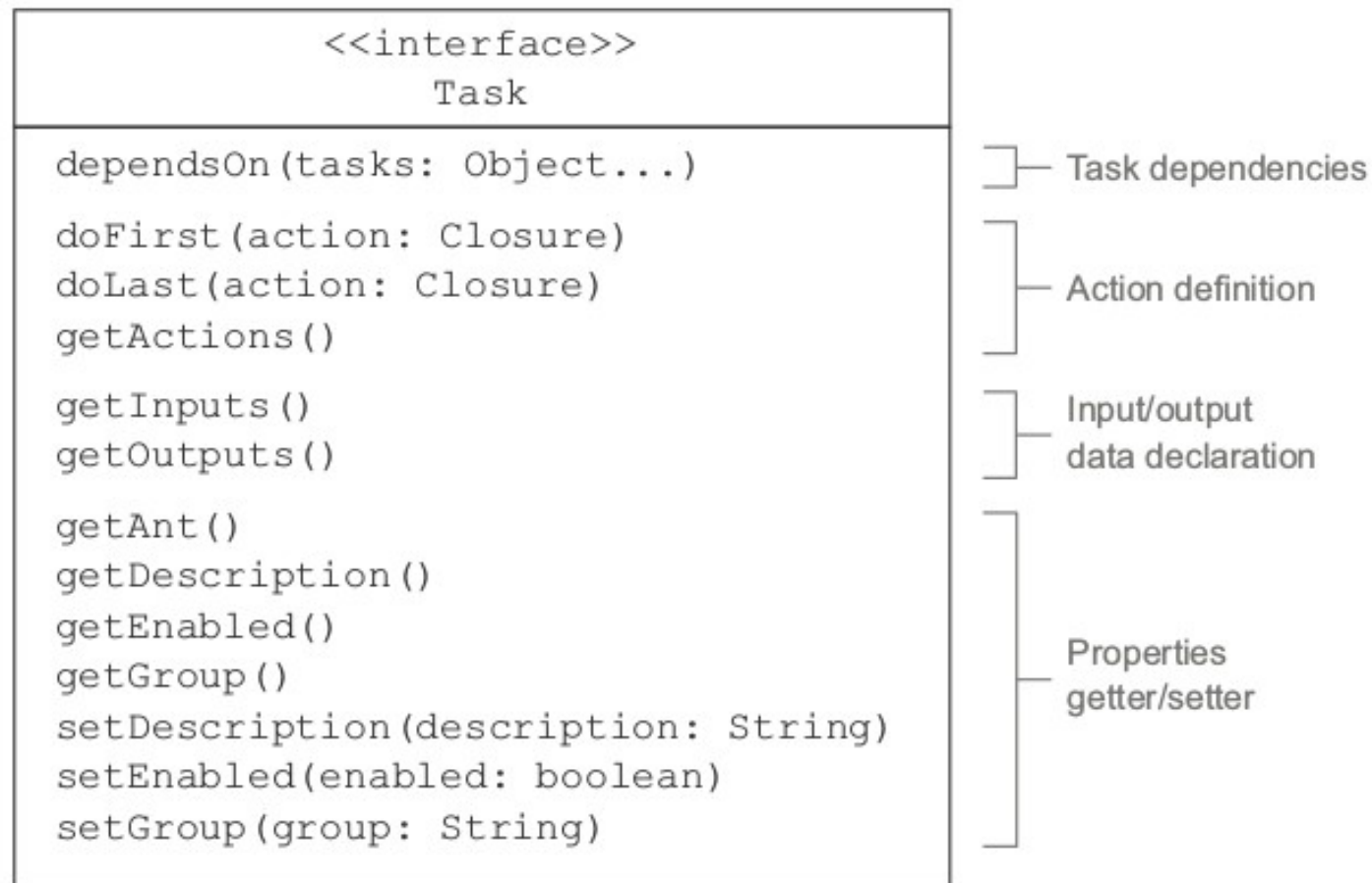
# Dépendances entre tâches

---

Une tâche peut avoir des **dépendances** sur d'autres tâches.

- Gradle garantit le séquençement des tâches dépendantes lors de l'exécution du build.
- Il est également possible de contrôler l'ordre d'exécution des dépendances

# Task





# Propriétés de configuration Project

---

Les instances de *Project* fournissent des propriétés de configuration accessibles via des méthodes *get/set*

- Des propriétés sont prédéfinies pour un projet accessible par `getProperties`  
*rootDir, buildDir, configurations, plugins, ....*

Des nouvelles propriétés peuvent être définies via :

- Le fichier ***gradle.properties*** (dans le répertoire projet ou dans `USER_HOME`)
- La ligne de commande gradle ***-P***
- Des variables d'environnement préfixées par ***ORG\_GRADLE\_PROJECT\_***



# Extension de Project

---

*Gradle* permet également d'augmenter les classes de type ***ExtensionAware*** (*Project*, *Task*, ...)

De nouveaux attributs, les *extra properties*, sont stockées dans des *Map* sous la forme clé/valeur

L'espace de nom ***ext*** doit être utilisé.

Exemples :

```
project.ext.myProp = 'myValue'
ext {
    someOtherProp = 123
}
assert myProp == 'myValue'
println project.someOtherProp
ext.someOtherProp = 567
```





# Propriétés en ligne de commande

---

Bien distinguer, les propriétés projets des propriétés configurant le comportement du moteur Gradle :

- Flag :  
Exemple *--build-cache*
- Propriétés system JVM :  
*-D*  
*systemProp.http.proxyHost* dans *gradle.properties*
- Propriétés Gradle  
*org.gradle.caching* dans *gradle.properties*
- Variables d'environnement  
*GRADLE\_OPTS*



# Concepts cœur

---

Composants d'un build

**Manipulation des tâches**

Hook du cycle de build

Plugins



# Déclarer les actions

---

Une tâche consiste en une séquence d'actions

L'interface *Task* fournit 2 méthodes pour ajouter une action : ***doFirst(Closure)*** et ***doLast(Closure)*** .

```
task printVersion {  
    doFirst {  
        println "Before reading the project version"  
    }  
    doLast {  
        println "Version: $version"  
    }  
}  
printVersion.doFirst { println "First action" }
```



# Propriétés par défaut

---

Une tâche a 2 propriétés par défaut :

- **group** : le groupe auquel la tâche appartient
- **description**

*Gradle* fournit également une implémentation de logger basé sur SLF4J qui ajoute le niveau QUIET aux niveaux habituels

```
task printVersion {  
    group = 'versioning'  
    description = 'Prints project version.'  
    doLast {  
        logger.quiet "Version: $version"  
    }  
}
```



# Définir les dépendances

---

L'attribut ***dependsOn*** permet de déclarer des dépendances vers une ou plusieurs tâches.

Si plusieurs dépendances sont indiquées, Gradle ne garantit pas d'ordre d'exécution entre ces tâches

```
// Pas de garantie sur l'ordre de second et first
task printVersion(dependsOn: [second, first]) {
    logger.quiet "Version: $version"
}
```



# Ordre des tâches

---

*Gradle* permet cependant de forcer l'ordre des exécutions des tâches.

Les relations d'ordre sont différentes des relations de dépendances car elle n'influent pas sur ce qui doit être exécutée mais seulement sur l'ordre.

2 méthodes sont disponibles sur *Task* :

- ***mustRunAfter*** : signifie que la tâche doit s'exécuter après si les 2 tâches sont dans le graphe d'exécution.
- ***shouldRunAfter*** : Moins restrictif. Ignoré si cette règle produit un cycle dans la séquence ainsi qu'en cas de parallélisme



# Finalizer

---

Les tâches ***finalizers*** permettent la libération de ressources après l'exécution d'une autre tâche quelque soit sa réussite.

```
// Second est déclenché par l'exécution de first
task first << { println "first" }
task second << { println "second" }
first.finalizedBy second
```



# Code arbitraire

---

Du code arbitraire *Groovy* (script ou classe) peut être ajouté dans le script de build.

- par exemple fournir des classes modélisant notre projet et les méthodes associées

```
version = new ProjectVersion(0, 1)
class ProjectVersion {
    Integer major
    Integer minor

    ProjectVersion(Integer major, Integer minor) {
        this.major = major
        this.minor = minor
    }

    @Override
    String toString() { "$major.$minor" }
}
```





# Tâches de configuration

---

Il est possible de définir des tâches contenant des instructions dans leur corps.

```
task myTask {  
    println "CONFIGURING MY TASK"  
    doFirst {  
        println "EXECUTING TASK"  
    }  
}
```

Ces instructions seront systématiquement exécutées lors de la phase de configuration du build



# Phases d'exécution

---

Lors de l'exécution d'un build, 3 phases distinctes sont exécutées :

- **Initialisation** : *Gradle* crée une instance de *Project*. Dans un contexte multi-projets, détermine les dépendances entre projets
- **Configuration** : En fonction de la ligne de commande, *Gradle* construit le graphe de tâches prenant part au build. Le code de configuration est exécuté
- **Exécution** : Les tâches du graphe d'exécution sont exécutées. L'ordre d'exécution est déterminé par leurs dépendances. Les tâches à jour ne sont pas exécutées.



# Exemple tâche de configuration

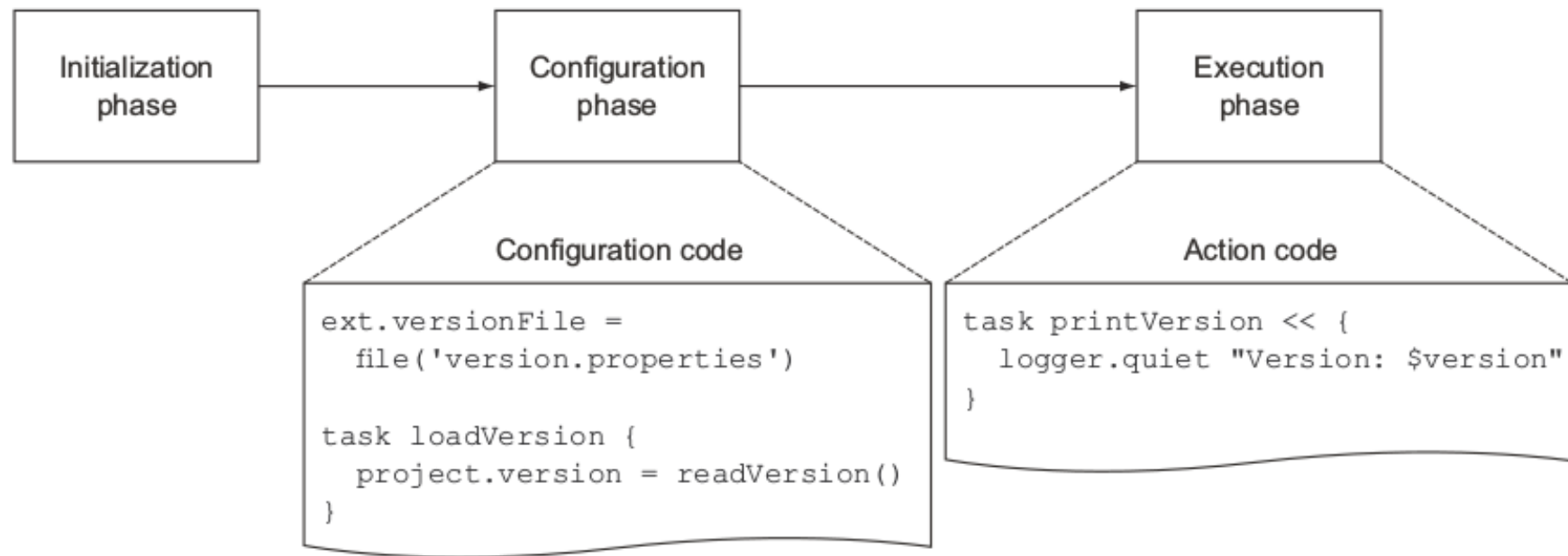
---

```
ext.versionFile = file('version.properties')
// Pas d'utilisation de doFirst ou doLast => Tâche de configuration
// Exécutée dans la phase de configuration
task loadVersion {
    project.version = readVersion()
}
ProjectVersion readVersion() {
    logger.quiet 'Reading the version file.'
    if(!versionFile.exists()) {
        throw new GradleException("Required version file does not exist:
$versionFile.canonicalPath")
    }

    Properties vProps = new Properties()
    versionFile.withInputStream { stream -> vProps.load(stream) }

    new ProjectVersion(vProps.major.toInteger(), vProps.minor.toInteger(),
vProps.release.toBoolean())
}
```

# Cycle de vie





# Build incrémental

---

*Gradle* détermine si une tâche est à jour en comparant les entrées et les sorties d'une tâche entre 2 builds

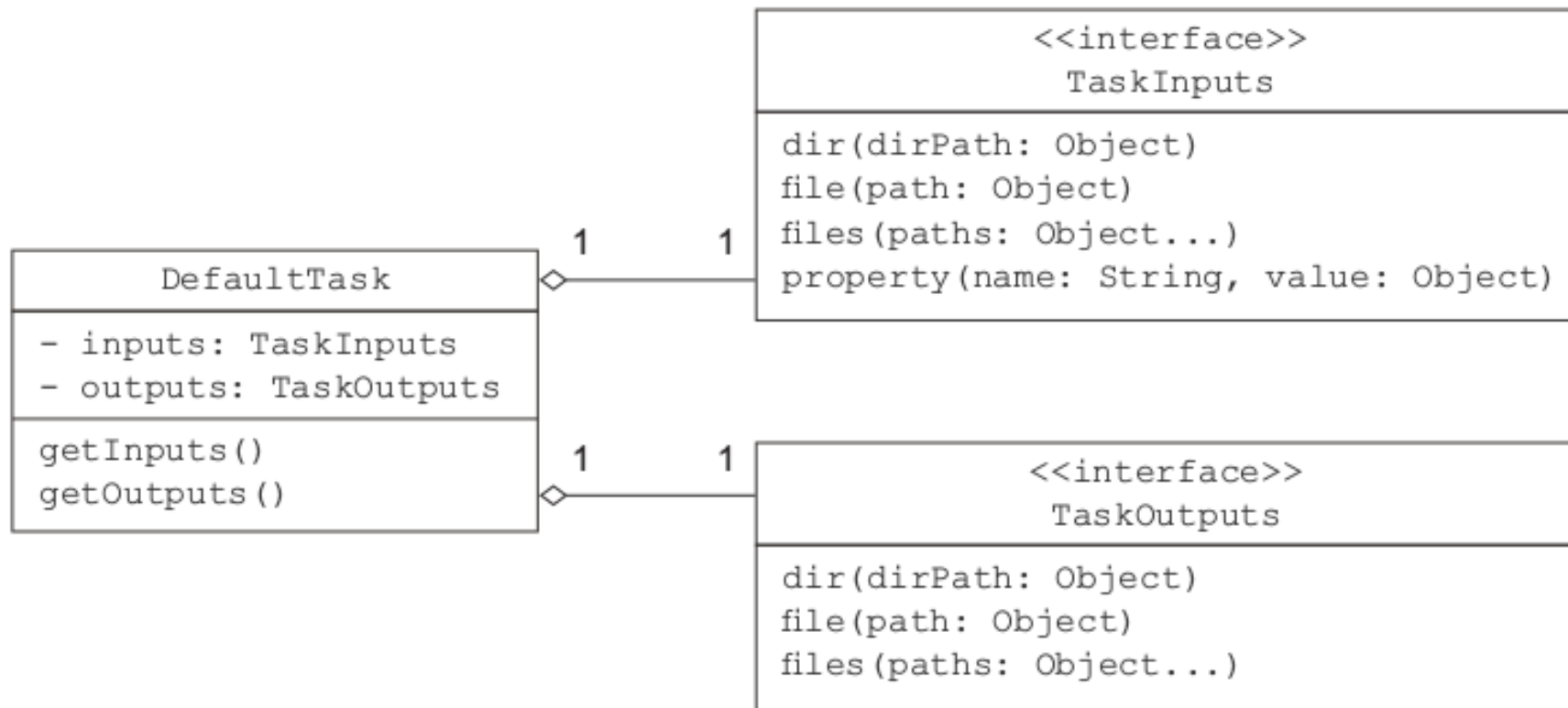
Les entrées/sorties (*input/output*) sont des champs de la classe *Task*

- Une entrée peut être :
  - un répertoire
  - un ou plusieurs fichiers
  - une propriété arbitraire
- Une sortie peut être :
  - un répertoire
  - un ensemble de fichiers

L'initialisation des entrées/sorties est effectuée pendant la phase de configuration



# *TaskInputs et TaskOutputs*





# Exemple

```
task makeReleaseVersion(group: 'versioning', description: 'Makes project
    a release version.') {
    // Configuration phase
    // l'entrée est une propriété qui prend comme valeur version.release
    // la sortie est le fichier modifié par la tâche
    inputs.property('release', version.release)
    outputs.file versionFile

    doLast {
        version.release = true
        ant.propertyfile(file: versionFile) {
            entry(key: 'release', type: 'string', operation: '=', value:
                'true')
        }
    }
}
```



# Types de tâche

---

Par défaut, les tâchesinstanciées sont de type  
***DefaultTask***

Il est possible de définir un type à la création et de profiter de comportement prédéfini

En général, il suffit de préciser la configuration en indiquant les entrée/sorties de la tâche

```
task copyDocs(type: Copy) {  
    from 'src/main/doc'  
    into 'build/target/doc'  
}
```





# Tâches prédéfinies Gradle

---

*Gradle* fournit beaucoup d'implémentations de l'interface *Task*

- *Copy, Zip, Jar, Exec, tar, Delete*
- *JavaCompile, Javadoc, JavaExec*
- *Jar, war, ear*
- *Jacoco, Checkstyle, FindBugs*
- ...

Voir : <https://docs.gradle.org/current/dsl/index.html>



# Exemple

---

```
task makePretty(type: Delete) {  
    // Appel de la méthode  
    // pendant la phase de configuration  
    delete 'uglyFolder', 'uglyFile'  
}
```

La tâche *Delete* contient une propriétés *delete* qui liste les fichiers,répertoires à supprimer

Elle expose la méthode *delete(args)* qui ajoute les arguments à la propriétés *delete*



# Exemple

---

```
task createDistribution(type: Zip, dependsOn: makeReleaseVersion) {  
    // référence implicite à la sortie de la tâche War  
    // => Une dépendance est déduite par Gradle  
    from war.outputs.files  
    // Les fichiers sources sont placés dans le répertoire src du fichier Zip  
    from(sourceSets*.allSource) {  
        into 'src'  
    }  
    // Ajouter le fichier de version au zip  
    from(rootDir) {  
        include versionFile.name  
    }  
}  
task backupReleaseDistribution(type: Copy) {  
    from createDistribution.outputs.files  
    into "$buildDir/backup"  
}  
task release(dependsOn: backupReleaseDistribution) << {  
    logger.quiet 'Releasing the project...'  
}
```



# Custom Task

---

Il est possible de définir ses propres types de tâche.

```
class ReleaseVersionTask extends DefaultTask {  
    // Déclaration des entrées/sorties par annotation  
    // Entrée requise sinon : TaskValidationException, (Voir @Optional)  
    @Input Boolean release  
    @OutputFile File destFile  
  
    ReleaseVersionTask() {  
        group = 'versioning'  
        description = 'Makes project a release version.'  
    }  
    @TaskAction  
    void start() {  
        project.version.release = true  
        ant.propertyfile(file: destFile) {  
            entry(key: 'release', type: 'string', operation: '=', value: 'true')  
        }  
    }  
}
```



# Utilisation

---

Pour utiliser la classe, il faut définir une tâche du type désiré et initialiser ses propriétés dans le bloc de configuration.

```
task makeReleaseVersion(type: ReleaseVersionTask) {  
    release = version.release  
    destFile = versionFile  
}
```



# Task rule

---

Gradle définit **task rule** qui exécute de la logique basée sur un gabarit de nom de tâche composé :

- D'une partie statique
- D'un « placeHolder » permettant d'influencer la logique

Exemple :

Gabarit : *increment<Classifier>Version*

Usage : *incrementMajorVersion*

La définition de task rule s'effectue sur une classe *TaskContainer* en utilisant la variable **tasks**



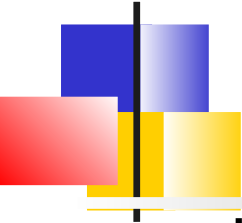
# Example

---

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

---

```
> gradle -q pingServer1
Pinging: Server1
```



# Sources pour le build

---

Le répertoire ***buildSrc*** peut être une alternative pour placer du code de build.

Le code peut alors être organisé en package et peut contenir des classes de test.

*Gradle* propose une arborescence pour les sources de build. Tout code trouvé dans ces répertoires est ajouté au classpath du script de build

- *src/main/java*
- *src/main/groovy*

Il faut alors importer les packages dans le script de build





# Concepts cœur

---

Composants d'un build  
Manipulation des tâches  
**Hooks du cycle de build**  
Plugins



# Callback

---

Il y a 2 façons d'exécuter du code réagissant à des événements de build (*callback*):

- Avec une closure
- En implémentant une interface *listener* fournie par l'API *Gradle* .

De nombreux hooks sont fournis par *Gradle* (Voir l'API des classes *Project* et *Gradle*)



## *org.gradle.api.invocation.Gradle*

---

***addBuildListener(buildListener)*** : Ajout d'une implémentation de *BuildListener*. Événements lors de l'exécution du build.

***addProjectEvaluationListener(listener)***  
Listener lors de l'évaluation des projets

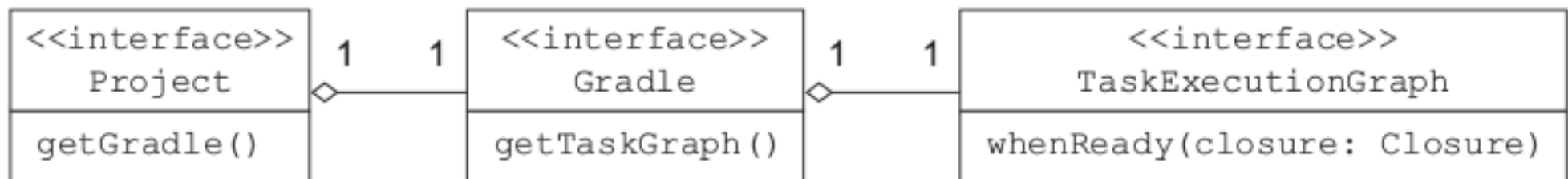
***afterProject(closure)*** : Ajout d'une closure après l'évaluation d'un projet



# Gradle

L'objet *Gradle* permet également d'accéder au graphe dirigé des tâches (avec toutes les dépendances calculés)

Il est possible alors d'utiliser la méthode *whenReady* pour ajouter un traitement par Closure





# Example

---

```
task distribution {
    doLast {
        println "We build the zip with version=$version"
    }
}

task release(dependsOn: 'distribution') {
    doLast {
        println 'We release now'
    }
}

gradle.taskGraph.whenReady {taskGraph ->
    if (taskGraph.hasTask(release)) {
        version = '1.0'
    } else {
        version = '1.0-SNAPSHOT'
    }
}
```



# Concepts cœur

---

Composants d'un build  
Manipulation des tâches  
Hooks du cycle de build  
**Plugins**



# Introduction

---

Toutes les fonctionnalités utiles de *Gradle* (comme compiler du code Java) sont apportées par les plugins.

## Les plugins

- Ajoutent de nouvelles tâches (Ex : *JavaCompile*)
- De nouveaux objets du modèle (Ex : *SourceSet*)
- Des valeurs par défaut, i.e des conventions (Ex : Code source dans *src/main/java*)
- Des extensions aux objets cœur



# Types de plugins

---

2 types de plugins :

- Les **plugins script** sont des scripts de build additionnels qui ajoutent des détails de configuration et utilisent une approche déclarative pour modifier un build final.
- Les **plugins binaires** sont des classes qui implémentent l'interface *Plugin* et adoptent une approche programmatique pour manipuler le build.  
Typiquement fourni via un jar.





# Utilisation d'un plugin

---

Lors de l'utilisation d'un plugin, Gradle doit :

- Le localiser
- L'appliquer. Exécuter *Plugin.apply(T)*

Cela est fait en une seule fois via une syntaxe déclarative légèrement différente en fonction du type du plugin



# Plugin script

---

Les plugins script sont résolus à partir :

- D'un emplacement relatif au répertoire projet ou à *buildSrc*
- D'une URL (HTTPS en général)

```
apply from: 'other.gradle'
```



# Plugin binaire

---

Les plugins binaires sont résolus et appliqués grâce à leur ***plugin id***.

- Les plugins cœur de *Gradle* fournissent des noms courts (comme *'java'*).
- Les autres fournissent un nom qualifié (comme *com.github.foo.bar*)



# Résolution

---

La résolution de l'emplacement d'un plugin peut être fait de 2 façons :

- Inclure le plugin à partir du portail *Gradle* ou d'un dépôt personnalisé en utilisant le DSL ***plugins***
- Inclure le plugin à partir d'un jar externe défini comme dépendance externe du script de build



# DSL *plugins*

---

L'utilisation du DSL ***plugins*** apporte certains avantages :

- Optimise le chargement et à la réutilisation des classes du plugin
- Permet à différents plugins d'utiliser différentes versions des dépendances
- Permet l'assistance lors de l'édition

La syntaxe est :

```
plugins {  
    id «plugin id» version «plugin version» [apply «false»]  
}
```

Aucun autre code n'est permis dans le bloc plugins

Ex :

```
plugins {  
    id 'java'  
    id 'com.jfrog.bintray' version '0.4.1'  
}
```



# Plugin Management

---

La résolution se fait du vis à vis du portail Gradle.

Il est possible de spécifier des dépôts spécifiques via le bloc ***repositories {}*** à l'intérieur d'un bloc ***pluginManagement {}*** dans *settings.gradle*

```
pluginManagement {  
    repositories {  
        maven {  
            url 'maven-repo'  
        }  
        gradlePluginPortal()  
        ivy {  
            url 'ivy-repo'  
        }  
    }  
}
```



# Legacy plugin

---

Si ce n'est pas possible d'utiliser DSL, on peut encore utiliser l'ancienne méthode qui n'est pas pré-compilée

Appliquer via un id

*apply plugin: 'java'*

Ou via la classe

*apply plugin: JavaPlugin*



# Jar externe

---

Les plugins binaires qui ont été packagés sous forme de *jar* peuvent être ajoutés au *classpath* du script de build puis appliqué. Il faut utiliser alors la méthode ***buildScript()*** Exemple :

```
buildscript {  
    repositories {  
        jcenter()  
    }  
    dependencies {  
        classpath "com.jfrog.bintray.gradle:gradle-bintray-plugin:0.4.1"  
        classpath files('relative/path/to/plugin.jar')  
    }  
}
```

```
apply plugin: "com.jfrog.bintray"  
apply plugin: "com.myCompany.ourPlugin"
```





# Écriture de plugin

---

```
class GreetingPlugin implements Plugin<Project> {  
    void apply(Project project) {  
        project.task('hello') {  
            doLast {  
                println 'Hello from the GreetingPlugin'  
            }  
        }  
    }  
}  
  
// Apply the plugin  
apply plugin: GreetingPlugin
```



# Plugin *init*

---

Le plugin ***init*** automatiquement disponible ajoute systématiquement les tâches :

- ***init*** : Permet la création ou la conversion de différents types de projet
- ***wrapper*** : Permet de créer des scripts shell ou bat qui encapsule l'appel à Gradle.



# Wrapper

---

L'utilisation d'un ***wrapper*** permet

- Qu'un développeur puisse exécuter un script Gradle sans avoir à préalablement installer le runtime le wrapper télécharge automatiquement le runtime Gradle à son premier lancement et le stocke localement *\$HOME\_DIR/.gradle/wrapper/dists*
- Cela garantit également que le build est exécuté avec une version spécifique de Gradle.

=> L'objectif étant de créer des build reproductibles et surs quelque soit l'OS, ou la version de *Gradle* installée manuellement



# Tâches *init*

---

*init* supporte différents type de mise en place via l'argument *--type*

- pom (Maven conversion)
- java-application, java-library
- Scala-library
- *groovy-library, groovy-application*
- basic



# Java et C++

---

## **Gestion des dépendances**

Plugin Java

Plugin C++



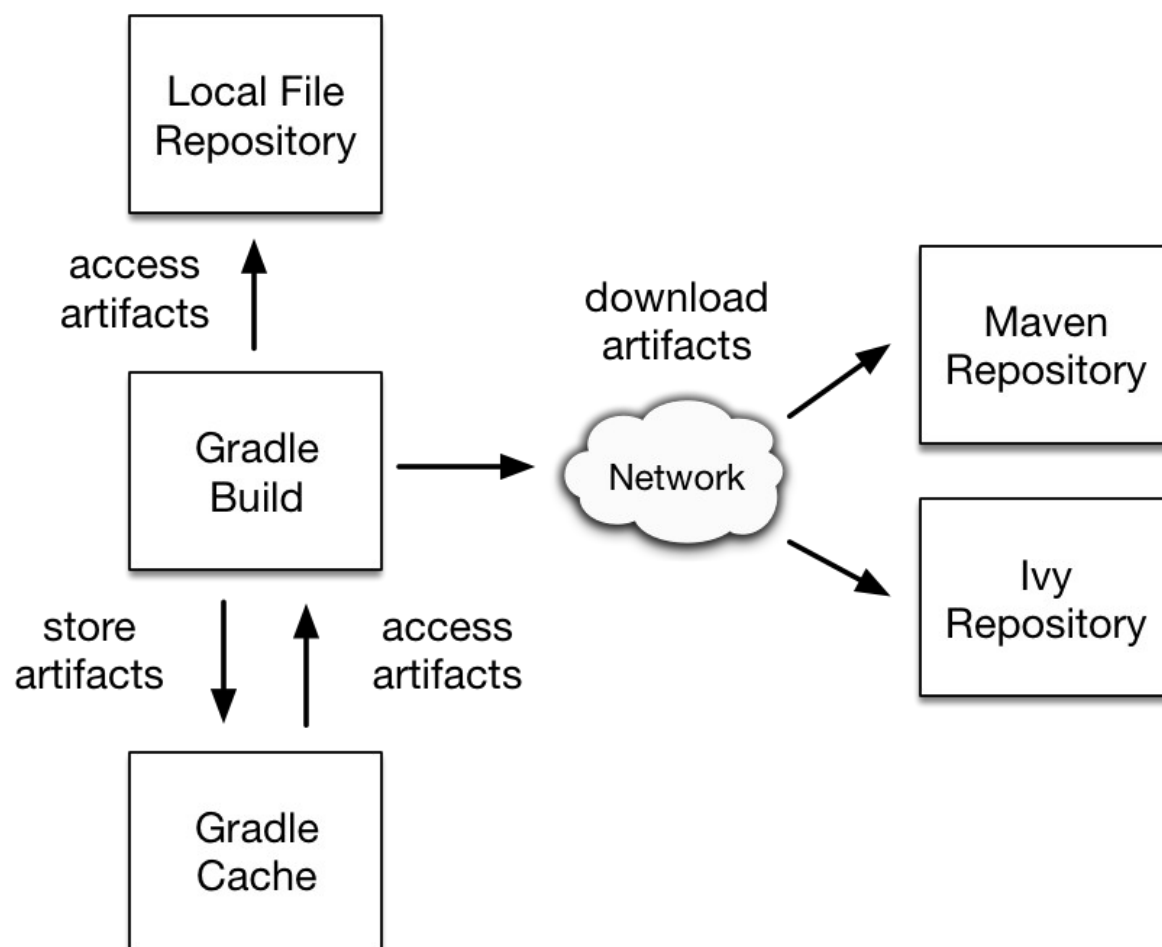
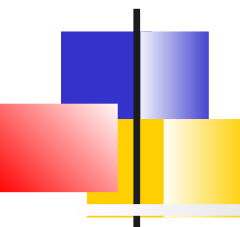
# Introduction

---

L'ajout de dépendances vers des librairies externes s'effectue en 2 étapes :

- Avec la closure ***dependencies***, on définit les libraires dont on dépend ainsi que leur *scope* ou *configuration*
- Avec la closure ***repositories***, on indique l'origine de ces dépendances .

*Gradle* résout alors automatiquement les dépendances, les télécharge et les stocke dans son cache local





# Configuration

---

Les **configurations** définissent le périmètre d'une dépendance.

Chaque projet contient une classe *ConfigurationContainer* qui gère les configurations d'un projet.

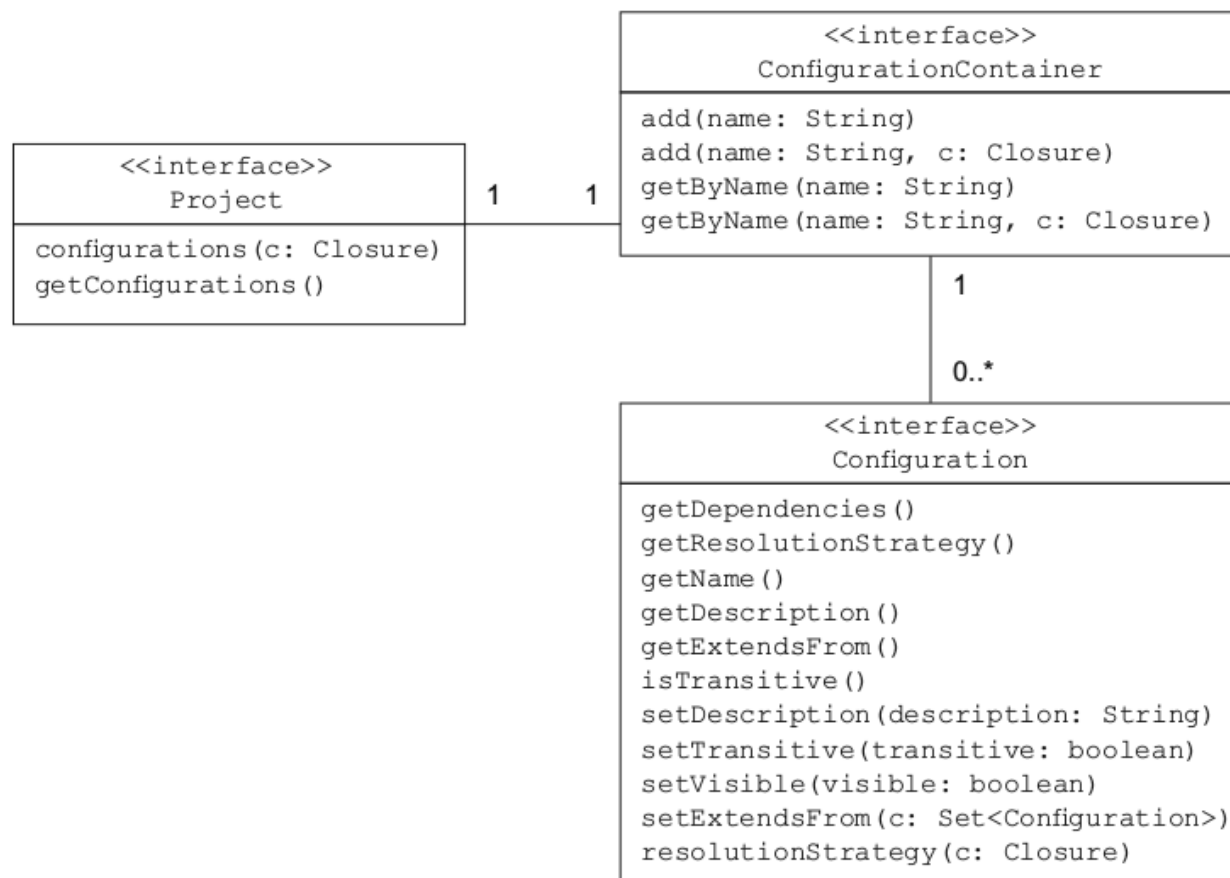
Une configuration regroupe les dépendances pour une fonctionnalité spécifique.

Par exemple, le plugin Java fournit différentes configurations : *compileOnly*, *implementation*, *runtime*, *testImplementation*, *testRuntime*, *archives* ...





# Interface Configuration





# Définir une configuration

---

On peut définir ses propres configurations via le DSL ***configurations***.

Une configuration consiste d'un nom et d'une visibilité (cas multi-projets)

La configuration peut ensuite être accédée via son nom à partir de la variable ***configurations*** pour référencer un classpath par exemple

.



# Exemple

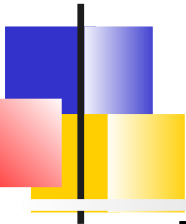
---

```
configurations {
    jasper
}

repositories {
    mavenCentral()
}

dependencies {
    jasper 'org.apache.tomcat.embed:tomcat-embed-jasper:9.0.2'
}

task preCompileJsps {
    doLast {
        ant.taskdef(classname: 'org.apache.jasper.JspC',
                    name: 'jasper',
                    classpath: configurations.jasper.asPath)
        ant.jasper(validateXml: false,
                    uriroot: file('src/main/webapp'),
                    outputDir: file("$buildDir/compiled-jsps"))
    }
}
```



# Héritage des configurations

---

Une configuration peut étendre une autre configuration.

Les configurations filles héritent de toutes les dépendances de ses parents.

```
configurations {  
    smokeTest.extendsFrom testImplementation  
}  
  
dependencies {  
    testImplementation 'junit:junit:4.12'  
    smokeTest 'org.apache.httpcomponents:httpclient:4.5.5'  
}
```



# Bloc *dependencies*

---

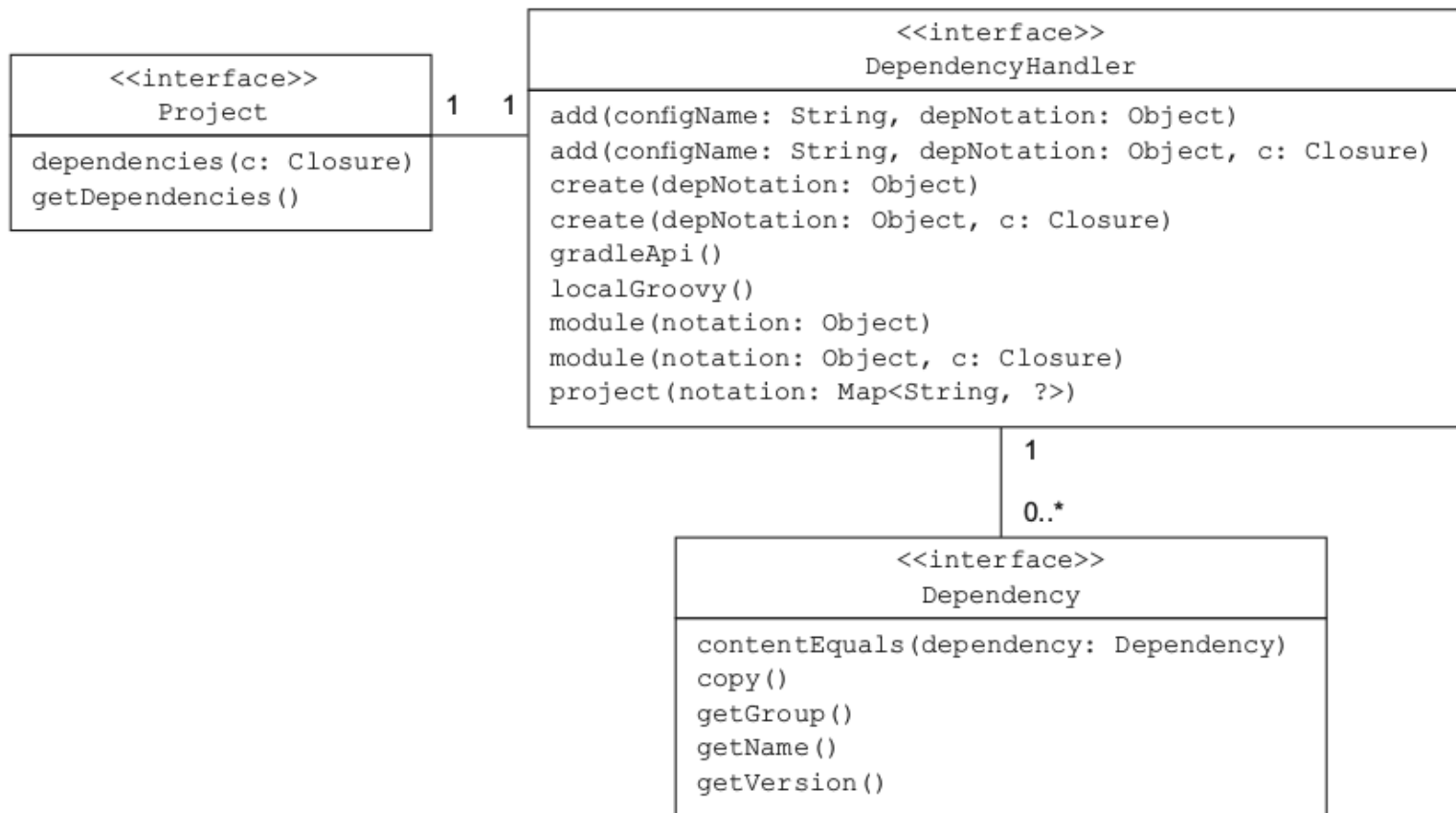
Le bloc ***dependencies*** est utilisé pour affecter une ou plusieurs dépendances à une configuration

Les dépendances peuvent être exprimées vis à vis :

- D'un artefact d'un dépôt
- D'un projet
- D'un ensemble de fichiers du système de fichiers



# Interface *Dependency*





# Coordonnées d'une dépendance

---

Une dépendance est localisée via les attributs suivants :

- **group** : Habituellement le nom de domaine d'une organisation.
- **name** : Le nom de l'artefact.
- **version** : Une chaîne de caractère composée d'une version majeure et mineure
- **classifier** : Éventuellement, pour distinguer les artefacts avec les mêmes groupes, nom et version

La syntaxe est alors :

```
dependencies {  
    configurationName dependencyNotation1,  
    dependencyNotation2, ...  
}
```



# Version dynamique

---

La déclaration de versions dynamique a plusieurs syntaxes :

**// Dernière version**

org.codehaus.cargo:cargo-ant:latest-integration

**// Dernière Version 1.x**

org.codehaus.cargo:cargo-ant:1.+

**// Version comprise entre 1.0 et 2.0 inclus**

org.codehaus.cargo:cargo-ant:[1.0,2.0]

**// Version comprise entre 1.0 inclus et 2.0 exclus**

org.codehaus.cargo:cargo-ant:[1.0,2.0[

**// Version à partir de 1.0**

org.codehaus.cargo:cargo-ant:[1.0, )





# Rapport de dépendances

---

Lorsque l'on exécute :

***gradle dependencies***

L'arbre complet de dépendances est affiché

On y voit pour chaque configuration les dépendances directes et les dépendances transitives

Les dépendances omises sont marquées d'un astérisque.

- Cela est dû à une double dépendance vers le même artefact.
- Par défaut, *Gradle* choisit la version la plus haute en cas de disparité dans les versions



# Exclure des dépendances transitives

La méthode ***exclude*** avec comme paramètre une *Map* permet d'exclure des dépendances transitives

Exemple :

```
dependencies {  
    cargo('org.codehaus.cargo:cargo-ant:1.3.1') {  
        exclude group: 'xml-apis', module: 'xml-apis'  
    }  
    cargo 'xml-apis:xml-apis:2.0.2'  
}
```

L'attribut ***transitive*** permet d'exclure toutes les dépendances transitives

```
dependencies {  
    cargo('org.codehaus.cargo:cargo-ant:1.3.1') {  
        transitive = false  
    }  
    // Déclarer explicitement toutes les dépendances nécessaires  
}
```



# Contraintes sur les dépendances et les dépendances transitives

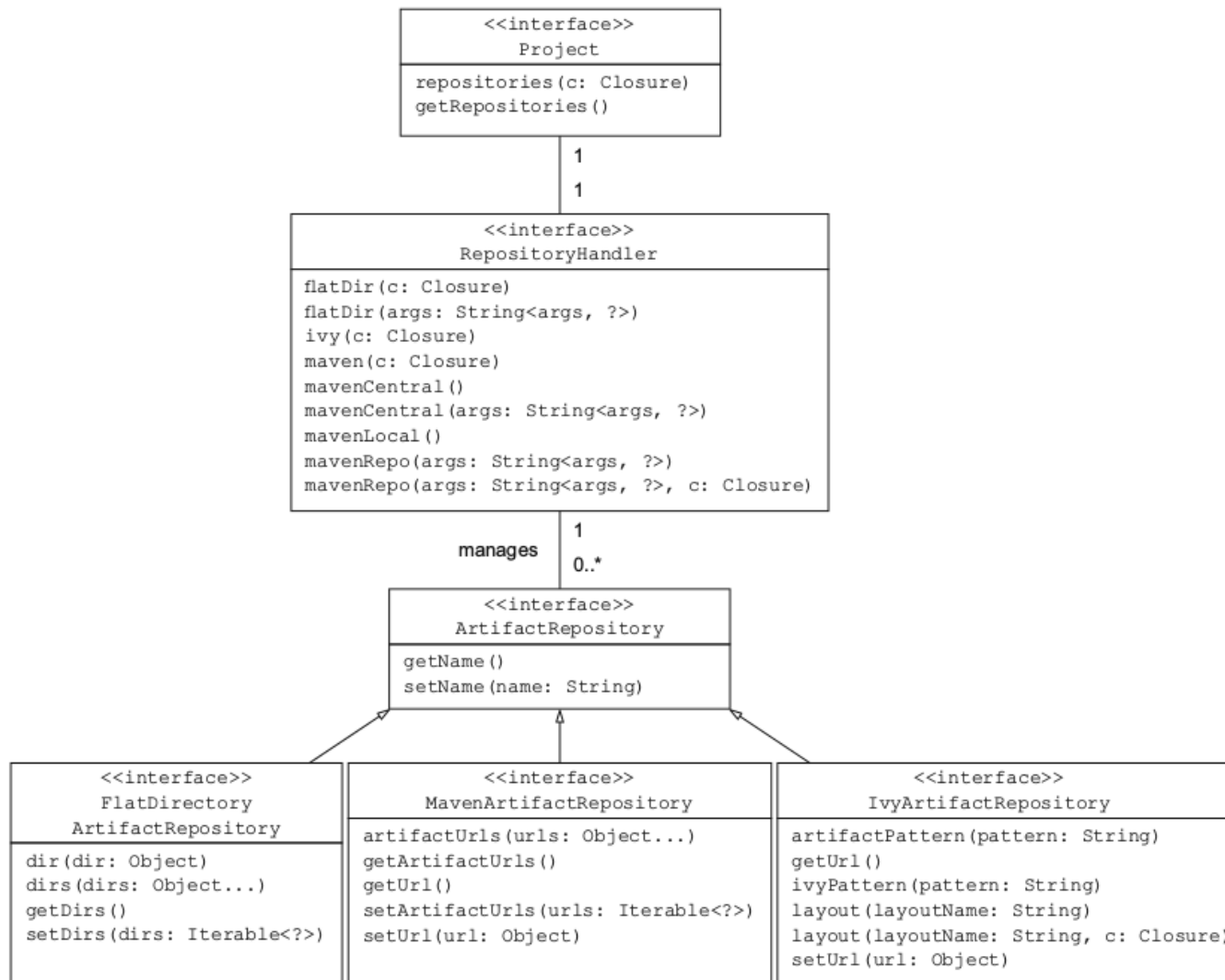
---

*Gradle* en cas de latitude sur une version favorise toujours la version la plus grande.

Si ce comportement n'est pas voulu, on peut fixer des contraintes sur les dépendances directes et transitives

```
dependencies {  
    implementation 'org.apache.httpcomponents:httpclient'  
    constraints {  
        implementation('org.apache.httpcomponents:httpclient:4.5.3') {  
            because 'previous versions have a bug impacting this application'  
        }  
        implementation('commons-codec:commons-codec:1.11') {  
            because 'version 1.9 pulled from httpclient has bugs affecting this application'  
        }  
    }  
}
```

# Interface *Repository*





# Dépôts

---

*Gradle* supporte les dépôts les plus habituels (maven, google, ...)

En dehors des dépôts centraux pré-configurés, il est possible de fournir une URL arbitraire, une référence vers le système de fichier local, des informations d'authentification

Dans le bloc *repositories*, on peut déclarer plusieurs dépôts, ils seront interrogés par Maven dans leur ordre de déclaration.



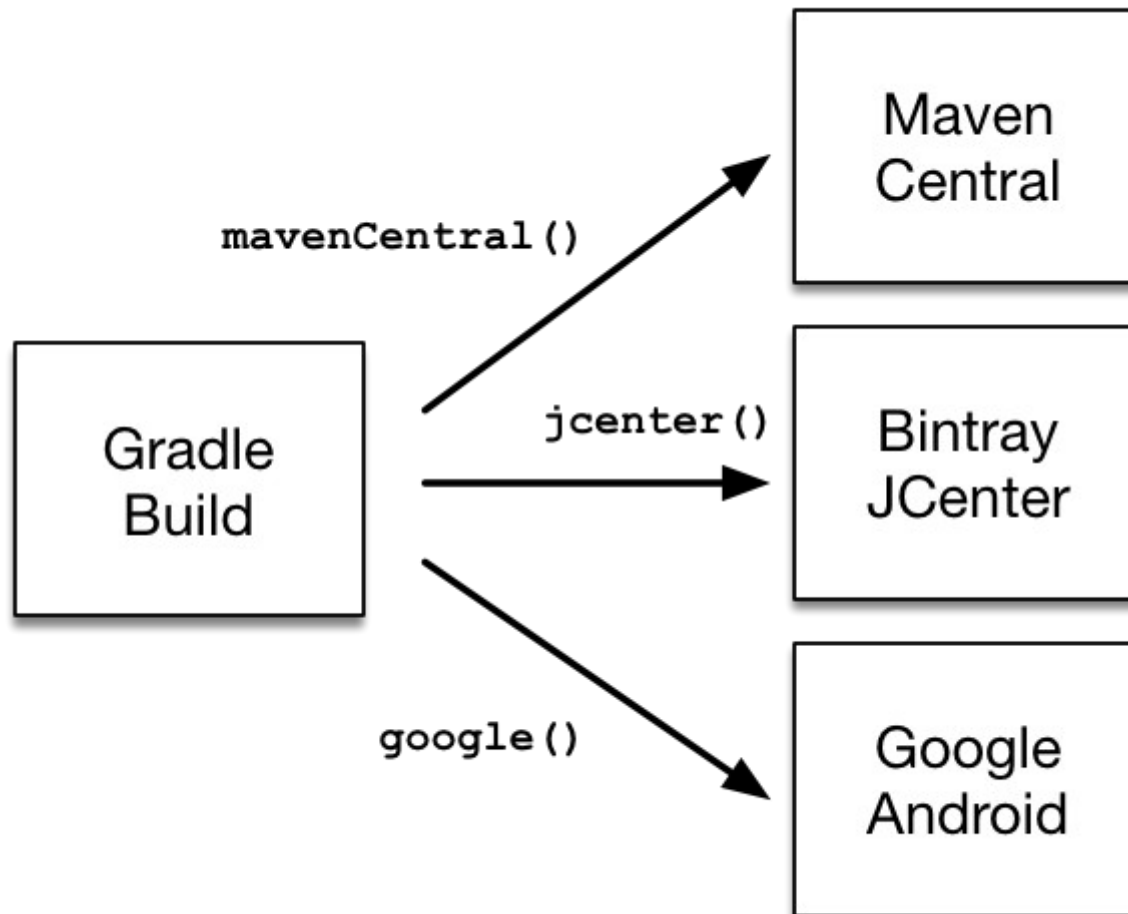
# Dépôts publics classiques

---

L'interface *RepositoryHandler* fournit plusieurs méthodes permettant de définir les dépôts habituels :

- Maven
  - `mavenCentral()`
  - `mavenLocal()`
- Bintray JCenter
  - `jcenter()`
- Dépôt de Google Android
  - `google()`

# Raccourci





# Dépôt Maven custom

---

L'API Gradle permet de configurer un repository Maven via la méthode ***maven()***

Exemple

```
repositories {  
    maven {  
        credentials {  
            username = 'joe'  
            password = 'secret'  
        }  
        url  
        "http://localhost:8081/nexus/content/groups/public"  
    }  
}
```





# Cache Gradle

---

Le répertoire racine pour stocker les artefacts en local est :

`<USER_HOME>/ .gradle/caches`

2 types de stockage sont gérés :

- Basé sur les fichiers (jars ou méta-données comme *pom.xml*). Le chemin de stockage contient un checksum SHA1 .
- Un stockage binaire contenant le résultat de la résolution de versions dynamiques et autres.



# Java et C++

---

Gestion des dépendances

**Plugin Java**

Plugin C++



# Introduction

---

Le plugin **Java** ajoute les tâches de compilation et de tests d'un projet.

Il sert de base pour de nombreux autres plugins *Gradle*

```
plugins {  
    id 'java'  
}
```



# Structure projet par défaut

---

***src/main/java*** : Source de production  
Java

***src/main/resources*** : Ressources de  
production

***src/test/java*** : Classes de test

***src/test/resources*** : Ressources de test



# Répertoires

---

***reporting.baseDir*** : Le nom répertoire pour générer des rapport  
(*reports*)

***reportsDir*** : Le répertoire résolu ou sont générer les rapports  
(*buildDir/reports*)

***testResultsDir*** Le répertoire résolu ou sont générer les résultats de test  
en .xml (*buildDir/test-results*)

***testReportDir*** : Le répertoire pour générer le rapport de test.  
(*reportsDir/tests*)

***libsDir*** : Le répertoire pour générer les librairies (*buildDir/libs*)

***distsDir*** Le répertoire pour générer la distribution. (*buildDir/distributions*)

***docsDir*** Le répertoire pour générer la documentation (*buildDir/docs*)

***dependencyCacheDirName*** : le nom du répertoire pour cacher les  
informations de dépendances des sources (*dependency-cache*)



# Autres propriétés

---

***sourceSets*** : Tous les sourceSets. (Ensemble de sources *main* et *test* par défaut)

***sourceCompatibility*** : Version pour le compilateur. Par défaut la version de la JVM courante. Supporte des String ou Nombres

***targetCompatibility*** : Version pour la génération de classe. Par défaut *sourceCompatibility*

***archivesBaseName*** : Le nom de base pour générer les archives (JAR ou ZIP). Par défaut le nom du projet.

***manifest*** : Le manifeste à inclure dans le JAR. Par défaut vide.



# Tâches de build

---

***assemble(type: Task)*** : Assemble toutes les archives d'un projet.

***build(type: Task)*** : Build du projet : Assemble + test.

***buildDependents(type: Task)*** : Effectue un build complet du projet et de tous les projets qui en dépendent.

***buildNeeded(type: Task)*** : Effectue un build complet du projet et de tous les projets dont il dépend. Dépend de *build* et *buildNeeded* dans les dépendances projet.

***classes(type: Task)*** Assemble les répertoire des classes de production et de ressources.

***clean(type: Delete)*** : Supprime le répertoire de build .

***jar(type: Jar)*** : Assemble le fichier JAR.

***testClasses(type: Task)*** : Assemble les répertoires des classes et ressources de test.



# Autres groupes

---

2 autres groupes sont définis :

## Documentation

***javadoc(type:Task)*** : Génération de la documentation HTML

## Vérification

***check(type:Task)*** : Effectue tous les tests

***test(type:Task)*** : Effectue les tests unitaires





# Autres tâches

---

***compileJava(type: JavaCompile)*** Compile les fichiers sources de production avec *javac*. Dépend de toutes les tâches produisant le classpath (la tâche *jar* des dépendances).

***processResources(type: Copy)*** Copies les ressources de production dans le répertoire *resources*.

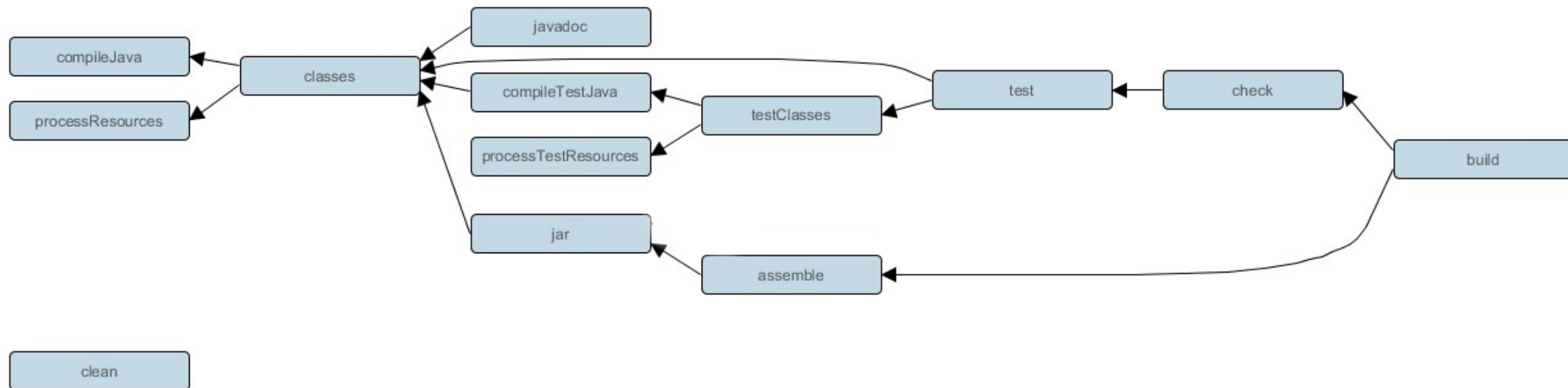
***compileTestJava(type: JavaCompile)*** Compiles les classes de test avec *javac*. Depend de *compile*, plus toutes les tâches produisant le classpath de *testCompile*.

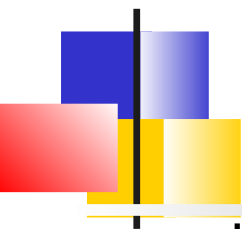
***processTestResources(type: Copy)*** Copie les ressources de test dans le répertoire *test resources directory*.

***clean<TaskName>(type: Delete)*** : Supprime les sorties d'une tâche

***build<TaskName>(type: Delete)*** : Assemble les artefacts d'une configuration

# Dépendances





# Configurations prédéfinies

---

Le plugin Java ajoute des configurations de dépendances

- ***annotationProcessor*** : Processeurs d'annotations utilisés pendant la compilation pour générer des sources
- ***implementation*** : Les dépendances de compilation,
- ***compileOnly*** : Les dépendances présentes seulement à la compilation
- ***compileClasspath*** *extends compileOnly, implementation* : Classpath utilisé à la compilation
- ***testImplementation*** *extends implementation*: Dépendances additionnelles pour compiler les tests.
- ***testCompileOnly*** : Dépendances additionnelles pour compiler les tests mais pas les exécuter
- ***testCompileClasspath*** *extends testImplementation, testCompileOnly* : Classpath utilisé par la tâche *compileTestJava*.



# Configurations prédéfinies (2)

---

***runtimeOnly*** : Les dépendances à l'exécution

***runtimeClasspath extends runtimeOnly, implementation*** :  
Implementation + runtime.

***testRuntimeOnly*** : Dépendances additionnelles pour exécuter les tests.

***testRuntimeClasspath*** *extends testRuntimeOnly,*  
*testImplementation* : Classpath d'exécution pour l'exécution des tests.



# *sourceSet*

---

Le plugin java ajoute donc 2 ***sourceSet*** :

- ***main*** : Le code de production assemblé dans un JAR. Le *sourceSet* par défaut
- ***test*** : Le code de test compilé et exécuté avec JUnit ou TestNG. Typiquement des tests unitaires.

Il est possible de :

- changer le layout des 2 *sourceSet* prédéfinis
- de définir d'autres *sourceSet*.



# Exemple : Chgt de layout

---

```
sourceSets {  
    main {  
        java {  
            srcDirs = ['src/java']  
        }  
        resources {  
            srcDirs = ['src/resources']  
        }  
    }  
}
```



# Exemple : ajout de *sourceSet*

---

```
sourceSets {
    intTest {
        java.srcDir file('src/intTest/java')
        resources.srcDir file('src/intTest/resources')
        compileClasspath += sourceSets.main.output +
configurations.testRuntime
        runtimeClasspath += output + compileClasspath
    }
}

configurations {
    intTestImplementation.extendsFrom implementation
}

dependencies {
    intTestImplementation 'junit:junit:4.12'
}
```



# Interface *SourceSet*

---

***name*** : Le nom

***java.srcDirs*** : Le répertoire source.

***resources.srcDirs*** : Le répertoire ressource

***output.classesDirs*** : Le répertoire pour générer les classes

***output.resourcesDir*** : Le répertoire pour généré les  
ressources

***compileClasspath*** : Le classpath de compilation.

***runtimeClasspath*** : Le classpath d'exécution .

***allJava*** : Tous les fichiers *.java* et les fichiers résultants des  
fichiers Groovy/Scala

***allSource*** : Toutes les sources





# Configurations pour un *sourceSet*

---

**<sourceSet>Implementation** : Dépendances de compilation pour le sourceSet spécifié

**<sourceSet>CompileOnly** : Compilation mais pas exécution

**<sourceSet>CompileClasspath** extends  
    <sourceSet>Implementation, <sourceSet>CompileOnly

**<sourceSet>AnnotationProcessor** : Processeur d'annotation utilisé durant la compilation

**<sourceSet>RuntimeOnly** : Dépendances exclusivement runtime pour le source set spécifié.

**<sourceSet>RuntimeClasspath** extends  
    sourceSetRuntimeOnly, sourceSetImplementation : Classpath résolu lors de l'exécution pour ce sourceSet



# Tâches liées à un *sourceSet*

***compile<SourceSet>Java(type: JavaCompile)*** :  
Compile le *sourceSet* spécifié.

***process<SourceSet>Resources(type: Copy)*** :  
Copie les ressources du *sourceSet* spécifié dans le répertoire des ressources.

***<sourceSet>Classes(type: Task)*** : Assemble les répertoire des classes de production et de ressources du *sourceSet* spécifié.



# Plugin Java library

---

Le plugin ***java-library*** étend le plugin Java

Une librairie est un composant Java consommé par d'autres composants.

Le plugin offre alors 2 nouvelles configurations de dépendances :

- ***api*** : Les classes publiques
- ***implementation*** : Les classes internes



# Plugin *application*

---

Le plugin ***application*** permet de désigner la classe contenant la méthode main pouvant être exécutée en ligne de commande.

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

```
mainClassName = 'App'
```



# Application web

---

Gradle fournit le plugin **war** ; un plugin communautaire **gretty** permet de tester et déployer une application sur *Jetty* ou *Tomcat*

Le plugin *War* étend le plugin *Java* en remplaçant la génération du JAR par la génération d'un WAR.

La structure par défaut du projet contient un répertoire ***src/main/webapp*** contenant les ressources web du projet

```
plugins {  
    id 'war'  
    id 'org.akhikh1.gretty' version '1.4.2'  
}
```



# Java et C++

---

Gestion des dépendances  
Plugin Java  
**Plugin C++**



# Introduction

---

*Gradle* supporte les outils *Clang*, *GCC* et *Visual C++*

Il propose plusieurs plugins :

- ***cpp-application*** : Pour les applications
- ***cpp-library*** : Pour les bibliothèques
- ***cpp-unit-test*** : Pour les tests unitaires



# Tâches des plugins

---

Les plugins ***cpp-application*** et ***cpp-library*** ajoutent les tâches de compilation, de link et d'assemblage :

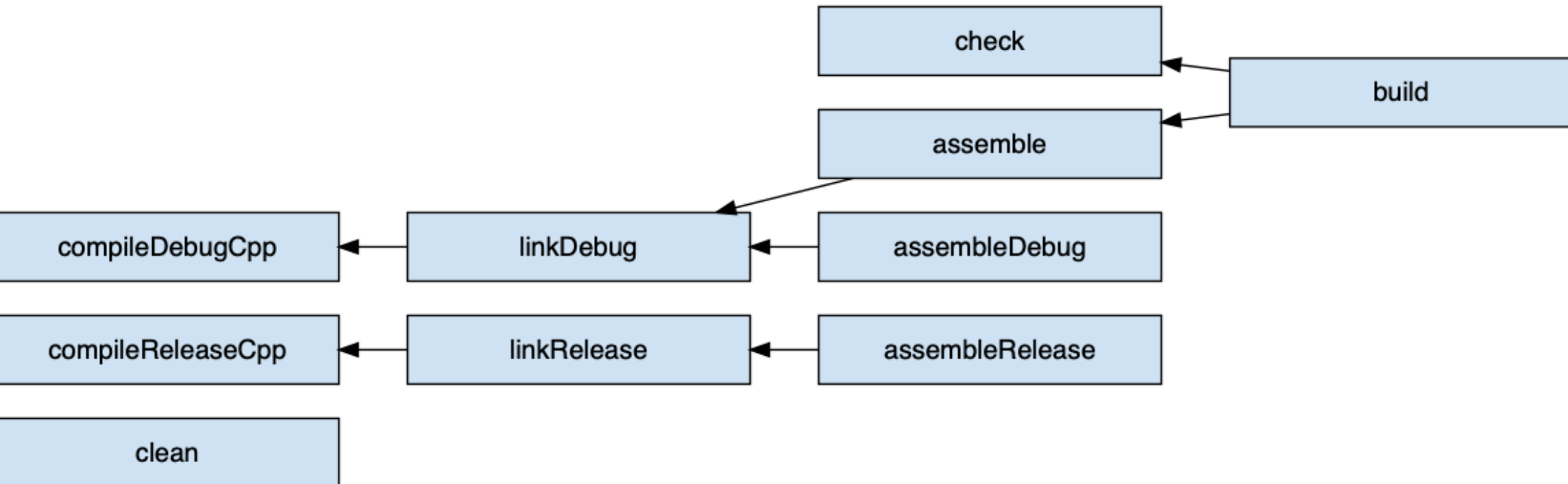
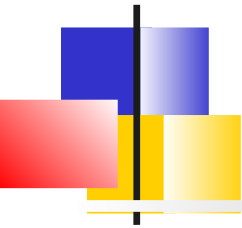
- ***compileDebugCpp*** et ***compileReleaseCpp*** qui compilent les fichiers source sous *src/main/cpp*.
- ***linkDebug*** et ***linkRelease*** qui lie les fichiers d'objets compilés en un exécutable.
- ***createDebug*** et ***createRelease*** qui assemblent les fichiers objets C ++ compilés dans une bibliothèque statique

Ces tâches sont intégrées aux tâches du cycle de vie standard.

Par exemple, la tâche produisant le binaire final est associée à ***assemble***. Par défaut, c'est la version de debug



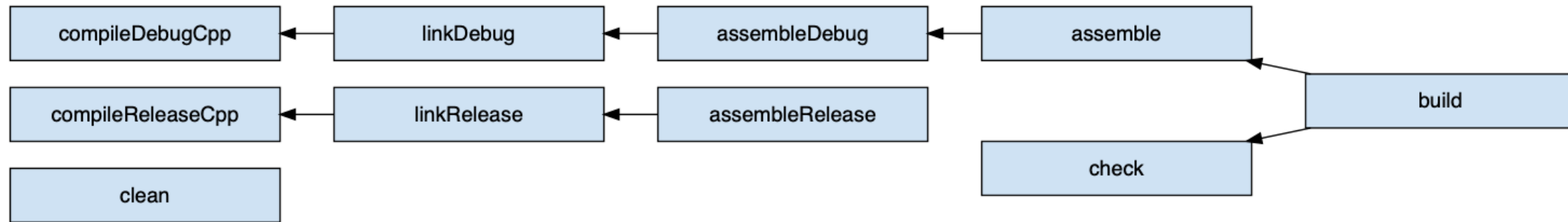
# Graphe de dépendances des tâches



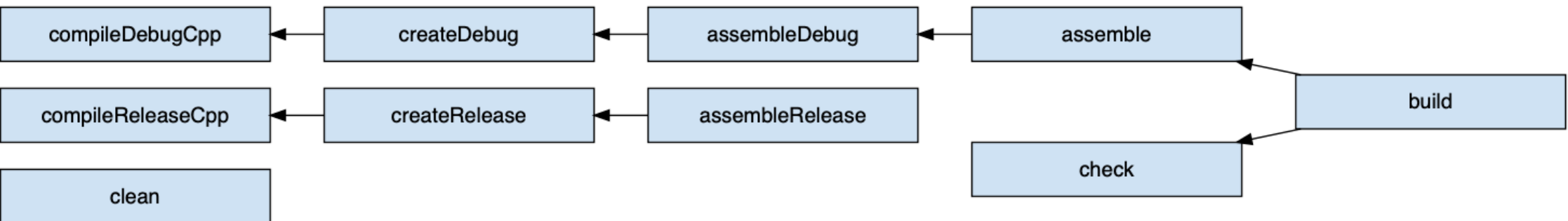


# Graphe de dépendances pour bibliothèques

## Librairie partagée



## Librairie statique





# Différence entre library et application

---

Les projets **library** sont consommées par d'autres projets C++

Les méta-données de dépendances sont publiées avec les binaires et les entêtes

Elles distinguent les dépendances :

- qui sont uniquement nécessaires pour compiler la bibliothèque
- De celles qui sont également requises pour compiler le consommateur : configuration **api**

Les projets **application** sont destinés à produire des exécutables

Le plugin permet :

- Une tâche **install** qui crée un répertoire permettant d'exécuter l'application
- Des scripts permettant de démarrer l'application



# Variantes de build

---

Généralement, les projets natifs produisent différents binaires (plateforme, architecture processeur, debug, ...)

*Gradle* gère cela à travers les concepts de **dimensions** et de **variantes**.

- Une *dimension* est simplement une caractéristique différenciante du build.  
Par exemple, pour les applications C++  
"build type" : *debug* ou *release*  
"Machine Cible" : *linux.x86\_64*, *windows.x86*
- Une variante est une combinaison de valeurs de dimension.  
Par exemple : *windows.x86/Debug*

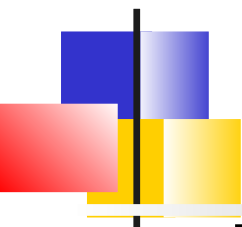


# Machines cibles

---

Par défaut, *Gradle* crée un binaire pour l'OS hôte et son architecture. Il est possible de remplacer cela en spécifiant un ensemble de machines cible dans les blocs *application* ou *library*:

```
application {  
    targetMachines = [  
        machines.linux.x86_64,  
        machines.windows.x86, machines.windows.x86_64,  
        machines.macOS.x86_64  
    ]  
}
```



# Configuration des sources

---

Par défaut, les sources sont présents dans :

- ***src/main/cpp***
- Les entêtes privées dans ***src/main/headers***
- Pour les librairies, les entêtes publiques dans ***src/main/public***

Les emplacement peuvent être modifiés dans les blocs  
*application* ou *library*

Exemple :

```
library {  
    source.from file('src')  
    privateHeaders.from file('src')  
    publicHeaders.from file('include')  
}
```

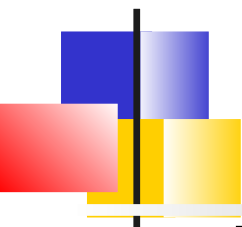


# Dépendances

---

*Gradle* permet de définir des dépendances vers des référentiels Maven ou vers un autre projet dans les blocs *application* ou *library*

```
application {  
    dependencies {  
        implementation project(':common')  
        implementation 'org.gradle.cpp-samples:list:1.5'  
    }  
}
```



# Configurations

---

Les principales configurations pour les dépendances sont :

- ***implementation*** : Compilation, linking et exécution
- ***cppCompileVariant*** : Dépendances nécessaires pour la compilation mais pas pour le link et l'exécution
- ***nativeLinkVariant*** : Seulement pour le link
- ***nativeRuntimeVariant*** : Seulement pour l'exécution
- ***api*** : (Apporté par le plugin *cpp-library*) Dépendances nécessaires pour la compilation et le link du module courant et des modules qui dépendent du module courant.





# Autres cas d'usage

---

Définition d'un référentiel personnalisé compatible Maven

Déclarer des dépendances avec des versions changeantes (par exemple SNAPSHOT) et dynamiques (plage)

Contrôle des dépendances transitives et de leurs versions

Test de vos correctifs pour la dépendance tierce via des [build composites](#).



# Compilation et link

---

La compilation et le link d'un projet C++ est facile si :

- On place le code source dans ***src/main/cpp***
- On déclare les dépendances dans le scope ***implementation***
- On exécute la tâche ***assemble***



# Options de compilation et de link

---

Les options de compilation et du link accessibles via les tâches *compileVariantCpp*, *linkVariant* et *createVariant* qui sont de type *CppCompile*, *LinkSharedLibrary* et *CreateStaticLibrary* respectivement.

```
tasks.withType(CppCompile).configureEach {  
  
    // Une option de compilation  
    compilerArgs.add '-W3'  
  
    // Des options spécifiques à certains outils  
    compilerArgs.addAll toolChain.map { toolChain ->  
        if (toolChain in [ Gcc, Clang ]) {  
            return ['-O2', '-fno-access-control']  
        } else if (toolChain in VisualCpp) {  
            return ['/Zi']  
        }  
        return []  
    }  
}
```



# Accès aux variantes

---

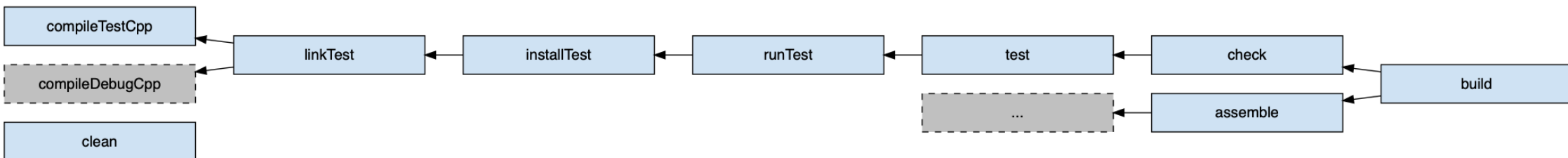
Il est également possible de trouver l'instance pour une variante spécifique via ***binaries*** de type BinaryCollection dans un bloc *application* ou *library* :

```
application {  
    binaries.configureEach(CppStaticLibrary) {  
        // Définir une option de compilation  
        compileTask.get().compilerArgs.add '-W3'  
  
        // En fonction de la toolChain  
        if (toolChain in [ Gcc, Clang ]) {  
            compileTask.get().compilerArgs.addAll(['-O2', '-fno-access-control'])  
        } else if (toolChain in VisualCpp) {  
            compileTask.get().compilerArgs.add('/Zi')  
        }  
    }  
}
```



# Test

Le plugin ***cpp-unit-test*** fournit les tâches, les configurations et les conventions pour l'intégration avec un framework de test tel que Google Test.





# Bloc unit-test

---

Le plugin ajoute le bloc ***unitTest*** dans lequel on peut définir :

- Les machines cibles
- Les dépendances
- ...

Exemple :

```
unitTest {  
    targetMachines = [  
        machines.linux.x86_64,  
        machines.windows.x86, machines.windows.x86_64,  
        machines.macOS.x86_64  
    ]  
}
```



# Packaging et publication

---

*Gradle* fournit le plugin *maven-publish* qui permet de :

- Publier les exécutables dans des référentiels Maven.
  - Les librairies partagées et statiques sont publiées dans les référentiels Maven avec un zip des entêtes publics.

Pour les applications, *Gradle* permet l'installation et l'exécution de l'exécutable avec toutes ses dépendances de bibliothèque partagée dans un emplacement spécifié.



# Multi-projets





# Structure

---

Un multi-projets consiste de plusieurs répertoires dont un répertoire contient le projet parent.

2 dispositions sont possibles :

- Arborescente : les sous-projets sont des sous-répertoires du projet parent
- A plat : le projet parent et les sous-projets sont au même niveau, le projet parent s'appelle *master*

Le projet parent coordonne la construction des sous-projets et peut définir des comportements communs ou spécifiques aux sous-projets

Il contient également l'unique répertoire de code de build *buildSrc*



# Phase de configuration

---

La phase de configuration d'un projet consiste à exécuter le script *build.gradle*

- Par défaut, la configuration de tous les sous-projets survient avant l'exécution de toute tâche (même si on ne désire qu'exécuter une seule tâche d'un sous-projet). Pour de gros projets, ceci peut être pénalisant.

A partir de Gradle 1.4, le mode **configuration à la demande** a été introduit. Il a pour but de ne configurer que les projets nécessaires à l'exécution d'une tâche particulière.

- Propriété de *gradle.properties* :  
`org.gradle.configureondemand=true`
- A terme seul ce mode sera effectif par défaut



# Déclaration des sous-projets

---

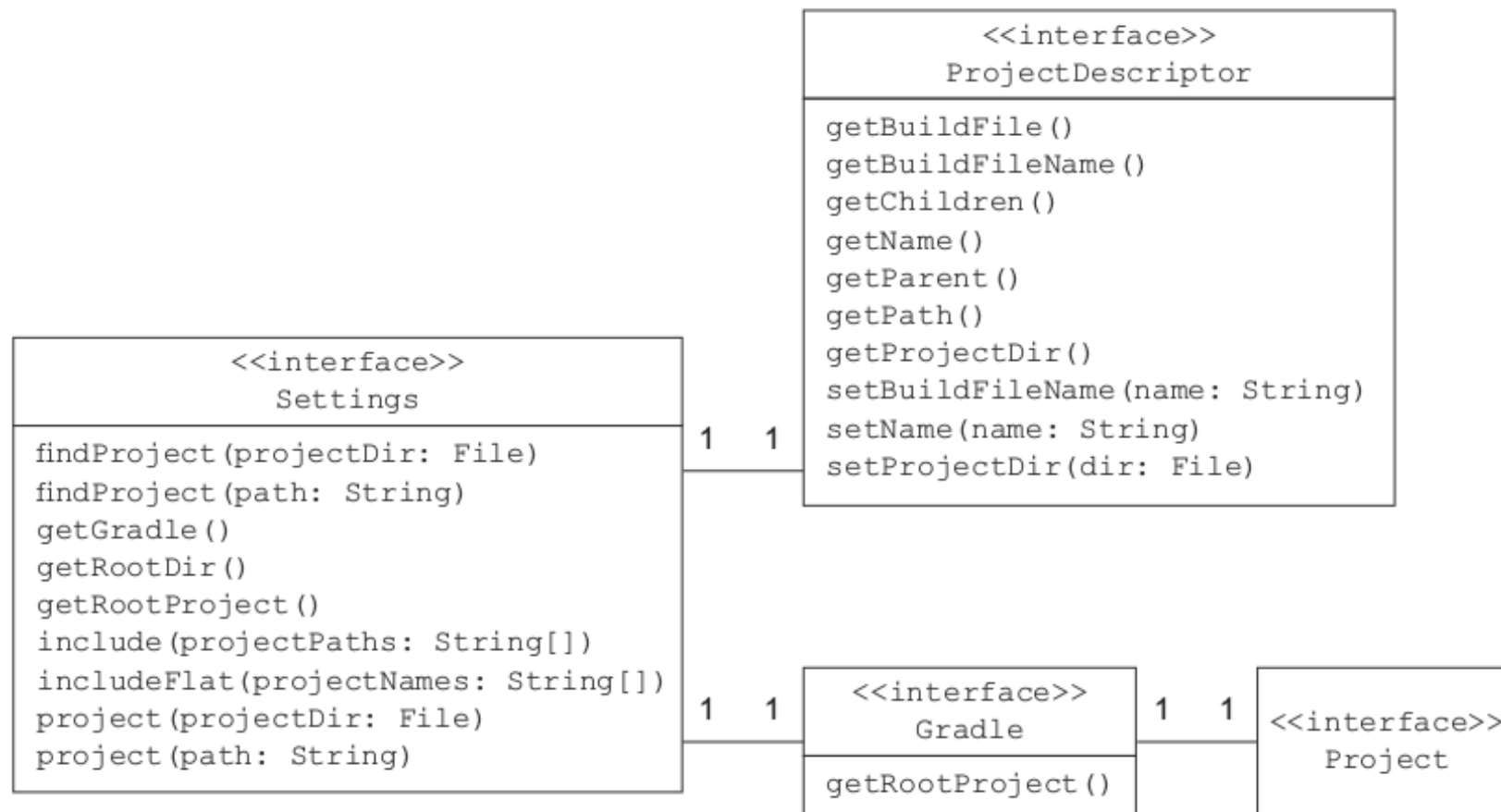
Le fichier ***settings.gradle*** dans le répertoire du projet parent liste les sous-projets via la méthode ***include***

```
include 'model'  
include 'repository', 'web'
```

La ligne de commande *gradle projects* liste l'arborescence projet

*settings.gradle* est accessible  
programmatically via la variable ***settings***

# Interface *Settings*





# Résolution des *Settings*

---

*Gradle* permet d'exécuter le build à partir du projet racine ou des répertoires des sous-projets.

*Gradle* recherche le fichier *settings* en 2 étapes :

- 1ère étape il recherche un répertoire nommé *master* au même niveau que le répertoire d'exécution
- Sinon, il remonte dans l'arborescence



# Comportement commun

---

L'API *Project* offre les méthodes ***allprojects*** et ***subprojects*** permettant de définir les comportements communs aux sous-projets.

Exemple :

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



# Comportement spécifique

---

Les comportements spécifiques peuvent être spécifiés dans le fichier *build.gradle* des sous-projets mais également dans le projet racine avec la méthode ***project***

```
project(':bluewhale').hello {  
    doLast {  
        println "- I'm the largest animal on this planet."  
    }  
}
```



# Autre exemple

---

## **build.gradle**

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

## **bluewhale/build.gradle**

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

## **gradle -q hello**

```
> gradle -q hello  
I'm water  
I'm bluewhale  
- I depend on water  
- I'm the largest animal on this planet.
```





# Méthode *configure()*

---

La méthode ***configure()*** prend comme argument une liste de projets et applique la configuration à tous les projets. Exemple :

```
configure(subprojects.findAll {it.name != 'tropicalFish'}) {  
    hello {  
        doLast {  
            println '- I love to spend time in the arctic  
waters.'  
        }  
    }  
}
```



# Configuration selon des propriétés

---

En utilisant les méthodes de callback d'un build, on peut configurer des projets après l'évaluation des propriétés.

Exemple :

```
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.' }  
            }  
        }  
    }  
}
```



# Règle d'exécution des tâches

---

Lors de l'indication d'une tâche, la ligne de commande *gradle* parcourt l'arborescence en partant du répertoire courant et exécute toutes les tâches du nom spécifié dans les différents sous-projets.

Pour exécuter une seule tâche d'un sous-projet, il faut indiquer le chemin absolu

Exemple :

```
$ gradle :model:build
```



# Dépendances entre sous-projets

---

Plusieurs types de dépendances existent entre les sous-projets

- Dépendance sur la configuration.  
Ex : Le parent qui injecte sa configuration chez les enfants
- Dépendance à l'exécution, influant sur l'ordre d'exécution des tâches
  - Dépendance implicite à cause des entrées/sorties des tâches
  - Dépendance explicite en utilisant *dependsOn*
  - Dépendance explicite, un sous-projet a besoin d'un autre pour un *classpath*

Si il n'y a aucune dépendance, Gradle exécute les tâches dans l'ordre alphabétique



# Dépendances vers un classpath d'un sous-projets

---

Déclarer une dépendance vers un sous-projet est similaire à la déclaration d'une dépendance vers une librairie. Il faut utiliser *dependencies* et préciser la configuration. Exemple :

**settings.gradle**

```
include 'web', 'repository', 'model'
```

**build.gradle**

```
project(':repository') {  
    dependencies {  
        implementation project(':model')  
    }  
}  
  
project(':web') {  
    dependencies {  
        implementation project(':repository'), project(':model')  
    }  
}
```

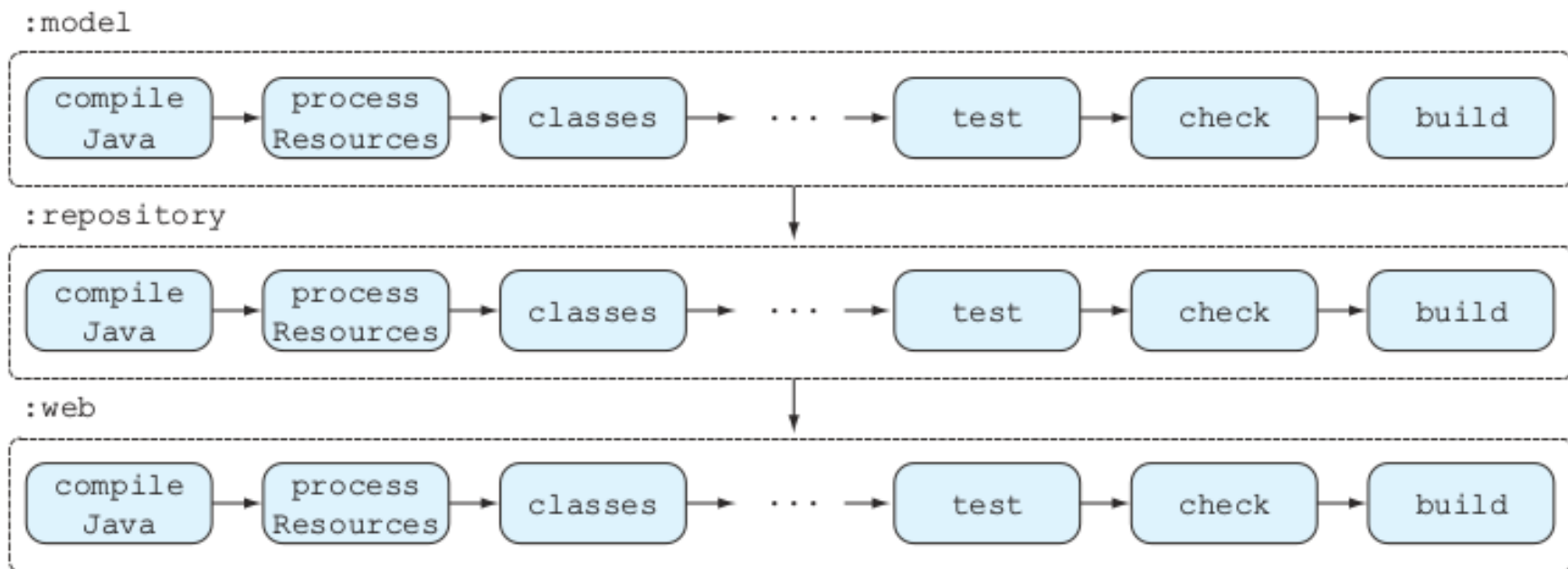


# Dépendances et exécution des tâches

Lors de l'exécution d'une tâche particulière, toutes les tâches du même nom des dépendances d'un sous-projet sont exécutées en premier.

Exécuter des tâches du projet racine, permet de s'assurer que toutes les classes dont les projets dépendent sont à jour.

*\$ gradle build*





# Build partiel

---

Pour des multi-projets volumineux, *Gradle* fournit une fonctionnalité nommée build partiel via l'option **-a** ou **-no-rebuild**

Cela permet à *Gradle* de ne pas vérifier si les projets dont on dépend doivent être reconstruits

Par exemple, si on sait que les classes de *model* n'ont pas changé

```
$ gradle :repository:build -a
```



# Forcer l'exécution des tests des dépendances

---

Autre cas, si on travaille seulement sur le projet *repository* et que l'on effectue  
`gradle :repository:build`

- Cela a pour effet de provoquer la compilation du projet dépendant *model* mais pas l'exécution des tests.

Pour cela :

**`gradle :repository:buildNeeded`**





# *buildDependents*

---

Avec la tâche ***buildDependents***, on peut vérifier l'impact de ses modifications en construisant et testant les projets dépendants.

Exemple :

```
gradle:repository:buildDependents
```



# Dépendances entre tâches des sous-projets

---

Il est possible de déclarer une dépendance vers une tâche d'un autre sous-projet. Cependant, cela introduit un couplage fort entre sous-projets.

La méthode recommandée est plutôt de déclarer la sortie de la tâche dont on veut dépendre comme une sortie du SourceSet main et d'effectuer une dépendance sur le projet



# Exemple

---

## **build.gradle**

```
task buildInfo(type: BuildInfo) {  
    version = project.version  
    outputFile = file("${buildDir}/generated-resources/build-info.properties")  
}
```

## **build.gradle**

```
sourceSets {  
    main {  
        output.dir(buildInfo.outputFile.parentFile, builtBy: buildInfo)  
    }  
}
```

```
dependencies {  
    runtime project(':producer')  
}
```



# Pipeline de déploiement continue

---

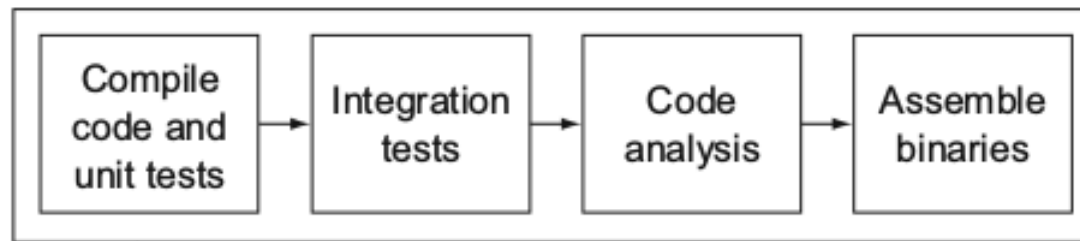
Tests  
Analyse de Code  
Intégration pipeline



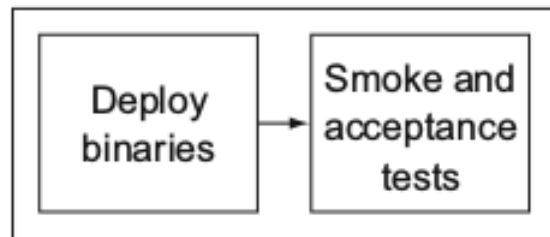
# Pipeline de déploiement continue

---

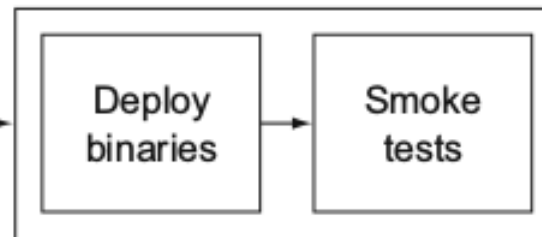
Commit stage



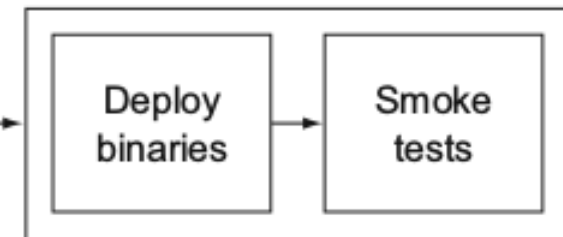
Acceptance stage



UAT



Production



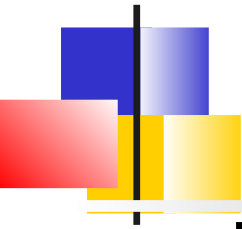


# Pipeline de déploiement continue

---

## **Tests**

Analyse de Code  
Intégration pipeline



# Rappels

---

Par défaut :

- Les classes de tests sont dans *src/test/java*
- Les ressources dans *src/test/resources*
- Les classes compilées dans *build/classes/test*

Les frameworks de test produisent les résultats des exécutions généralement

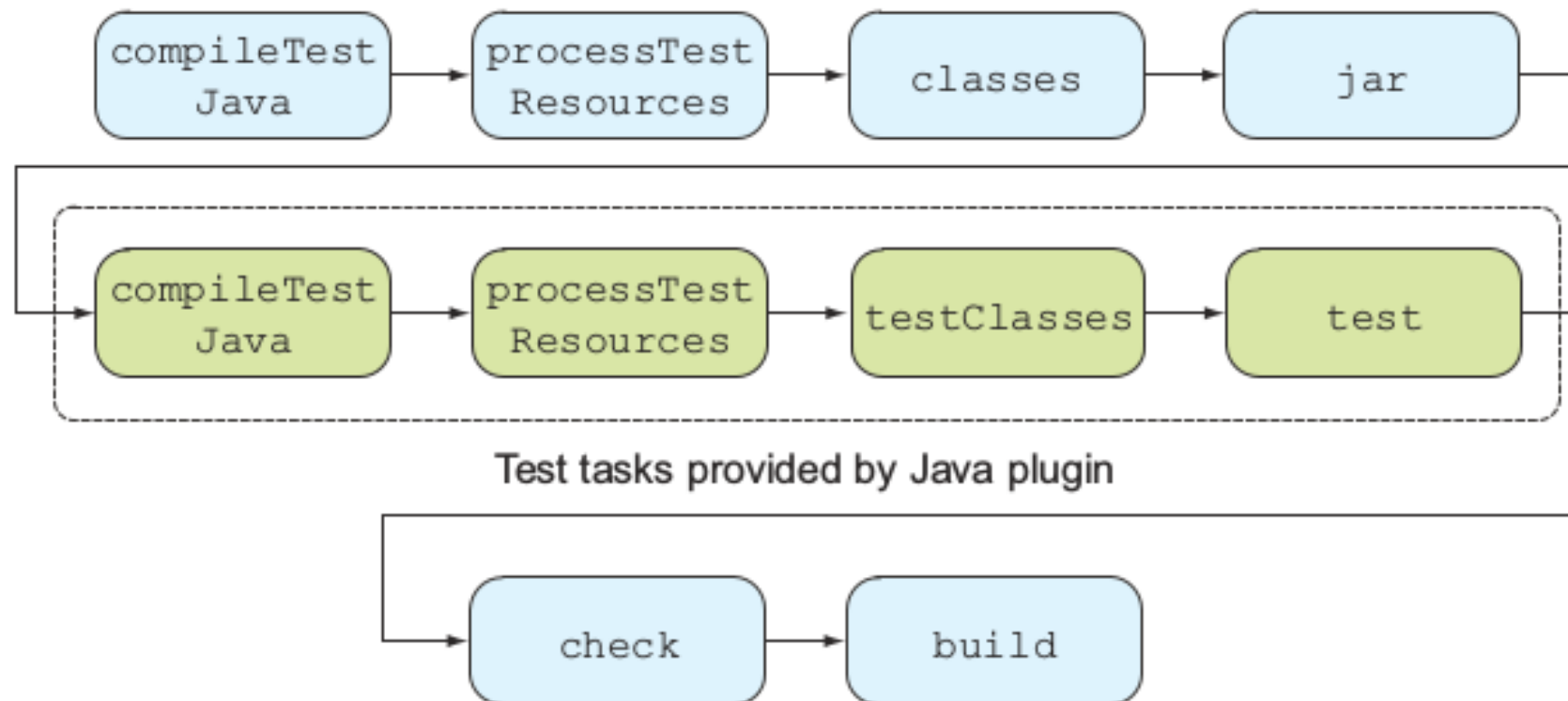
- Au format XML (*build/test-results*)
- Des rapports au format HTML (*build/reports/test*) .

2 configurations de classpath sont disponibles :  
*testImplementation* et *testRuntime*



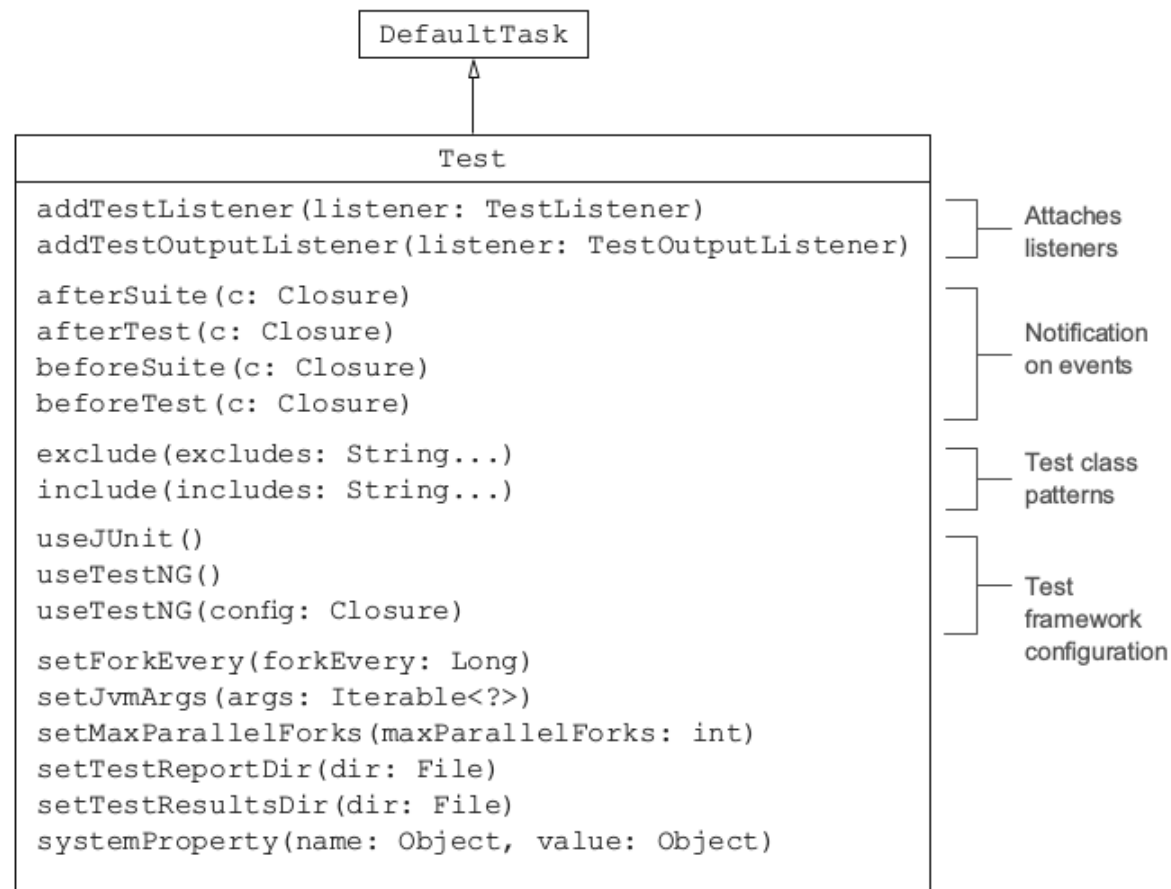
# Tâches de test

---





# Tâche Test





# Introduction

---

Gradle supporte *JUnit* et *TestNG*

Les axes de configuration sont :

- **L'exécution** : Contrôler comment les tests sont exécutés
- **La détection** : Comment Gradle détecte les tests
- **Le filtrage** : Comment sélectionner les tests à exécuter
- **Les groupes** : Permettent également de sélectionner les tests à exécutés. Les groupes apparaissent également dans le rapport de test
- **Le rapport** : Le format du rapport des tests



# Exécution des tests

---

Gradle exécute les tests dans une JVM séparée ('forked')

Des propriétés de la tâche Test peuvent contrôler l'exécution :

- **maxParallelForks** (défaut 1) : Si l'on veut du parallélisme
- **forkEvery** (défaut 0) : Re-fork une JVM au bout d'un certain nombre de tests
- **ignoreFailures** (défaut false) : Ne pas faire échouer le build si un test échoue
- **failFast** (défaut false) : Arrêt du build au 1<sup>er</sup> test échoué



# Options JVM

---

On peut également passer les options habituelles d'une JVM ou fournir des propriétés système

```
test {  
    systemProperty 'items', '20'  
    minHeapSize = '128m'  
    maxHeapSize = '256m'  
    jvmArgs '-XX:MaxPermSize=128m'  
}
```



# Callback

---

On peut réagir aux événements qui surviennent lors des tests :

- ***beforeSuite / afterSuite*** : Avant/après qu'une suite de test soit exécutée
- ***beforeTest / afterTest*** : Avant/après qu'une classe de test soit exécutée



# Logger

`<<interface>>`  
`TestLoggingContainer`



`<<interface>>`  
`TestLogging`

```
setEvents(events: Iterable<?>)
setExceptionFormat(format: Object)
setShowExceptions(flag: boolean)
setShowStackTraces(flag: boolean)
setShowStandardStreams(flag: boolean)
```



# Exemple

---

```
test {  
    testLogging {  
        // Afficher les System.out  
        showStandardStreams = true  
        // Toute la stack trace  
        exceptionFormat 'full'  
        // Événements de test  
        events 'started', 'passed', 'skipped', 'failed'  
    }  
}
```



# Détection des tests

---

Les classes de tests lors de l'utilisation de *JUnit* sont détectées comme suit :

- Toute classe étendant ***junit.framework.TestCase*** ou ***groovy.util.GroovyTestCase***
- Toute classe annotée par ***@RunWith.***
- Toute classe contenant une méthode annotée par ***@Test.*** (JUnit ou Test NG)





# Sélection des tests à exécuter

---

Gradle permet de filtrer les tests à exécuter à partir d'un nom de classe ou de méthode, les wildcards sont supportés. Le filtre est présent dans *build.gradle* ou fourni lors de l'exécution via l'option **--tests**

```
test {  
    filter {  
        //include specific method in any of the tests  
        includeTestsMatching "*UiCheck"  
        //include all tests from package  
        includeTestsMatching "org.gradle.internal.*"  
        //include all integration tests  
        includeTestsMatching "*IntegTest"  
    }  
}
```



# Les groupes

---

Depuis *JUnit 4.8*, il est possible de catégoriser les tests. Avec *JUnit5* on les tagge.

Gradle permet alors de sélectionner les tests à exécuter via leurs catégories ou tags.

```
test {  
    useJUnit {  
        includeCategories 'org.gradle.junit.CategoryA'  
        excludeCategories 'org.gradle.junit.CategoryB'  
    }  
    useJUnitPlatform {  
        includeTags 'fast'  
        excludeTags 'slow'  
    }  
}
```



# Tests d'intégration

---

Les tests d'intégration sont généralement plus long à s'exécuter et nécessitent des systèmes externes (BD, serveur, ...)

Au niveau du build, cela implique :

- Fournir des tâches séparées pour exécuter les tests unitaires et les tests d'intégration.
- Séparer les résultats et les rapports des 2 types de test.
- Déclencher les tests d'intégration lors de tâche de cycle de vie *check*



# Techniques

---

2 techniques peuvent facilement s'implémenter avec *Gradle* :

- Se baser sur des règles de nommage des classes de test pour différencier tests unitaires et tests d'intégration
- Définir un nouveau *sourceSet* pour les tests d'intégration. (mieux)



# Convention de nommage

---

```
project(':repository') {
    Repositories { mavenCentral() }
    dependencies {
        compile project(':model')
        runtime 'com.h2database:h2:1.3.170'
        testCompile 'junit:junit:4.11'
    }
    test {
        exclude '**/*IntegTest.class'
        reports.html.destination = file ("$reports.html.destination/unit")
        reports.junitXml.destination = file("$reports.junitXml.destination/unit")
    }
    task integrationTest(type: Test) {
        include '**/*IntegTest.class'
        reports.html.destination = file("$reports.html.destination/integration")
        reports.junitXml.destination =
            file("$reports.junitXml.destination/integration")
    }
    check.dependsOn integrationTest
}
```



# *sourceSet*

---

*Gradle* ajoute automatiquement les tâches requises pour compiler, tester ce nouveau *sourceSet*. Les tâches sont dérivées du nom du *sourceSet*

```
sourceSets {
    integrationTest {
        java.srcDir file('src/integTest/java')
        resources.srcDir file('src/integTest/resources')
        compileClasspath = sourceSets.main.output
                           + configurations.testRuntime
        runtimeClasspath = output + compileClasspath
    }
}
task integrationTest(type: Test, group : 'Verification') {
    testClassesDirs = sourceSets.integrationTest.output.classesDir
    classpath = sourceSets.integrationTest.runtimeClasspath
    testResultDir = file("${testResultDir}/integration")
}
check.dependsOn integrationTest
```



# Exemple démarrage serveur H2

---

```
class H2DatabaseStarter extends DefaultTask {  
    @Input  
    Integer tcpPort  
  
    @Input  
    Integer blockMs  
  
    @TaskAction  
    void start() {  
        new Thread(new H2Server(tcpPort)).start()  
        Thread.sleep(blockMs)  
    }  
  
    private class H2Server implements Runnable {  
        final Integer tcpPort  
  
        H2Server(Integer tcpPort) {  
            this.tcpPort = tcpPort  
        }  
  
        @Override  
        void run() {  
            org.h2.tools.Server.main('-tcp', '-tcpPort', tcpPort.toString())  
        }  
    }  
}
```



# Script pour BD

---

```
buildscript {  
    repositories { mavenCentral() }  
    dependencies { classpath 'com.h2database:h2:1.3.170' }  
}  
ext.h2TcpPort = 9092  
  
task startDatabase(type: H2DatabaseStarter) {  
    tcpPort = h2TcpPort  
}  
  
task stopDatabase(type: JavaExec) {  
    classpath = buildscript.configurations.classpath  
    main = 'org.h2.tools.Server'  
    args = ['-tcpShutdown', "tcp://localhost:${h2TcpPort}"]  
}  
  
task buildSchema(type: JavaExec, dependsOn: startDatabase) {  
    classpath = buildscript.configurations.classpath  
    workingDir = projectDir  
    main = 'org.h2.tools.RunScript'  
    args = ['-url', 'jdbc:h2:~/todo', '-user', 'sa', '-script', 'create-schema.sql']  
}  
// Dépendances  
integrationTest.dependsOn startDatabase  
integrationTest.finalizedBy stopDatabase
```





# Pipeline de déploiement continue

---

Tests  
**Analyse de Code**  
Intégration pipeline



# Analyse de code

---

L'analyse de code doit faire partie de la pipeline de déploiement continu

Les analyses peuvent également être longues à s'exécuter. C'est pourquoi, il faut disposer de tâches *Gradle* séparées au niveau du build.

Ces tâches sont généralement fournies par des plugins



# Couverture des tests avec *Jacoco*

---

Le framework **JaCoCo** est le framework le plus habituellement répandu.

Un plugin Gradle est directement applicable par  
`apply plugin: "jacoco"`

Une nouvelle tâche ***jacocoTestReport*** dépendante de *test* est disponible.

Un rapport est généré dans  
*\$buildDir/reports/jacoco/test*. Par défaut, un rapport HTML est également généré.



## Exemple JacoCo : prise en compte des tests d'intégration

---

```
apply plugin : 'jacoco'
```

```
task jacocoIntegrationTestReport(type : JacocoReport) {  
    sourceSets sourceSets.integrationTest  
    executionData integrationTest  
}
```



# Intégration Sonar

---

Le plugin **org.sonarqube** permet de démarrer une analyse Sonar via *Gradle*

Les propriétés de l'instance Sonar sont définies dans *gradle.properties*

```
systemProp.sonar.host.url=http://localhost:9000  
systemProp.sonar.login=<token>
```

Le plugin *org.sonarqube* est appliqué

```
plugins {  
    id "org.sonarqube" version "2.6.2"  
}
```

La tâche *sonarqube* est alors disponible, de nombreuses propriétés Sonar peuvent être affinées

```
gradle sonarqube
```



# Pipeline de déploiement continue

---

Tests  
Analyse de Code  
**Publication**  
Intégration pipeline



# Exemple

## Ajout des infos de build dans l'archive

---

```
apply plugin : 'war'
task createBuildInfoFile {
    doLast {
        def buildInfoFile = new File("$buildDir/build-info.properties")
        Properties props = new Properties() ;
        props.setProperty('version', project.version)
        props.setProperty('timestamp', project.buildTimestamp)
        props.store(buildInfoFile.newWriter(), null)
    }
}
war {
    dependsOn createBuildInfoFile
    baseName 'myWar'
    // Ajout du fichier dans l'archive
    from(buildDir) {
        include 'build-info.properties'
        into 'WEB-INF/classes'
    }
}
```



# Maven

---

Avec Java, il est courant de déployer vers des dépôts Maven.

Le plugin ***maven-publishing*** permet de

- Déclarer des dépôts maven privé (snapshot et release)
- Publier des artefacts vers ce dépôt ou vers le dépôt local

Les 2 principales tâches sont :

- ***publish***
- ***publishToMavenLocal***





# Configuration

---

La configuration s'effectue via le DSL  
***publishing*** qui contient 2 sous-blocs :

- ***publications*** : Permettant de définir ce qui va être publié
- ***repositories*** : Définit les accès aux repositories

```
publishing {  
    publications {  
    }  
    repositories {  
    }  
}
```



# DSL publications

---

Le plugin apporte le DSL *publications* permettant de configurer 4 aspects :

- Un composant via la méthode ***from***
- Des artefacts personnalisés via la méthode ***artifact***
- Les méta-données standards : artifactId, groupId et version.
- Des contenus additionnels au POM via la méthode ***pom***



# DSL repositories

---

Le DSL ***repositories*** définit principalement :

- ***name*** : Un nom
- ***url*** : L'URL de base du dépôt
- ***credentials*** : login/password



# Exemple basique

---

```
publishing {
  publications {
    maven(MavenPublication) {
      artifact("build/libs/finance-$version"+"*.jar") {
        extension 'jar'
      }
    }
  }

  repositories {
    maven {
      name 'nexus'
      url 'http://nexus/repository/maven-releases/'
      credentials {
        username project.repoUser
        password project.repoPassword
      }
    }
  }
}
```



# Docker

---

Pas de plugin coeur Gradle mais le plugin

***com.bmuschko.docker-remote-api***

qui apporte de nouvelles types de tâches :

- Création de DockerFile
- Build image
- Pull/Push



# Pipeline de déploiement continue

---

Tests  
Analyse de Code  
Publication  
**Intégration pipeline**



# Intégration Jenkins

---

L'intégration Jenkins est facilité via le *wrapper* Gradle qui assure que la même version de *gradle* est installée et utilisé sur l'esclave exécutant le Job

Il est ainsi aisé :

- De faire un freestyle Job avec ou sans le plugin Gradle installé
- Ou utiliser les pipeline Jenkins (également Groovy)



# FreeStyle + Plugin Gradle

**Build**

**Invoke Gradle script**

☐ Invoke Gradle

☒ Use Gradle Wrapper

Make gradlew executable

☐

Wrapper location

Tasks

▼

Switches

▼

X

?

?

?

?

?

?





# Build Scan

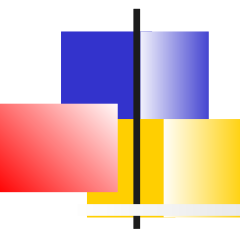
---

L'option `--build-scan` de la commande Gradle permet de générer un rapport sur le site *gradle.com* qui peut aider lors du debug d'un build.

Le plugin Gradle de Jenkins permet de générer un lien vers ce rapport.

En version Pipeline, cela donne :

```
node {  
    withGradle {  
        sh './gradlew build'  
    }  
}
```



# Sortie console

Step	Arguments	Status
Start of Pipeline - (2,3 sec in block)		
node - (2,2 sec in block)		
node block - (2 sec in block)		
git - (0,56 sec in self)		
withGradle - (1,3 sec in block)		
withGradle block - (1,2 sec in block)		
sh - (1,1 sec in self)	./gradlew build --scan	



# Merci !!

---

Pour votre attention !



# Annexes

---

## Stratégies de résolution des dépendances



# Introduction

---

Gradle permet d'influencer directement le comportement du moteur de résolution des dépendances.

Méthode à utiliser seulement si les mécanismes plus haut-niveau ne suffise pas



# API

---

Les règles de résolution de dépendance fournissent un moyen puissant de contrôler la résolution de dépendances.

- A chaque résolution d'une dépendance, l'ensemble des règles configurées sont appliquées.
- Une règle peut changer le groupe, le nom, la version de la résolution demandée.

Voir la classe *ResolutionStrategy*<sup>1</sup>

1.<https://docs.gradle.org/current/dsl/org.gradle.api.artifacts.ResolutionStrategy.html>



# Exemple

---

Usage : les développeurs n'indiquent pas de versions dans leur dépendances mais utilisent un mot clé. Ex : *default*.

Une règle modifie le mot-clé *default* par un numéro de version géré au niveau corporate



# Exemple

---

```
configurations.all {  
    resolutionStrategy.eachDependency { DependencyResolveDetails details ->  
        if (details.requested.version == 'default') {  
            def version =  
findDefaultVersionInCatalog(details.requested.group,  
details.requested.name)  
            details.useVersion version.version  
            details.because version.because  
        }  
    }  
}  
  
def findDefaultVersionInCatalog(String group, String name) {  
    // Une logique qui résoud la version par défaut  
    [version: "1.0", because: 'tested by QA']  
}
```