# Compilation / exécution d'un projet Java

Michel Billaud

8 mai 2020

## Table des matières



## 1 Objectif de ce document.

L'utilisation d'un IDE (*Integrated Development Environment*) comme *Eclipse*, *NetBeans*, *IntelliJ*, etc. est un grand confort pour le développement d'applications.

Le programmeur, si il reste conscient qu'il y a des phases d'édition, de compilation, d'exécution des programmes et des tests, n'est pas au courant (et préfère ne pas l'être) des détails techniques de ces phases, qui lui restent cachés.

Pour mieux comprendre ce qui se passe, nous avons

- mis en place un petit projet Java, une "application" avec 2 packages et trois classes, et des tests unitaires ;
- étudié les commandes lancées par Netbeans pour compiler et exécuter ;
- essayé de les reproduire avec un `Makefile`

Une méthode simple pour voir les commandes

- Dans un shell, se placer dans le répertoire du projet ;
- faire `ant -v build` pour compiler, `ant -v run` pour faire tourner ;
- reconstituer les commandes qui ont été lancées à partir des affichages.

En pratique c'est un peu laborieux, mais on y arrive.

Dans un premier temps, on présente la **compilation des sources** de l'application. Ensuite on verra le **traitement des tests unitaires**.

## 2 Le projet

Quand le projet est compilé, il est dans un répertoire qui a la structure suivante :

```
Projet
|-- build                    # construit par la compilation
|   |-- classes
|   |   |-- appli
|   |   |   |-- Aux.class
|   |   |   `-- Prog.class
|   |   `-- pkg
|   |       `-- Counter.class
|   `-- test
|       `-- classes
|           `-- my
|               `-- pkg
|                   |-- OtherTest.class
|                   `-- TestCounter.class
|-- src                              # sources des classes
|   |-- appli
|   |   |-- Aux.java
|   |   `-- Prog.java
|   `-- pkg
|       `-- Counter.java
`-- test                             # sources tests unitaires
    `-- my
        `-- pkg
            |-- OtherTest.java
            `-- TestCounter.java
```

- Les sources écrits par le programmeur sont dans `src` et `test`.
- le contenu de `build` est construit par la compilation (et effacé par un “clean”).

## 3 Les fichiers source

```java
// appli/Prog.java

package appli;
import pkg.Counter;

public class Prog {

    public static void main(final String[] args) {
        Aux.hello("guys");
        final Counter c = new Counter();
        c.inc();
        System.out.println("compteur = " + c.get());
    }
}
```

```java
// srs/appli/Aux.java

package appli;

public class Aux {
    public static void hello(String name) {
        System.out.println("Hello " + name);
    }
}
```

```java
// src/pkg/Counter.java

package pkg;

public class Counter {
    int n = 0;

    public int get() {
        return n;
    }

    public Counter inc() {
        n += 1;
        return this;
    }
}
```

## 4 La compilation de l'application

La compilation peut se faire "à la main" par :

```
javac -cp build/classes -d build/classes \
    src/pkg/Counter.java \
    src/appli/Prog.java  \
    src/appli/Aux.java   \
```

L'option `-d` indique où placer les classes, avec une hiérarchie similaire à celle des packages.

La recherche des fichiers sources peut être automatisée grâce à la commande `find` :

```
javac -cp build/classes -d build/classes \
    $(find src -name "*.java")
```

## 5 Exécution de l'application

Il faut indiquer l'emplacement des classes, et le nom de celle dont on veut exécuter la méthode `Main`

```
java -cp build/classes Appli.prog
```

## 6 Les tests unitaires

Les tests unitaires sont dans une branche à part, avec une structure de package parallèle à celle des classes de l'application

```java
//  test/my/pkg/TestCounter.java

package my.pkg;

import org.junit.Test;
import static org.junit.Assert.*;

import pkg.Counter;

public class TestCounter {
    @Test
    public void testGet() {
        System.out.println("get");
        final Counter c = new Counter();
        assertEquals(0, c.get());
        c.inc();
        assertEquals(1, c.get());
```

```
        c.inc().inc().inc();
        assertEquals(4, c.get());
    }
}
```

La classe `OtherTest` est semblable. Il y a 2 classes de tests pour aborder ensuite la problématique “faire exécuter tous les tests”.

# 7 Compilation des tests unitaires

Attention : les tests unitaires utilisent les classes de l'application, qui doivent avoir été compilées avant, et se trouvent dans `build/classes`.

Pour la compilation, on cite les bibliothèques Junit 4.12 et Hamcrest 1.3 dans le *classpath* :

```
EXT_DIR=/usr/local/netbeans-11.3/netbeans/platform/modules/ext
JUNIT_JAR=${EXT_DIR}/junit-4.12.jar
HAMCREST_JAR=${EXT_DIR}/hamcrest-core-1.3.jar

javac -cp build/classes :build/test/classes :${JUNIT_JAR} :${HAMCREST_JAR} \
      -d build/test/classes \
      $(find test -name "*.java")
```

# 8 Exécution d'un test unitaire

Dans le framework, les tests sont exécutés par la classe `JUnitTestRunner`, qui charge et exécute un des tests (on lui donne le nom de la classe contenant les tests à exécuter).

- Le `JUnitTestRunner` fait partie de la bibliothèque `JUnit`, et utilise des bibliothèques d'`ant`.
- Aux bibliothèques déjà mentionnées pour la compilation, on ajoute dans le *classpath* :

```
/usr/share/ant/ant-launcher-1.10.5.jar
/usr/share/ant/lib/ant.jar
/usr/share/ant/lib/ant-junit.jar
/usr/share/ant/lib/ant-junit4.jar
```

- ce n'est pas tout, il faut encore une certaine quantité d'options :

```
skipNonTests=false
filtertrace=true
haltOnError=false
haltOnFailure=false
showoutput=true
outputtoformatters=true
logfailedtests=true
threadid=0
logtestlistenerevents=false
formatter=org.apache.tools.ant.taskdefs.optional.junit.BriefJUnitResultFormatter
```

dont il faudrait chercher la signication précise dans la documentation.

Pour exécuter un test, on lance donc une longue commande

```
java -cp /usr/share/ant/ant-launcher-1.10.5.jar :\
/usr/share/ant/lib/ant.jar :\
/usr/share/ant/lib/ant-junit.jar :/usr/share/ant/lib/ant-junit4.jar :\
build/classes :build/test/classes :\
/usr/local/netbeans-11.3/netbeans/platform/modules/ext/junit-4.12.jar :\
/usr/local/netbeans-11.3/netbeans/platform/modules/ext/hamcrest-core-1.3.jar \
-ea org.apache.tools.ant.taskdefs.optional.junit.JUnitTestRunner \
my.pkg.TestCounter \
'skipNonTests=false' 'filtertrace=true' 'haltOnError=false' \
```

```
'haltOnFailure=false' 'showoutput=true' 'outputtoformatters=true' \
'logfailedtests=true' 'threadid=0' 'logtestlistenerevents=false' \
'formatter=org.apache.tools.ant.taskdefs.optional.junit.BriefJUnitResultFormatter'
```

Le résultat qui s'affiche :

```
Testsuite : my.pkg.TestCounter
get
Tests run : 1, Failures : 0, Errors : 0, Skipped : 0, Time elapsed : 0,01 sec

------------- Standard Output ---------------
get
------------- ---------------- ---------------
test my.pkg.OtherTest
Testsuite : my.pkg.OtherTest
get
Tests run : 1, Failures : 0, Errors : 0, Skipped : 0, Time elapsed : 0,012 sec
```

# 9 Exécuter tous les tests

Traditionnellement, les tests sont des classes dont le nom commence ou finit par `Test`.

On peut les rechercher dans le répertoire `test` par un `find` :

```
$ find test \( -name "*Test.java" -o -name "Test*.java" \)
test/my/pkg/TestCounter.java
test/my/pkg/OtherTest.java
```

Avec `sed`, on élimine "`test/`" au début et "`.java`" à la fin, et on remplace les barres par des points :

```
$ find test \( -name "*Test.java" -o -name "Test*.java" \)  | \
> sed -e "s,test/,," -e "s/.java$$//" -e "s,/,.,g"
my.pkg.TestCounter.java
my.pkg.OtherTest.java
```

et il ne reste plus qu'à lancer une boucle pour faire exécuter tous les tests.

# 10 Makefile

Un `Makefile` permet d'automatiser tout cela

```
#
# Makefile pour compilation/execution de projets Java
# + tests unitaires
#

main_class = appli.Prog

test_classes = $(shell \
    find test \( -name "*Test.java" -o -name "Test*.java" \)  | \
    sed -e "s,test/,," -e "s/.java$$//" -e "s,/,.,g" )


SOURCES = $$(find src  -name "*.java")
TESTS =   $$(find test -name "*.java")

EXT = /usr/local/netbeans-11.3/netbeans/platform/modules/ext
JUNIT_JARS = $(EXT)/junit-4.12.jar :$(EXT)/hamcrest-core-1.3.jar

ANT_DIR = /usr/share/ant
ANT_JARS := $(ANT_DIR)/ant-launcher-1.10.5.jar
```

```
ANT_JARS := $(ANT_JARS) :$(ANT_DIR)/lib/ant.jar
ANT_JARS := $(ANT_JARS) :$(ANT_DIR)/lib/ant-junit.jar
ANT_JARS := $(ANT_JARS) :$(ANT_DIR)/lib/ant-junit4.jar

RUNNER = org.apache.tools.ant.taskdefs.optional.junit.JUnitTestRunner

TEST_OPTIONS = 'skipNonTests=false'
TEST_OPTIONS +=  'filtertrace=true'
TEST_OPTIONS +=  'haltOnError=false'
TEST_OPTIONS +=  'haltOnFailure=false'
TEST_OPTIONS +=  'showoutput=true'
TEST_OPTIONS +=  'outputtoformatters=true'
TEST_OPTIONS +=  'logfailedtests=true'
TEST_OPTIONS +=  'threadid=0'
TEST_OPTIONS +=  'logtestlistenerevents=false'
TEST_OPTIONS +=  'formatter=org.apache.tools.ant.taskdefs.optional.junit.BriefJUnitResultFormatter'


.PHONY : build compile-src compile-tests run run-tests clean

compile-src : build/classes
    javac -cp build/classes -d build/classes $(SOURCES)

compile-tests : compile-src build/test/classes
    javac -cp build/classes :build/test/classes :$(JUNIT_JARS) -d build/test/classes $(TESTS)


run : compile-src build/classes
    java -cp build/classes $(main_calls)

run-tests : compile-tests
    for t in ${test_classes}; do \
        java -cp $(ANT_JARS) :build/classes :build/test/classes :$(JUNIT_JARS) -ea $(RUNNER) $$t $(TEST_OP
    done

build/classes build/test/classes :
    mkdir -p $@

clean :
    $(RM) $$(find . -name "*~")
    $(RM) -r build/*
```

# 11 Complément : fabrication du jar

Le format JAR est destiné aux archives contenant les classes, et un manifeste indiquant en particulier la classe dont il faut, par défaut, lancer le `Main`.

Il suffit d'ajouter quelques lignes au `Makefile` :

```
project_name = demo-1.0

# ...

dist : compile-src
    $(RM) -rf dist
    mkdir dist
    jar cef $(main_class) dist/$(project_name).jar -C build/classes .
```