Paquetages et modules

Université Française d'Égypte
Version O 2.6 – 16/10/20
Richard Grin

1

Plan de cette partie

- Paquetages
- Quelques principes de programmation
- Modules

2

PAQUETAGE - PACKAGE

3

Définition d'un paquetage

- Ensemble de classes qui ont un centre d'intérêt commun
- Par exemple toutes les classes qui servent à la gestion des étudiants

4

Exemples de paquetages du JDK

- `java.lang` : classes de base de Java (par exemple la classe `String`)
- `java.util` : utilitaires, collections

5

Nommer une classe

- Le nom complet d'une classe (*qualified name*) est le nom de la classe préfixé par le nom de son paquetage : `java.util.ArrayList`
- Une classe du même paquetage peut être désignée par son nom « terminal » (`ArrayList`)
- Une classe d'un autre paquetage doit être désignée par son nom complet

6

Importer une classe d'un paquetage

- Permet de désigner une classe d'un autre paquetage par son nom terminal :

```
import java.util.ArrayList;

public class Classe {
  ...
  ArrayList liste = new ArrayList();
```


7

Importer une classe d'un paquetage

- Seulement une facilité d'écriture ; ce code fonctionne aussi :

```
public class Classe {
  ...
  java.util.ArrayList liste =
      new java.util.ArrayList();
```


8

Importer toutes les classes d'un paquetage

- On peut importer toutes les classes d'un paquetage :

```
import java.util.*;
```

- Erreur fréquente chez les débutants :

~~`import java.util;`~~

- Les classes du paquetage `java.lang` sont automatiquement importées

9

Ajout d'une classe dans un paquetage

- `package` *nom-paquetage*`;`
  au début du fichier source de la classe
- Par défaut, une classe appartient au « paquetage par défaut » :
  - paquetage sans nom
  - toutes les classes situées dans le même répertoire y appartiennent
- Éviter d'utiliser le paquetage par défaut

10

Sous-paquetage

- Un paquetage peut avoir des sous-paquetages
- Par exemple, `java.util.regex` est un sous-paquetage de `java.util`
- L'importation des classes d'un paquetage n'importe pas les classes des sous-paquetages ; si on veut importer les 2 paquetages :

```
import java.util.*;
import java.util.regex.*;
```


11

Nom d'un paquetage

- Conseillé de préfixer ses propres paquetages par son « adresse Internet » :
  `eg.ufe.mohamed.liste`

12

Placement d'un paquetage

- Les fichiers `.class` doivent se situer dans l'arborescence d'un des répertoires du *classpath* à un endroit qui correspond au nom du paquetage
- Les classes de `eg.ufe.mohamed.liste` doivent se trouver dans un sous-répertoire `eg/ufe/mohamed/liste` d'un des répertoires du *classpath*

13

Encapsulation d'une classe dans un paquetage

- Si la définition de la classe commence par `public class` la classe est accessible de partout
- Sinon, la classe n'est accessible que depuis les classes du même paquetage
- De même, si un membre d'une classe n'est ni `public` ni `private`, il n'est accessible que par les classes du même paquetage

14

Compiler les classes d'un paquetage

`javac -d` *rep Classe.java*

- « `-d` » indique le répertoire *racine* où javac rangera les fichiers « `.class` »
- Le fichier `.class` d'une classe du paquetage `nom1.nom2` sera rangé dans le répertoire *rep/*`nom1/nom2`
- Permet de séparer les fichiers `.class` des fichiers `.java`

15

Compiler les classes d'un paquetage

- L'option `-d` est souvent associée à l'option `-classpath` :

`javac -classpath` *rep* `-d` *rep* `Classe.java`

- Le répertoire par défaut de `-d` et `-classpath` est le répertoire courant

16

Exécuter une classe d'un paquetage

- Donner le nom complet de la classe :

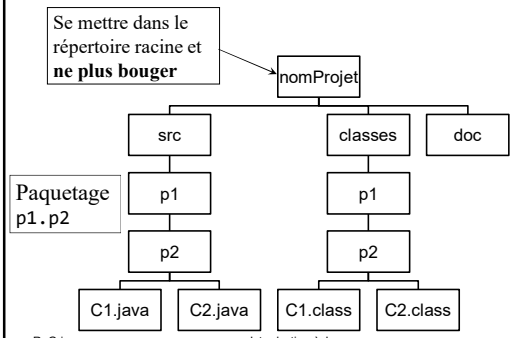
`java p1.p2.C`

- Si le fichier .class n'est pas juste sous le répertoire courant (en tenant compte du paquetage), il faut utiliser l'option `-classpath` :

`java –classpath` *rep* `p1.p2.C`

17

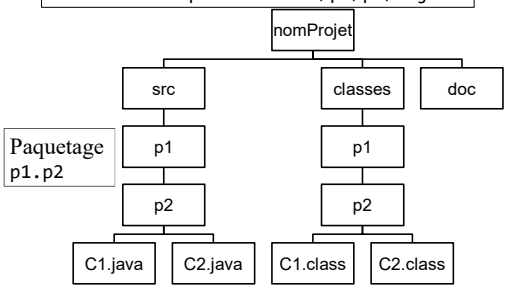
Placements des fichiers préconisés



18

Compilation classes paquetage

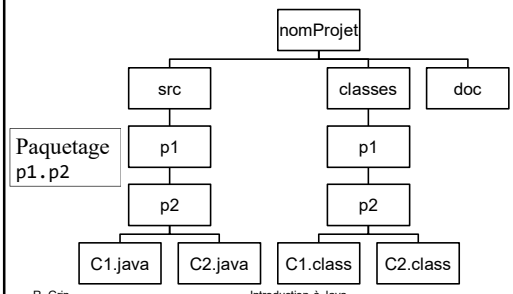
```
javac –d classes –classpath classes
      -sourcepath src src/p1/p2/*.java
```



19

Exécution

```
java –classpath classes p1.p2.C1
```



20

classpath

21

Classpath

- Contient les endroits où trouver les classes :
  - répertoires
  - fichiers `.jar`

22

Exemple de Classpath

- Sous Windows, séparateur « `;` » :
  `.;c:\java\mespackages;c:\mesutil\cl.jar`

Que signifie ce classpath ?

23

Annexe : Quelques principes de programmation

24

Ce qu'il faut rechercher

- Une plus grande facilité de programmation
- Mais surtout
  - une maintenance plus aisée
  - une extensibilité accrue

25

Comment ?

- Modularité : décomposer en éléments plus simples
- Encapsulation : cacher ce qu'il n'est pas indispensable de voir
- Lisibilité : faciliter la compréhension des programmes
- Réutilisabilité : écrire des modules réutilisables dans les futurs développements (difficile)

26

Modularité

- Un programme est modulaire s'il est découpé en modules (plus ou moins) indépendants
- Un bon découpage doit satisfaire les 2 critères :
  - forte cohésion des éléments d'un module
  - faible couplage entre deux modules différents
- Ces 2 principes favorisent l'utilisation, la réutilisation et la maintenance des modules :
  - plus de souplesse : un module - une fonctionnalité
  - les modifications dans un module ont moins d'impacts sur les autres modules

27

Encapsulation

- Ne montrer et ne permettre de modifier que ce qui est nécessaire à une bonne utilisation
  - on montre l'interface (services offerts) d'un module
  - on cache l'implémentation (comment sont rendus les services)
- Avantages :
  - simplification de l'utilisation (la complexité ne dépend que de l'interface publique)
  - meilleure robustesse du programme
  - simplification de la maintenance de l'application (un changement d'implémentation n'a pas d'impact sur le code extérieur)

28

Attribution des fonctionnalités

- Comment choisir l'objet qui doit être le responsable de l'exécution d'une action ?
- Déterminer les informations nécessaires à l'exécution
- L'objet qui possède le plus d'information est le mieux placé pour exécuter l'action
- Localisation => modularité et encapsulation facilitées

29

- Suite TP 2 sur les paquetages et les modules

30

MODULES

31

Quelques problèmes

- Si une classe est `public` dans un paquetage pour qu'elle puisse être utilisée par un autre paquetage, tous les autres paquetages ont accès à cette classe
- Sans parcourir tout le code, on ne sait pas quels sont les autres paquetages utilisés par un paquetage

32

Les modules comme solution

- Les modules permettent de résoudre ces problèmes
- Par exemple, on peut savoir quels sont les autres modules qu'un module utilise
- Ils sont surtout utiles pour les grosses applications ; ils ne seront pas étudiés en détails dans ce cours d'introduction

33

Définition

- Un module est une collection de paquetages, qui a un nom
- Un module contient un fichier descripteur de module `module-info.class` placé directement sous le répertoire du module
- Ce fichier indique explicitement de quels modules le module dépend, quels paquetages sont exportés par le module, et divers autres informations

34

Nom d'un module

- Le nom doit être unique dans toute l'application ; comme pour les paquetage, on préconise donc d'utiliser les noms avec un préfixe qui identifie le développeur ou l'organisme qui l'emploie

35

Exemple de module-info.java

```
module eg.ufe.codecorrecteur {
  exports eg.ufe.codecorrecteur;
  exports eg.ufe.testcodecorrecteur;
  requires eg.ufe.mescodes;
}
```

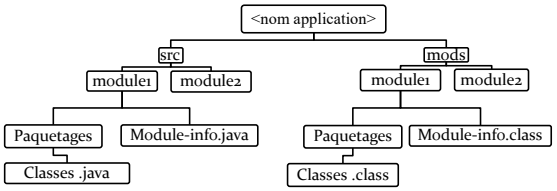

36

Organisation des fichiers préconisée

- Les fichiers source de l'application peuvent être placés de différentes façons
- Les transparents suivants donnent un exemple d'organisation efficace quand on travaille avec un ou plusieurs modules
- Vous suivrez cette organisation dans le cours

37

Placements préconisés pour une application avec plusieurs modules



38

Compilation

- Se placer dans le répertoire du projet et ne plus en bouger pour taper les commandes
- javac avec les options
  - `--module-path` pour désigner l'endroit où sont les modules
  - `-d` pour désigner l'endroit où le code compilé est rangé
- `javac -d mods --module-path mods --module-source-path src <tous les fichiers .java à compiler>`

39

Exemple

```
$ javac --module-path mods -d mods --module-source-path src
./src/eg.ufe.codecorrecteur/eg/ufe/codecorrecteur/*.java
./src/eg.ufe.codecorrecteur/eg/ufe/testcodecorrecteur/*.java
./src/eg.ufe.codecorrecteur/module-info.java
./src/testcodecorrecteur/ufe/test/*.java
./src/testcodecorrecteur/module-info.java
```

ou (les dépendances sont trouvées si on utilise l'option `--module-source-path`) :

```
$ javac --module-path mods -d mods --module-source-path src
./src/eg.ufe.codecorrecteur/eg/ufe/testcodecorrecteur/*.java
```


40

Exécution

- Indiquer le chemin des modules à la commande java et aussi la classe principale qui démarrera l'exécution
- On désigne la classe principale qui démarrera l'exécution avec l'option `--module` (ou `-m`) en donnant le nom du module suivi de « / » et du nom complet de la classe

41

Exemple

```
$ java --module-path mods
-m testcodecorrecteur/eg.ufe.test.TestCode
```


42

Quelques modules

- `java.base` contient ces paquetages, avec leurs sous-paquetages : `java.lang`, `java.util`, `java.math`, `java.time`, `java.io`, `java.nio`, `java.security`
- `java.sql`
- `java.xml`