Programmation Java

Cours 6

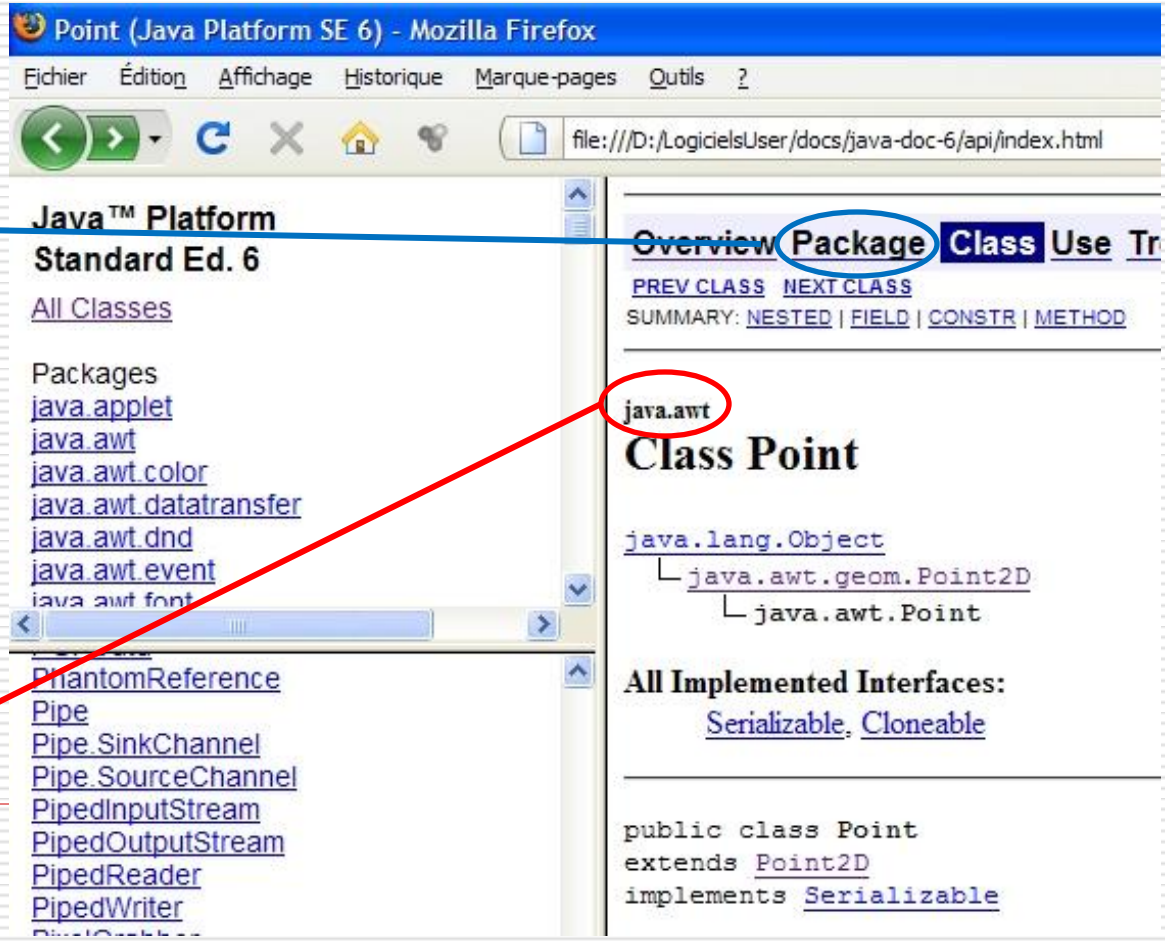
Packages

Généralités

- 1 package permet de :
  - Regrouper des types (classes, interfaces, exceptions, énumérations, ...) au sein d'un même module
  - Regroupement thématique
  - Découpage d'une grande application (milliers de classes) en modules
  - Signer ses classes
    - Ex : la classe Point d'un étudiant est différente de celle d'un autre et de celle de Java (java.awt.Point)

Packages de Java

- Les classes de l'API Java sont regroupées en de nombreux packages :



Packages de Java (2)

java
| io ----------Entrées-sorties
| lang ----------Classes de base de Java
| | ArithmeticException
| | ...
| | Boolean
| | Class ---------- La classe *Classe*
| | Object ---------- La classe mère *Object*
| | reflect ----------Le paquetage des classes pour la réflexion
| | | ...
| | | Method
| | | ...
| | String ---------- La classe *String* des chaînes de caractères
| | ...
| math ----------Nombres longs et méthodes d'arrondis
| net ----------Le paquetage des classes réseau
| sql ----------Interfaçage avec une BDD
| util ----------Le paquetage des classes utilitaires
| ...

Créer un paquetage

- Un paquetage est constitué d'un certain nombre de types rangés dans un même répertoire.
- Les types sont explicitement attribués au paquetage par le mot-clef **`package`**.
- En Java toute classe fait partie d'un **package**
- Un type non attribué se situe dans le paquetage anonyme local (implicitement ".").

Nommage des paquetages

- Les noms des packages ont une structure hiérarchique
- Les paquetages constituent un espace de nom universel (en théorie)
- Il est conseillé de préfixer ses propres paquetages par son adresse Internet
- On commence un paquetage par le nom de domaine de l'entité qui développe à l'envers. Ex. :

```
package fr.eisti.packagePath;
```

- On continue par un chemin qui désigne un module ou sous-module. Ex :

```
package fr.eisti.arel.authentification;
```

Utiliser un paquetage

- Une classe empaquetée a un nom complètement qualifié :

```
fr.eisti.arel.authentification.User;
```

- Pour l'utiliser à l'extérieur du paquetage :
  - par le nom qualifié de la classe
  - par le nom simple de la classe si
    - elle est importée.
    - son paquetage est importé.
- Pour l'utiliser au sein du paquetage
  - par le nom simple de la classe.

Utiliser un paquetage (2)

- Importer un membre unique :

```
import fr.eisti.arel.authentification.User;
```

- Importer un paquetage complet :

```
import fr.eisti.arel.authentification.*;
```

- Import automatique
  - Le paquetage java.lang est importé automatiquement. Il est **inutile** de mentionner

```
import java.lang.*;
```

Importer des constantes **static**

- Depuis le JDK 5.0 on peut importer des variables ou méthodes statiques d'une classe ou d'une interface
- On allège ainsi le code, par exemple pour l'utilisation des fonctions mathématiques de la classe **java.lang.Math**
- A utiliser avec précaution pour ne pas nuire à la lisibilité du code (il peut être plus difficile de savoir d'où vient une constante ou méthode)

Exemple d'import static

- ```
  import static java.lang.Math.*;
  x = max(sqrt(abs(y)), sin(y));
     // au lieu de Math.sqrt, Math.sin…
  ```
- On peut importer une seule variable ou méthode :

  ```
  import static java.lang.Math.PI;
  x = 2* PI;
  ```
- ```
  import static java.lang.System.*;
   out.println("Bonjour à tout le monde");
     // i.e. System.out
   exit(0); // i.e. System.exit
  ```

Portée dans un paquetage

- Une classe, un attribut ou une méthode, dont la définition n'est pas précédée de *public*, *private* ou *protected* a une portée limitée au paquetage dans lequel il ou elle est défini (mode *implicite* ou *package private*).
- Equivalent UML : ~

Sous-paquetage

- Un paquetage peut avoir des sous-paquetages
- Par exemple, **java.awt.event** est un sous-paquetage de **java.awt**
- L'importation des classes d'un paquetage n'importe pas automatiquement les classes des sous-paquetages ;

on devra écrire par exemple :

import java.awt.*;

import java.awt.event.*;

- Les paquetages **ne sont pas** une hiérarchie au sens de l'inclusion.

Exemple de sous paquetage

- Les paquetages sont rangés dans une structure de répertoire.
  - ```
    package monPaquetage;
    class X{ ... }
    ```
    La classe *X* appartient au paquetage *monPaquetage*.
    Le fichier *X.java* est rangé dans le répertoire *monPaquetage*.
  - ```
    package monPaquetage.monSousPaquetage;
    class Y{ ... }
    ```
    La classe *Y* appartient au pkg. *monPaquetage.monSousPaquetage*.
    Le fichier *Y.java* est rangé dans *monPaquetage/monSousPaquetage.*
- Pour utiliser X et Y en dehors de leur paquetage, on peut :
  - les désigner complètement :
    `monPaquetage.X`, ou `monPaquetage.monSousPaquetage.Y`
  - importer les paquetages, puis les désigner par leur nom simple :
    ```
    import monPaquetage.*;
    import monPaquetage.monSousPaquetage.*;
    ```
    `X ou Y`

Problèmes divers

- Conflits de noms.

  Si on importe deux paquetages, le risque que deux classes y portent le même nom existe.

  => Il faut référer aux classes par leur nom qualifié.

Exemple de conflits de noms

- Problème si deux paquetages différents contiennent deux classes de même nom :
- Les paquetages *java.util* et *java.sql* contiennent tous les deux une classe *Date* :

```
import java.util.*;
import java.sql.*;
      ...
Date d; // erreur de compilation
```

La solution est alors de nommer complètement la classe :

```
import java.util.*;
import java.sql.*;
java.util.Date d; // OK
```

Exemple de conflits de noms (2)

- 2ème solution : importer les classes et non les paquetages en entier :

```
import java.util.List;
import java.util.LinkedList;
import java.sql.Connection;
import java.sql.Statement;
import java.sql.Date;
Date d; // OK
```

Gérer les fichiers physiques

- Le compilateur et la machine virtuelle cherche les fichiers physiques dans les répertoires du CLASSPATH.
- La hiérarchie de noms induit une hiérarchie physique :

```
import fr.eisti.arel.authentification.User;
```

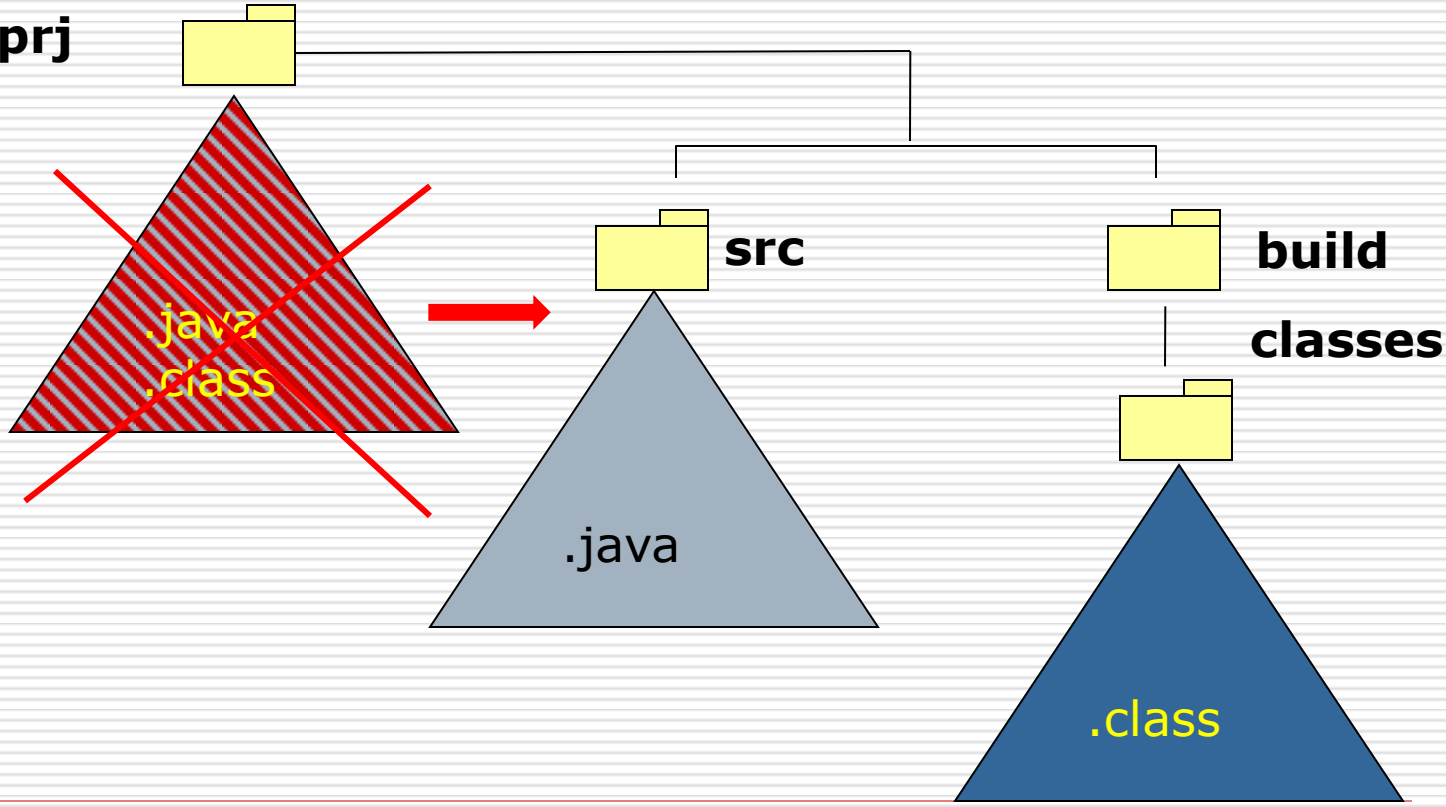
  - suppose un fichier `User.java` dans le "répertoire" :

```
${CLASSPATH}/fr/eisti/arel/authentification
```

- Le CLASSPATH peut être une liste de répertoires
- Les fichiers physiques sont recherchés :
  - selon l'ordre des répertoires du CLASSPATH
  - puis en suivant les règles d'équivalence entre nom de paquetage et hiérarchie de répertoires

Gérer les fichiers physiques (2)

- On préfère séparer le répertoire des .java de celui des .class



Exemple complet

fichier p1/C1.java :

```
package p1;
public class C1 {
    public int m;
    private int n;
    int o;
    protected int p;

    public void m1() {
        m = 1; // bon
        n = 2; // bon
        o = 3; // bon
        p = 4; // bon
    }
}
```

fichier p1/C2.java :

```
package p1;
public class C2 extends C1 {
    public void m2() {
        m = 1; // bon
        // n = 2; // interdit !
        o = 3; // bon
        p = 4; // bon
    }
}
```

fichier p2/C3.java :

```
package p2;
import p1.*;
public class C3 extends C2 {
    public void m3() {
        m = 1; // bon
        // n = 2; // interdit !
        // o = 3; // interdit !
        p = 4; // bon
    }
}
```

Exemple complet (suite)

fichier p1/D.java :

```
package p1;
public class D  {
    C1 c1;
    public void d() {
        c1 = new C1();
        c1.m = 1; // bon
        // c1.n = 2; // interdit
        c1.o = 3; // bon
        // c1.p = 4; // interdit
    }
}
```

fichier p2/E.java :

```
package p2;
import p1.*;
public class E  {
    C1 c1;
    public void d() {
        c1 = new C1();
        c1.m = 1; // bon
        // c1.n = 2; // interdit
        // c1.o = 3; // interdit
        // c1.p = 4; // interdit
    }
}
```

Exemple complet (fin)

fichier Essai.java :

```
import p1.*;
import p2.*;
public class Essai {
    public static void main(String args[]) {
        System.out.println("--");
        C1 c1 = new C1();
        C2 c2 = new C2();
        C3 c3 = new C3();
        D d = new D();
        System.out.println("--");
    }
}
```

Exécuter une classe d'un paquetage

- Par exemple, si la classe **C1** appartient au paquetage **p1** (et contient une méthode main) :

java p1.C1

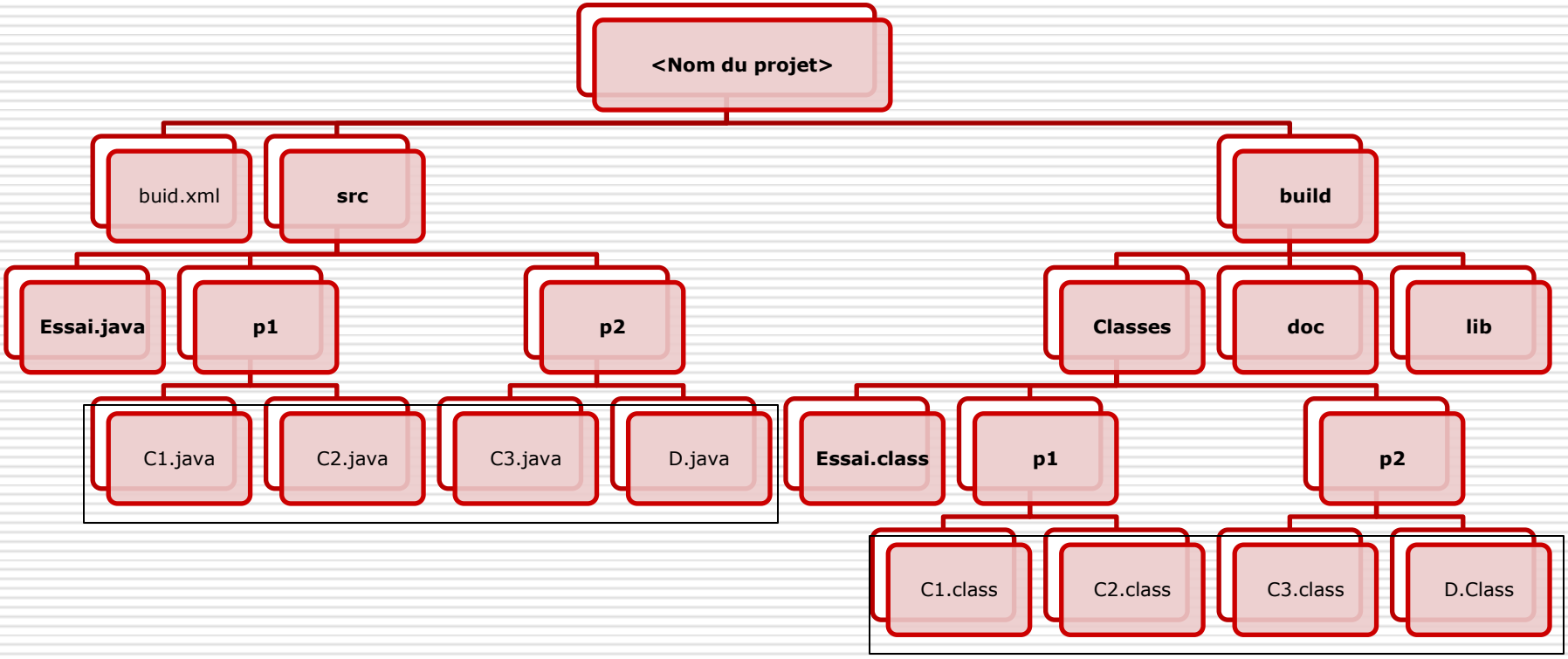
- Le fichier **C1.class** devra se situer dans un répertoire **p1** d'un des répertoires du *classpath*
  (option **-classpath** ou variable **CLASSPATH**)

Utilisation pratique des paquetages

- Les premières tentatives de développement avec paquetages conduisent à de grosses difficultés pratiques pour compiler les classes
- Ces difficultés peuvent être évitées
  - en respectant quelques principes simples pour le placement des fichiers sources et classes
  - en utilisant correctement les options **–classpath** et **–d**

- Remarque:
  - Si on compile sans l'option « **-d** », le fichier **.class** est rangé dans le même répertoire que le fichier **.java** (quel que soit le paquetage auquel appartient la classe)

Placements préconisés pour le développement d'une application



La hiérarchie des **paquetages** produit la même hiérarchie des fichiers compilés.

Commandes à lancer

- Si on se place dans le répertoire racine <Nom du projet>,
  - pour compiler (*en une seule ligne*) C1.java et C2.java:
    javac –d buid/classes –sourcepath src src/P1/*.java
  - pour compiler Essai.java :
    javac –d buid/classes –sourcepath src src/Essai.java
  - pour exécuter :
    java –classpath build/classes Essai
  - pour générer la documentation du paquetage p1 :
    javadoc –d build/doc –sourcepath src p1

- On peut ajouter d'autres répertoires ou fichiers .jar dans le *classpath*
  => utilisation de code externe déjà compilé

Définition du chemin de classe

On spécifie de préférence le chemin de classe avec l'option -classpath (ou -cp)

- **1. Sous Unix/Linux** : séparateur « : »

  java -classpath /home/user/répertoireclasses : . : /home/user/archives/archive.jar MonProgramme

- **2. Sous Windows** :séparateur « ; »

  java -classpath C:\répertoireclasses ; . ; C:\archives\archive.jar MonProgramme

Initialisation du CLASSPATH

- Le CLASSPATH contient par défaut le seul répertoire courant (il est égal à « **.** »)

- **1. Avec le Bourne Again (bash) :**

export CLASSPATH = .:/home/user/répertoireclasses : . : /home/user/archives/archive.jar

- **2. Avec le shell C :**

setenv CLASSPATH .:/home/user/répertoireclasses : . : /home/user/archives/archive.jar

- **3. Avec le shell Windows :**

set CLASSPATH = .;C:répertoireclasses ; . ; C:/archives/archive.jar

Placez votre code Java dans un fichier JAR

- JAR: Java Archive
  - Basé sur pkzip
  - Permet de grouper toutes les classes en un seul fichier
  - Equivalent à la commande tar
  - Directement géré par la machine virtuelle sans extraction
  - 2 utilisations :
    - Archivage d'une librairie (paquetage) à ajouter dans le classpath
    - Archivage d'une application pour exécution directe

Création du JAR

- Pour créer l'archive :

  (c : create / v : verbose / f : fichier destination)
  - à partir du répertoire contenant le bytecode :

    jar –cvf ma_librairie.jar *.class
  - à partir de la racine du projet :

  jar –cvf build/lib/ma_librairie.jar –C build/classes .
- Pour décompresser l'archive *:*

  jar xvf ma_librairie.jar
- Pour plus d'informations :

  jar –help

Utilisation du JAR

- Pour utiliser le JAR, il suffit de l'inclure dans le classpath lors du démarrage de l'application qui l'utilise :

```
java -classpath ma_librairie.jar MonApplication
```

Création d'un JAR exécutable

- Un fichier JAR peut contenir des classes, des images et d'autres ressources, ainsi qu'un fichier manifeste qui décrit les caractéristiques particulières de l'archive.
- Pour archiver une application, il faut ajouter dans le manifeste une ligne déclarant la classe contenant le main (Ex : Essai.class)
  - Déclaration dans un fichier *manifest.txt* :

```
Main-Class: Essai
```

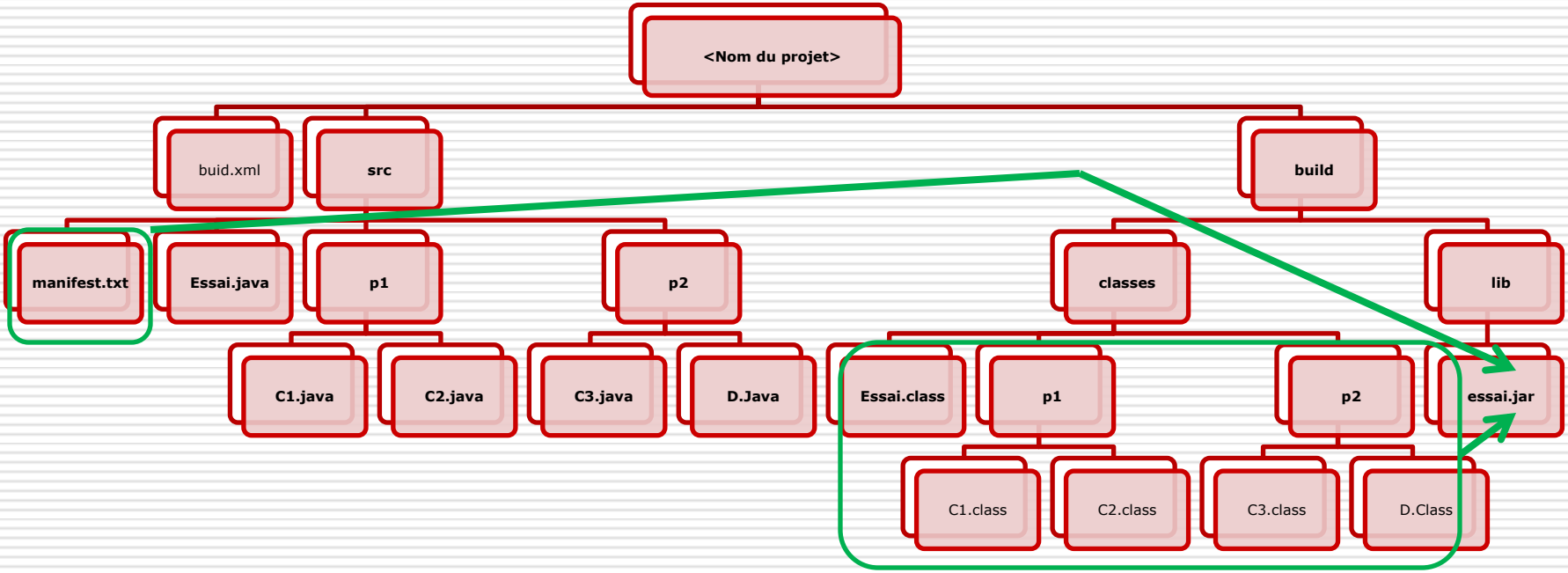
  - Création de *monAppli.jar*

```
jar -cvmf src/manifest.txt
   build/lib/monAppli.jar  -C build/classes .
```

  - Pour exécuter:

```
java -jar build/lib/monAppli.jar
```

Création d'un JAR (2)




```
jar -cvmf src/manifest.txt
   build/lib/monAppli.jar -C build/classes .
```

Classpath et ant

- 1 classpath simple = 1 attribut

```
<java(c) classpath="rep1:rep2:..." ... >
```

- 1 classpath complexe = 1 sous-balise

```
<java(c) ... >
   <classpath>
         <pathelement path="${classpath}"/>
         <fileset dir="lib">
               <include name="**/*.jar"/>
         </fileset>
         <pathelement location="classes"/>
         <dirset dir="${build.dir}">
               <include name="apps/**/classes"/>
               <exclude name="apps/**/*Test*"/>
         </dirset>
         <filelist refid="third-party_jars"/>
   </classpath>
</java(c)>
```

Classpath et ant (2)

Exemple de mutalisation de classpath

- Définition d'un chemin complexe :

```
<path id="lib.externes.path">
        <pathelement ...
</path>
```

- Utilisation à la compilation :

```
<javac classpathref="lib.externes.path"
destdir="${classes.dir}" srcdir=../>
```

- Utilisation à l'exécution : ajouter le répertoire contenant le bytecode

```
<java classname="...">
        <classpath>
                <pathelement location="${classes.dir}"/>
                <path refid="lib.externes.path"/>
        </classpath>
</java>
```