

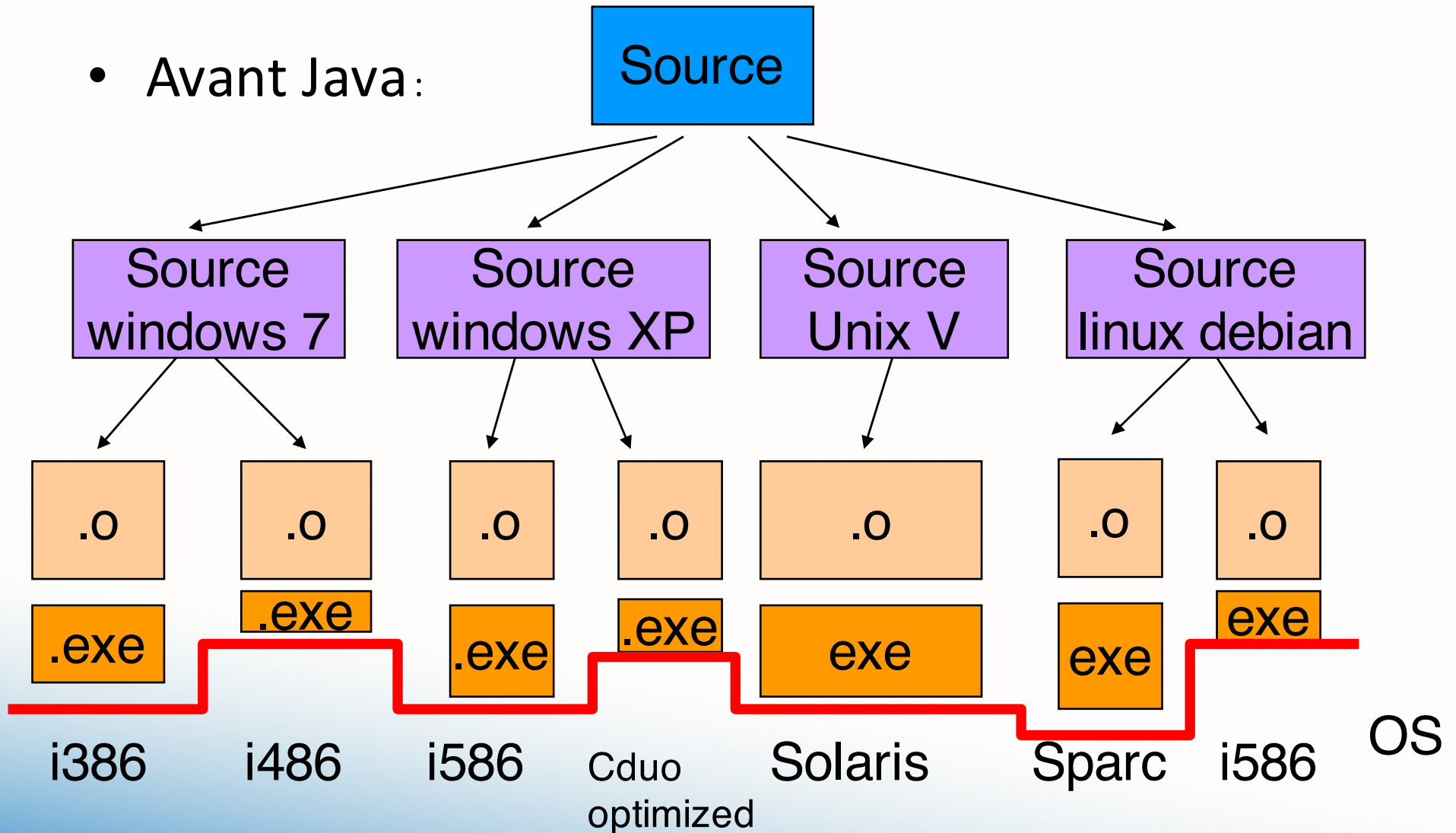


LE LANGAGE JAVA

Cours 1

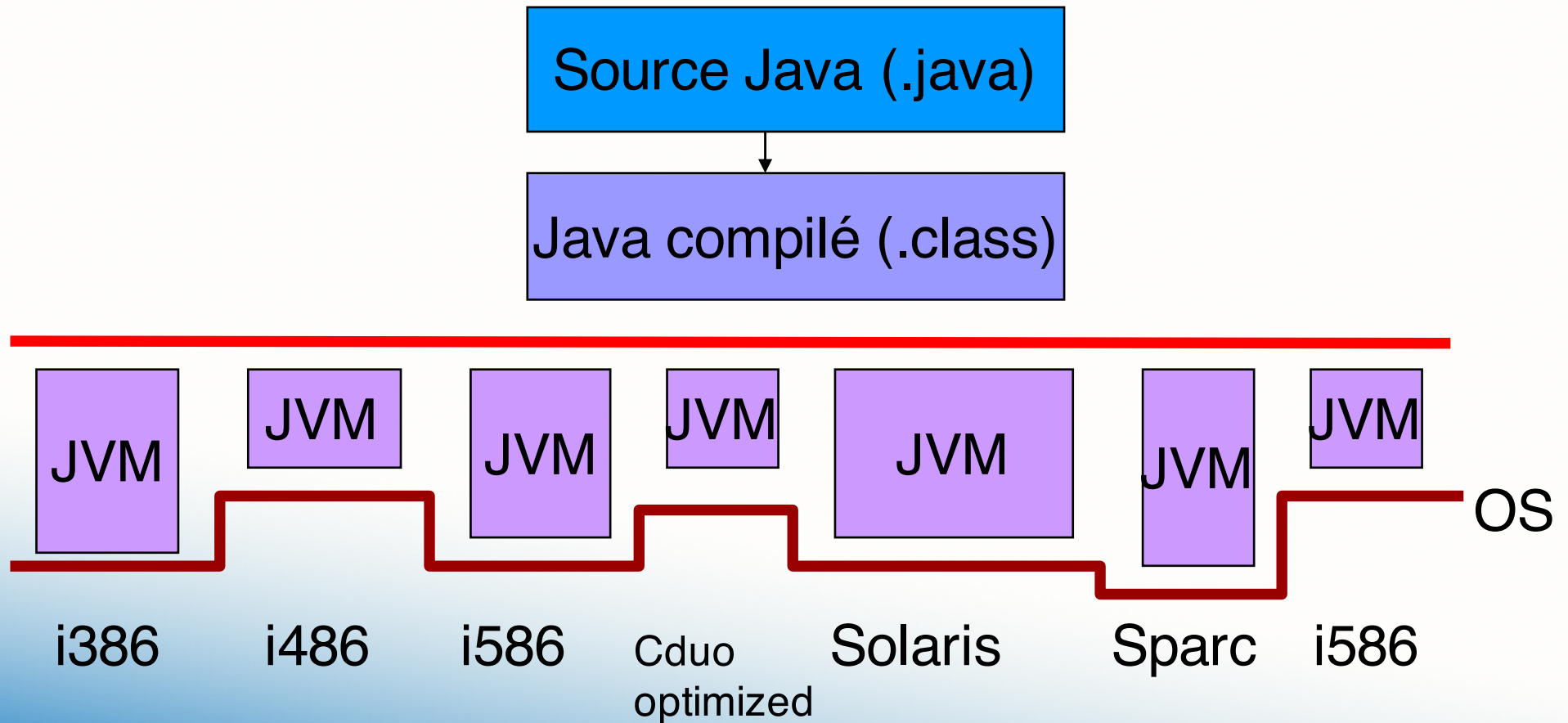
Langage compilé

- Avant Java:



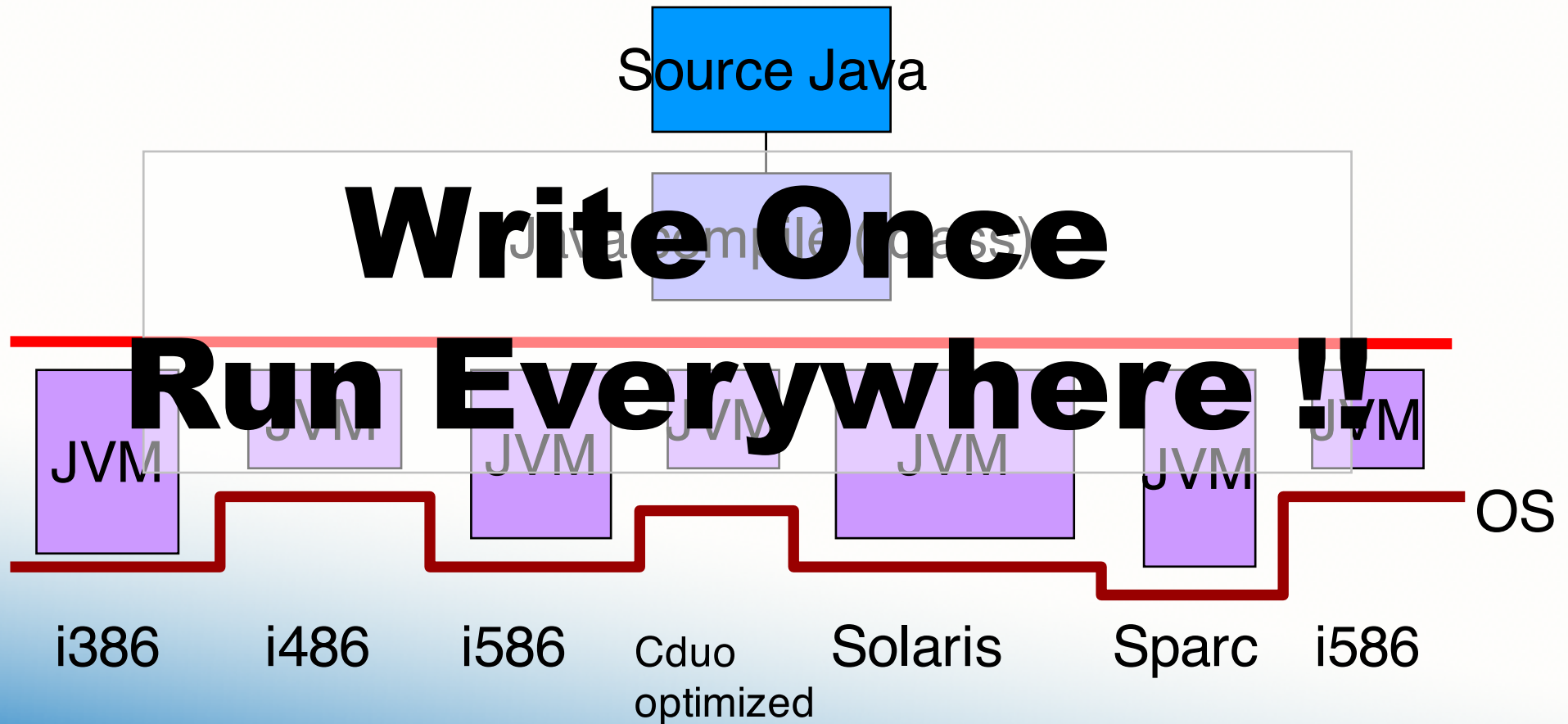
Bytecode Java

- Java:



Bytecode Java

- Java:



Technologie Java

- Un langage de programmation
 - Une syntaxe
 - Une spécification de pseudo-code
 - Un compilateur
- Une plate-forme
 - Une machine virtuelle d'exécution
 - Des outils de production
 - Un ensemble de bibliothèques (API Java)

Ce que Java sait faire

- Outils de développement (javac, java, javadoc)
- L'API Java
- Technologies de déploiement (Java Web Start)
- Boîtes à outils d'interface graphique (AWT, SWING, JFC)
- Bibliothèques d'intégration (JNI, JDBC, RMI, JNDI)

Première application

- Traditionnel « hello world »
- Vu en TD
- Commande de compilation « javac »
- Commande d'exécution « java »
- Ecrire >> Compiler >> Exécuter

Examen de « hello world »

```
/**
 * The HelloWorldApp class implements an application that
 * simply prints "Hello World!" to standard output.
 */
public class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); // Display the string.
    }
}
```

Ant

Ant

- <http://ant.apache.org/>
- équivalent de make pour le développement de projet Java
- basé sur XML
- indépendant du système (et du shell)
- multi-plateformes : Linux, Windows, MacOS, ...

Ant : Principes

- 1 fichier de configuration : build.xml (nom par défaut)
fichier build.xml <--> fichier Makefile
- build.xml contient un ensemble de cibles (target)
- chaque cible contient une ou plusieurs tâches
- dépendances entre cibles

Ant : Intérêts

- compilation
- tâches de pré-compilation
- tâches de post-compilation
- système de fichier
- déploiement d'application répartie
- archivage
- documentation
- exécution
- tests
- tâches distantes

Ant : Architecture (I)

Le développeur crée :

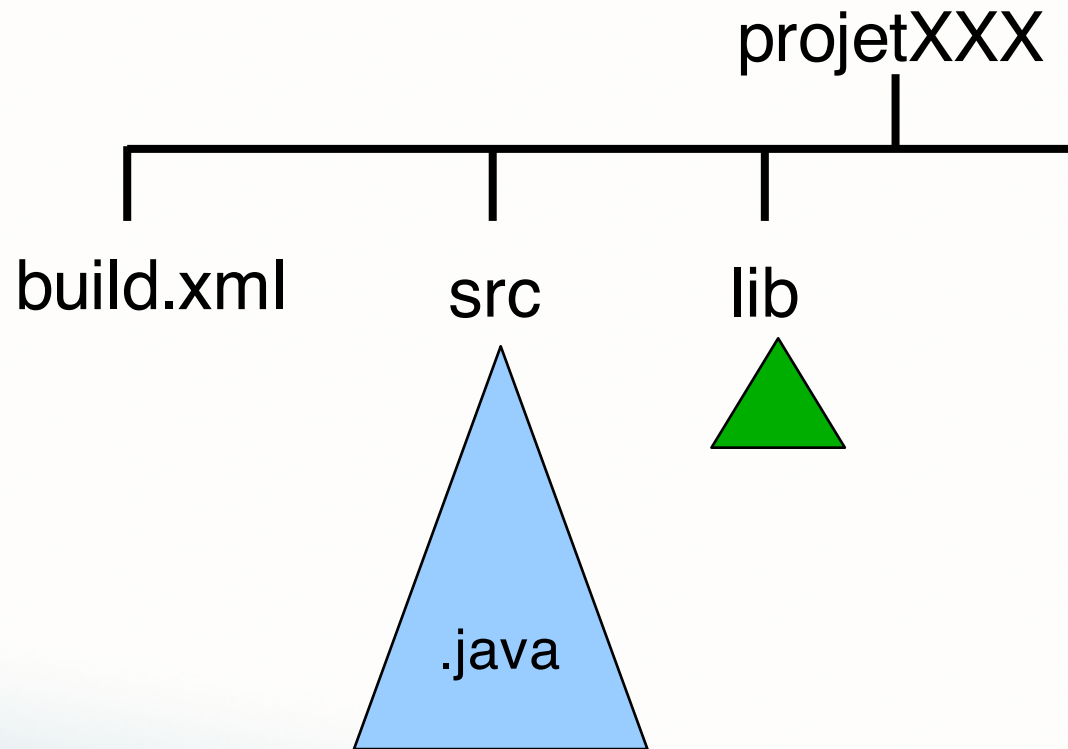
- 1 seul fichier à la racine du projet : build.xml
- 1 répertoire pour les sources du projet Java
- 1 répertoire pour les librairies externes (option)

Ant peut créer :

- 1 répertoire build qui contiendra :
 - ✓ les fichiers compilés (.class)
 - ✓ les librairies générées (.jar)
 - + 1 copie des librairies utilisées (en cas d'inclusion)
 - ✓ La documentation générée

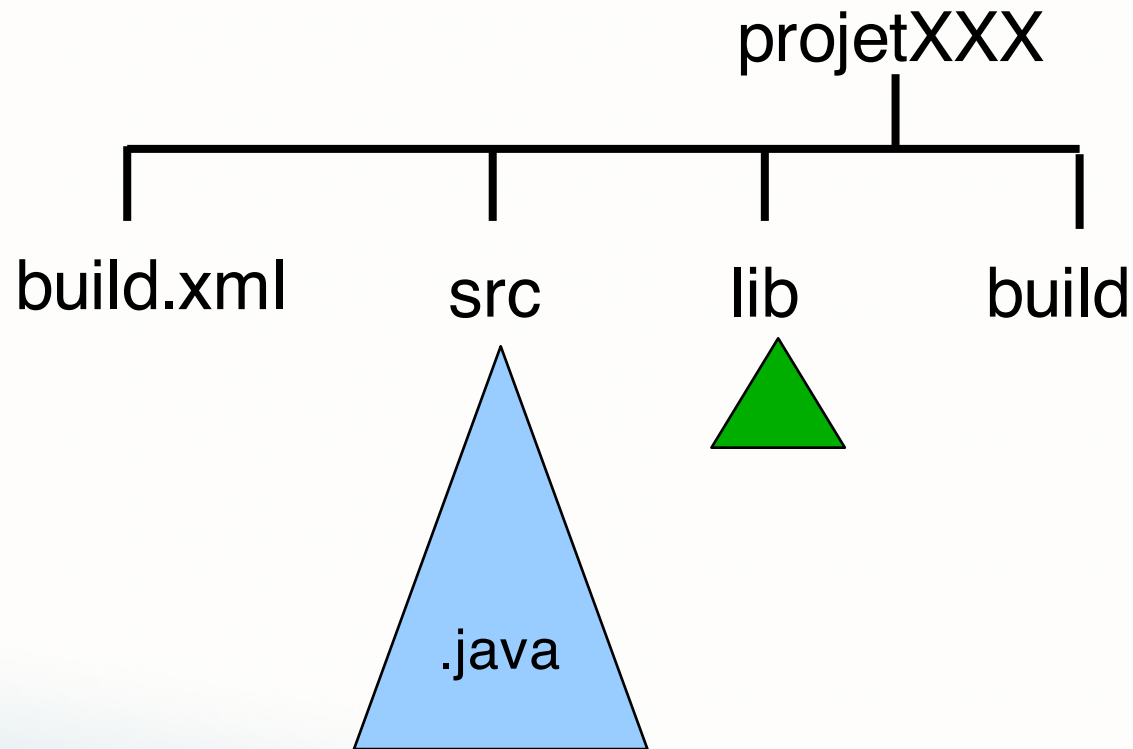
Ant : Architecture (II)

Exemple classique d'architecture d'un projet :



Ant : Architecture (II)

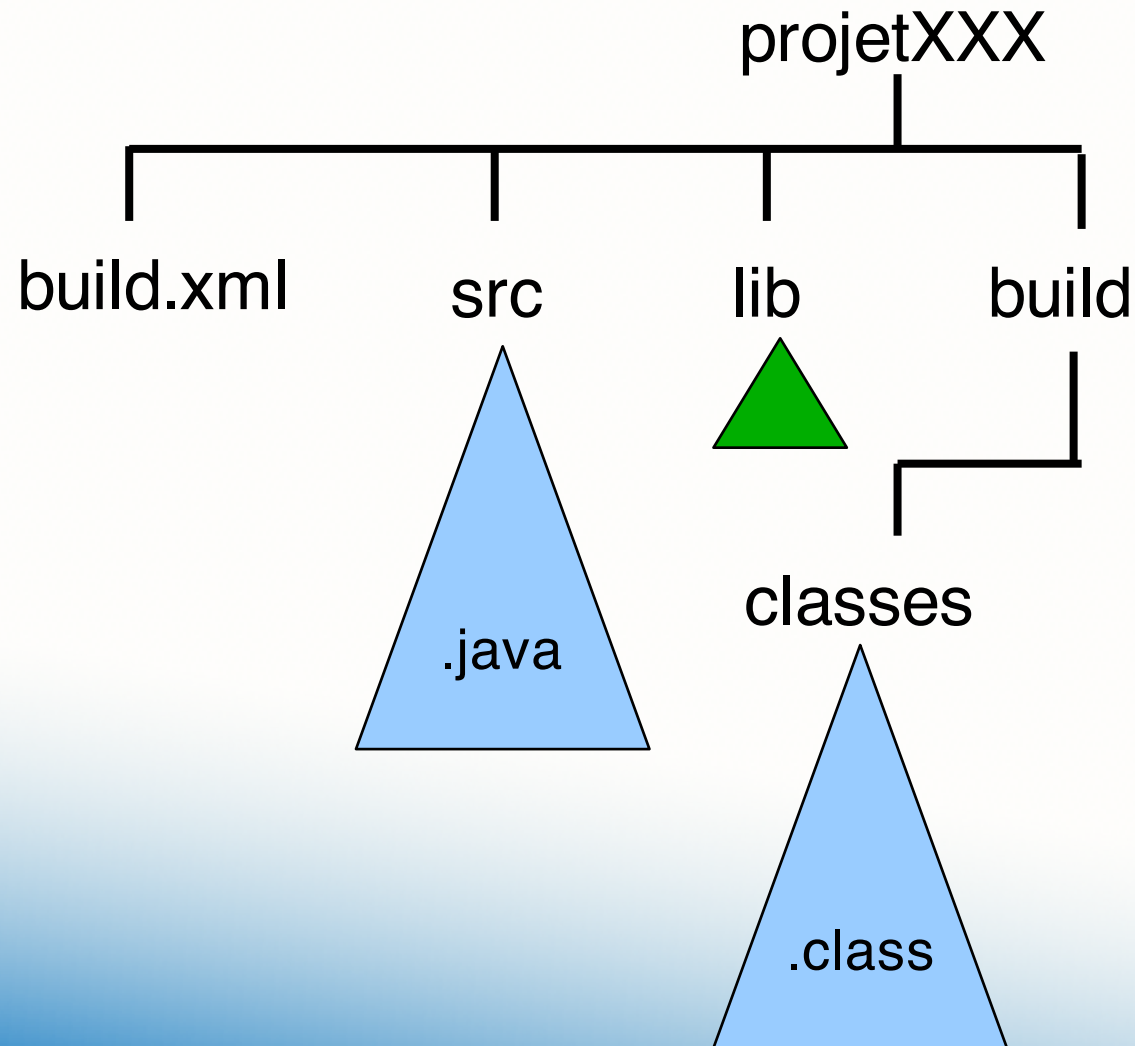
Exemple classique d'architecture d'un projet :



création répertoire
de travail

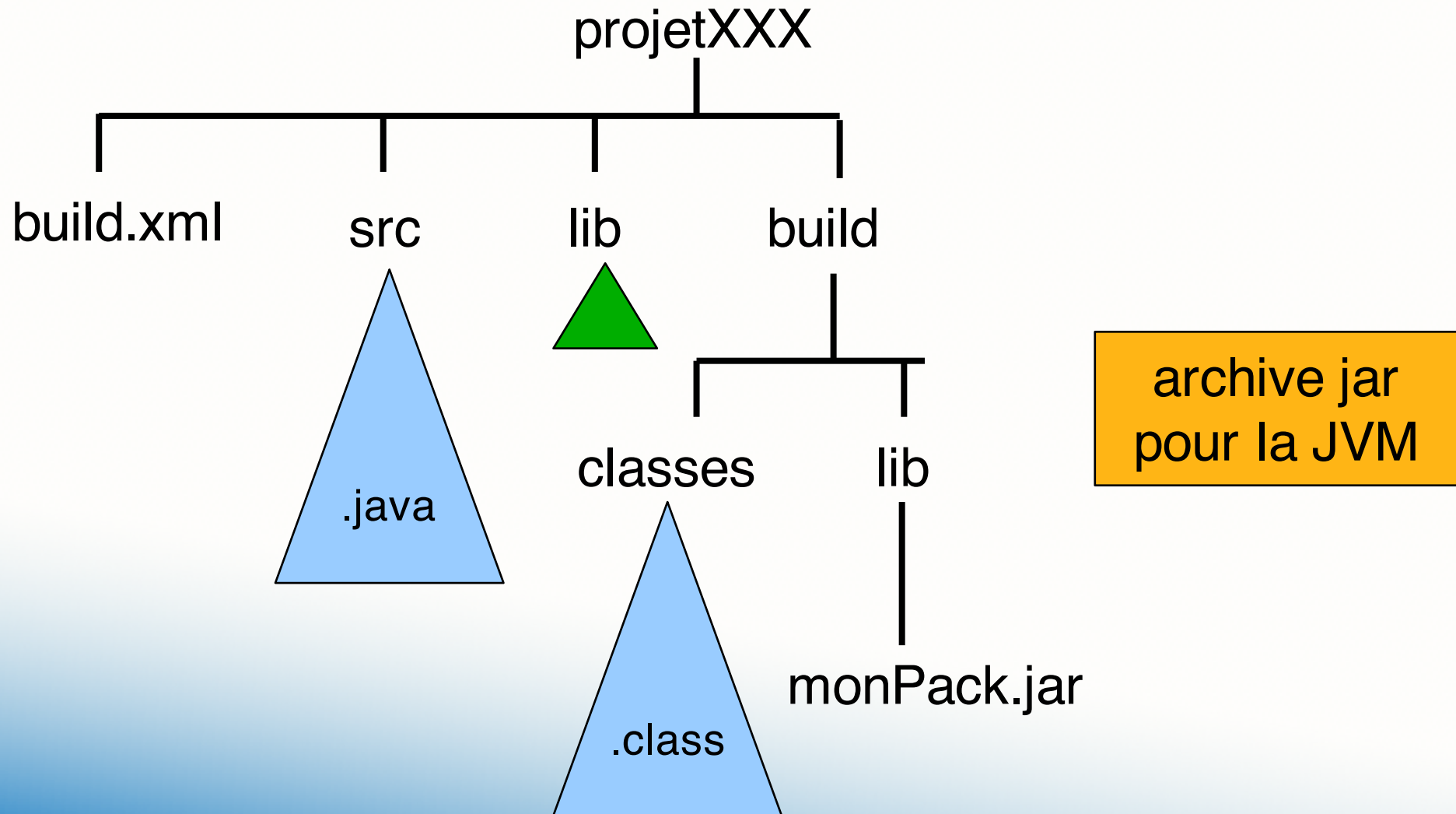
Ant : Architecture (II)

Exemple classique d'architecture d'un projet :



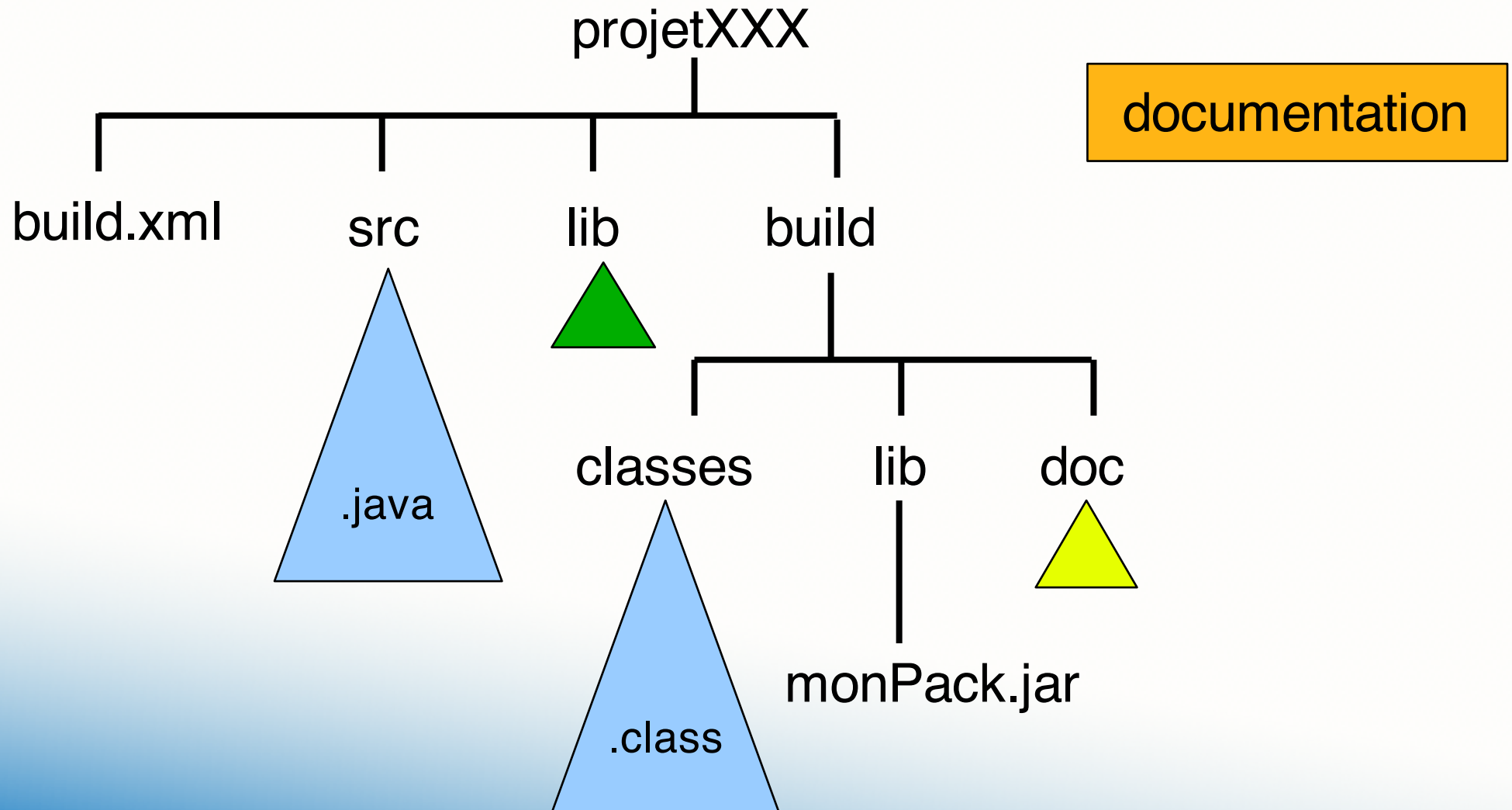
Ant : Architecture (II)

Exemple classique d'architecture d'un projet :



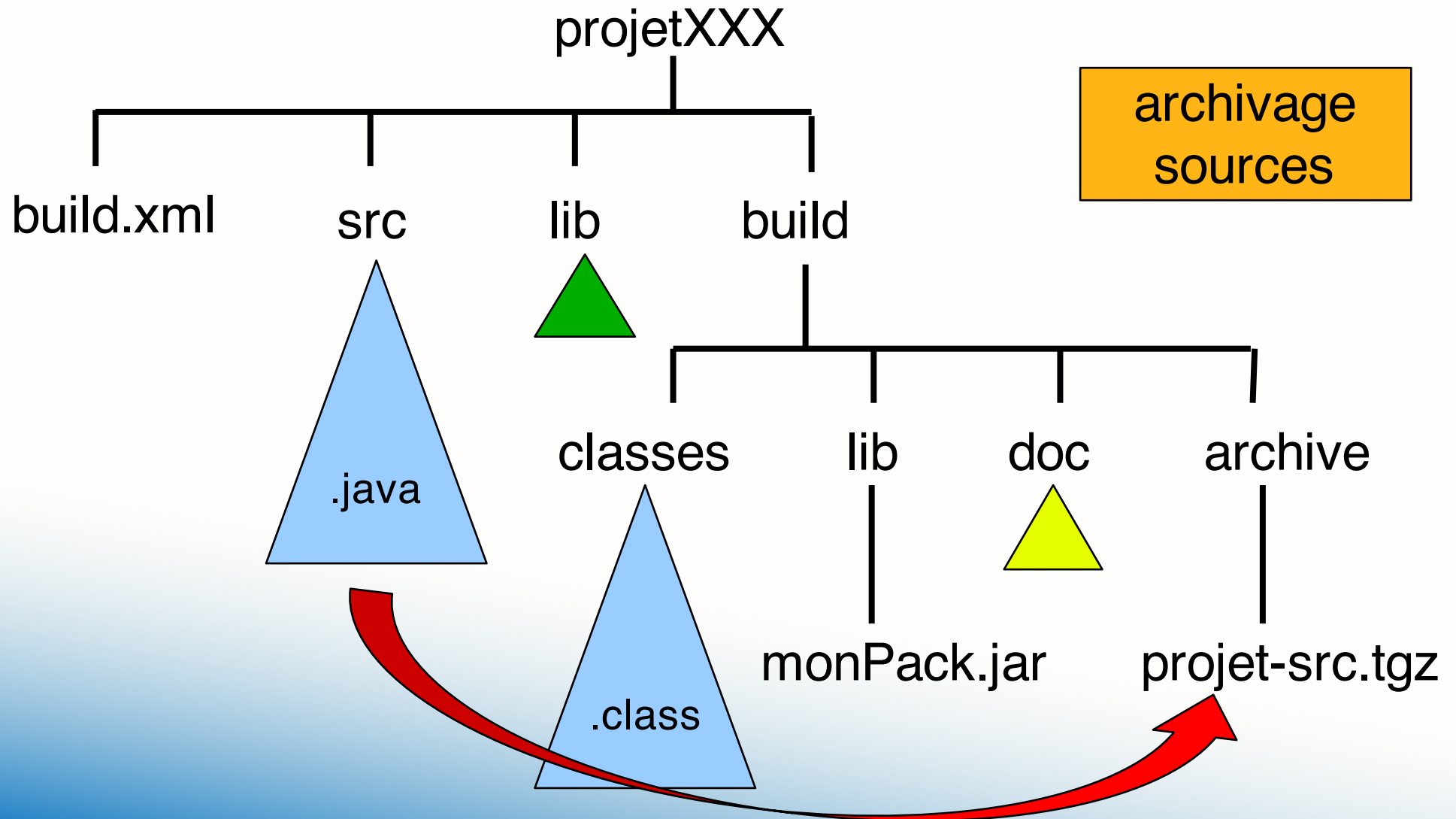
Ant : Architecture (II)

Exemple classique d'architecture d'un projet :



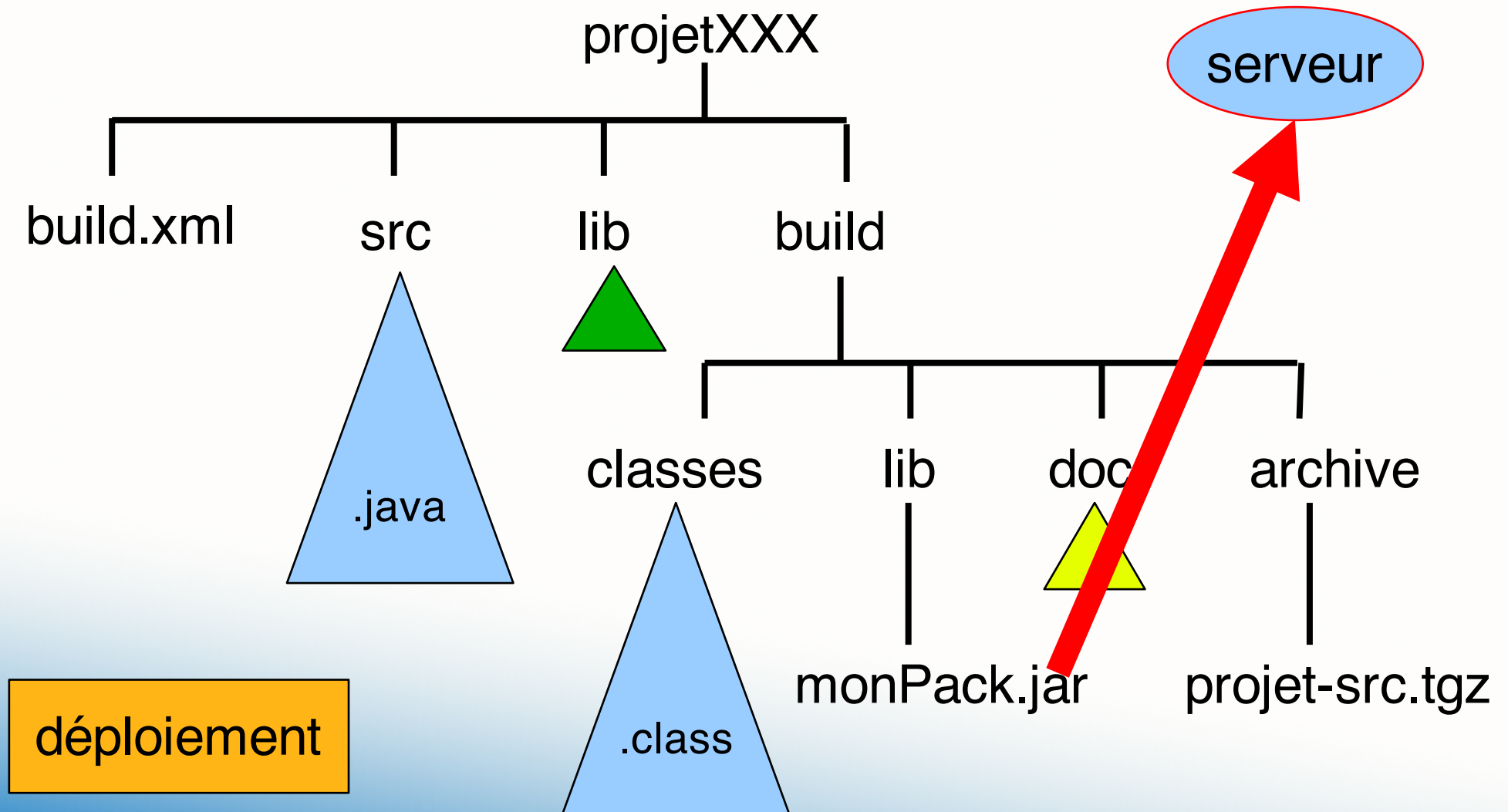
Ant : Architecture (II)

Exemple classique d'architecture d'un projet :



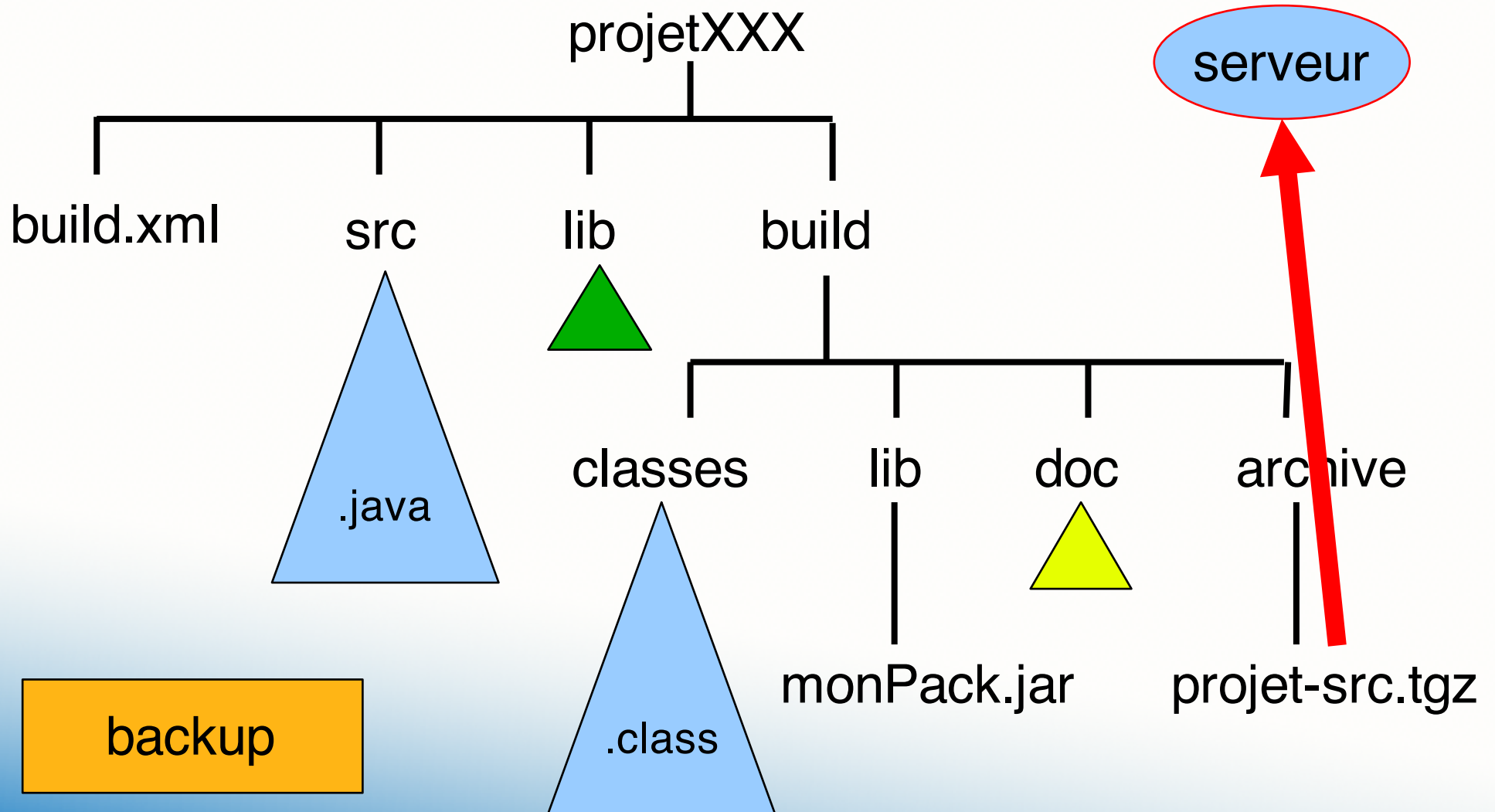
Ant : Architecture (II)

Exemple classique d'architecture d'un projet :



Ant : Architecture (II)

Exemple classique d'architecture d'un projet :



Ant – Intégration

- en ligne de commande :

ant

ant target

ant -f otherBuild.xml

- intégré dans un environnement de développement :

- ✓ JBuilder

- ✓ JDEE (emacs)

- ✓ NetBeans

- ✓ Eclipse

- ✓ Websphere

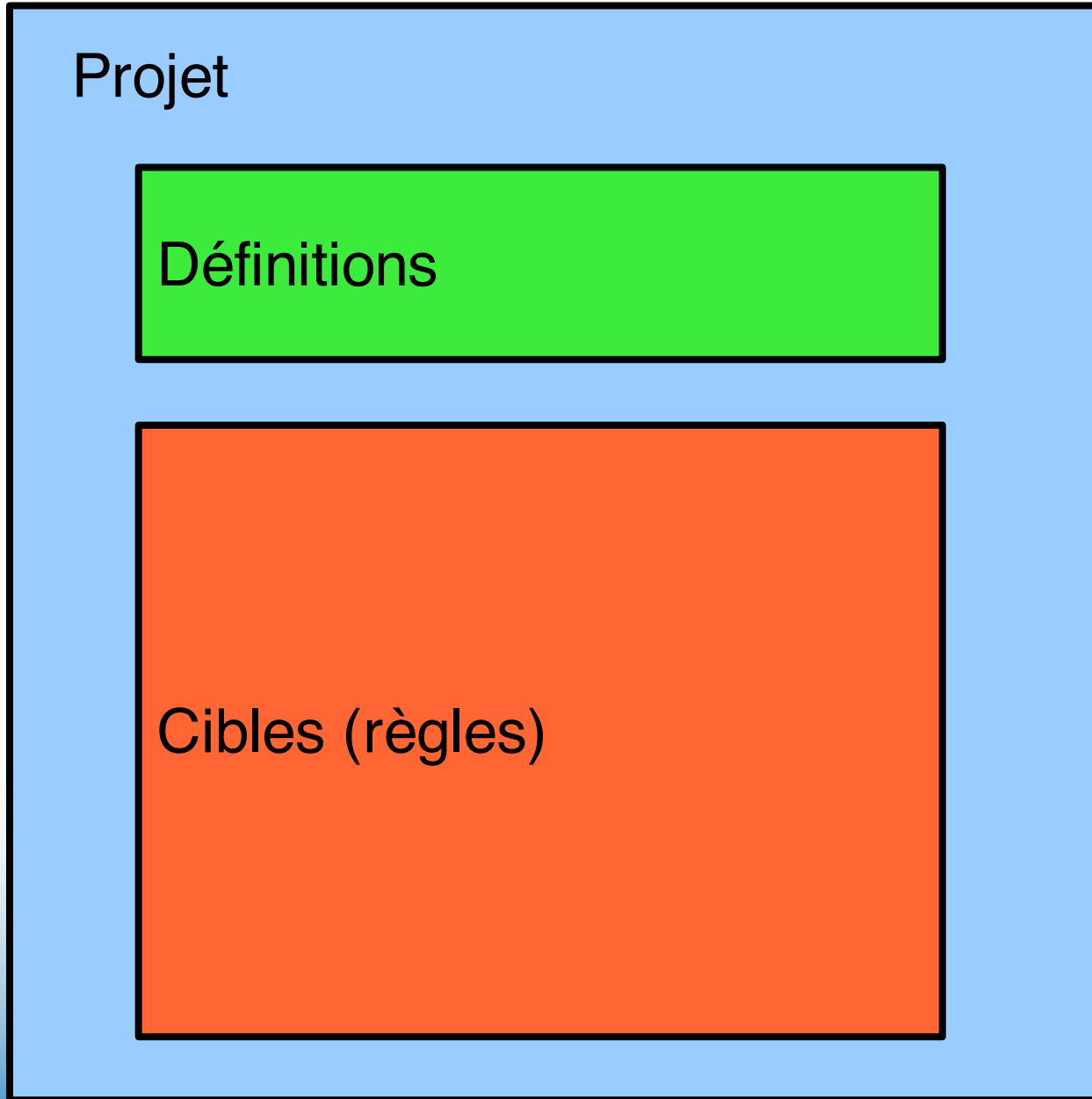
- ✓ jEdit

Ant – build.xml

Projet

Définitions

Cibles (règles)



Ant – build.xml

- fichier xml gérant un projet :

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<project>
```

```
...
```

```
</project>
```

- projet nommé avec une règle par défaut

```
<project name="monProjet" default="main">
```

Ant – cibles

- **1 cible = 1 suite de commandes**

```
<target name="compile">  
    ...  
</target>
```

- **1 cible dépendant d'une autre**

```
<target name="compile" depends="clean">  
    ...  
</target>
```

Ant – commandes

- commande = 1 élément XML
 - ✓ paramètres simples = attributs
 - ✓ paramètres complexes = sous-éléments
- exemples :

```
<mkdir dir="build/classes" />  
<copy todir="../dest/dir">  
  <fileset dir="src_dir">  
    <include name="**/*.java" />  
  </fileset>  
</copy>
```

Ant – Fichiers

➤ copy

➤ move

➤ mkdir

➤ delete

➤ chmod

➤ ...

⇒ appelle la commande correspondante
du système d'exploitation sous-jacent

Ant – compilation et exécution

➤ compilation de tous les sources :

```
<javac srcdir="src" destdir="build/classes"/>
```

➤ compilation de certains sources :

```
<javac srcdir="src" destdir="build/classes"  
  includes="monpackage1/**/* .java"  
  excludes="monpackage1/test/**"/>
```

➤ exécution simple :

```
<java classname="MaClasseApli"  
  classpath="build/classes"  
  fork="true"/>
```

Ant – divers

- création librairie, archivage :

jar

tar, zip, ...

- déploiement, sauvegarde :

scp, ssh

- documentation :

javadoc

- toutes les possibilités (documentation, exemples) :

<http://ant.apache.org/manual/index.html>

Syntaxe

1 - Les variables

- Quatre situations de variables
 - Les champs d'objet
(durée de vie de l'objet - dépendance à l'objet)
 - Les champs de classe
(durée de vie de l'application - dépendance à la classe)
 - Les paramètres de méthode
(durée de vie de la méthode)
 - Les variables locales
(durée de vie de la méthode à partir de leur déclaration)

2 - Les variables (2)

- **Nommage des variables**
 - Noms sensibles à la casse.
 - Caractères alphanumériques (et _ ou \$, déconseillé).
 - Noms longs et complets. (exceptions : i, j ... boucles)
 - Eviter les mots-clefs du langage.

 - Variables à « mot » unique : minuscules.
- Variables à mots multiples : minuscule initiale et majuscule à chaque mot.

```
int speed = 0 ;
```

```
int rotationSpeed = 10 ; //rpm
```

3 - Les variables (3)

- Respecter la culture de nommage, c'est :
 - Ménager sa « mémoire de travail »
 - Se donner à soi-même un plus grand confort de relecture.
 - Donner aux autres un plus grand confort de travail.
 - Eviter des erreurs bêtes.
 - Montrer son appartenance à la communauté métier.
 - Conventions : <http://java.sun.com/docs/codeconv/>

4 - Les types primitifs

- Le Java est **fortement typé**
 - Une variable doit être déclarée.
 - Une variable est associée à un type.
 - Le type représente la nature de la donnée que la variable contient.
 - Java définit huit types « primitifs » pour représenter des données simples (scalaires).
 - Les types primitifs occupent une certaine place en mémoire selon leur définition.

« *Type primitif ≠ Objet* »

5 - LES TYPES PRIMITIFS (2)

- Types courts
 - byte : 8 bits signés
 - int : 32 bits signés
 - float : 32 bits IEEE 754
- Types longs
 - short : 16 bits signés
 - long : 64 bits signés
 - double : 64 bits IEEE 754
- Types non numériques
 - boolean : true, false
 - char : **16 bits Unicode**
- Type quasi primitif
 - String : chaîne de caractères
 - « *String* \Leftrightarrow *java.lang.String* = *Objet* »

6 - Types primitifs (3)

- Valeurs par défaut
 - Champs d'objets : valeur nulle (false)
 - Variables : pas de valeur définie
 - => une variable non initialisée **ne peut être utilisée**
 - => erreur de compilation.
- Initialisation
 - par affectation d'une autre variable
 - par affectation d'une expression littérale

7 - Les tableaux

- Un tableau est un « container statique » pour des objets **de même nature**.
- Le tableau est un « presque objet »

- On le déclare comme une référence :

```
int[] tableauEntiers; // :-)  
int tableauEntiers[]; // :-)
```

- On le construit comme un objet :

```
tableauEntiers = new int[100];
```

- On le dimensionne à la construction.

- On ne peut plus changer sa taille, mais on peut changer les valeurs des éléments.

8 - Les tableaux (2)

- Initialisation statique :

```
int[] tableauEntiers = {3, 2, 10};
```

– la taille est déterminée automatiquement

- Tableaux multidimensionnels. En étendant la notation :

```
int[][] matriceEntiere = {{0, 1}, {-1, 0}};
```

- Manipulation des éléments par notation indicée classique :

```
tableauEntiers[1] = 5;  
matriceEntiere[0][1] = 2;
```

9 - Les tableaux (3)

- Attribut pseudo-objet : *length*

```
System.out.println(tableauEntier.length)
```

- Copie de tableau

- ✓ `tableau1 = tableau2;`

- ... ne copie pas le tableau.

- ✓ `System.arraycopy(tableau1, 0, tableau2, 0, 3);`

- `from, start, to, start, size`

- ... copie le tableau.

- les tableaux doivent être de même nature.

10 - Les opérateurs

- Opérateurs du langage C.
- Arité des opérateurs : nombre d'opérandes concernés.
 - opérateurs unaires (6).
 - opérateurs binaires (le reste).
 - opérateurs ternaires (1).
- « *Précédence des opérateurs* » : les règles qui guident l'ordre d'évaluation dans une expression de calcul

11 - Les opérateurs (1)

- Les opérateurs (sauf l'affectation) forment une expression calculée.
- Une expression calculée est assimilable à une variable du type final de l'expression
- Un opérateur (=) permet de transférer toute valeur (calculée ou "contenue") dans une variable d'arrivée.

12 - Les opérateurs (2)

- Sommaire des opérateurs unaires
 - postfixes : réalisent l'opération après l'évaluation de l'instruction

```
int ordre = 3;  
int ancienOrdre = ordre++;  
System.out.println(ancienOrdre);  
System.out.println(ordre);
```

```
> 3  
4
```

13 - Les opérateurs (3)

- Sommaire des opérateurs unaires (suite)

- changement de signe : inversion du signe

```
int ordre = 3;  
System.out.println(-ordre);  
> -3
```

- négation logique :

```
boolean flag = true;  
System.out.println(!flag);  
> false
```

- complément à 1 :

```
byte octet = 0x55;  
System.out.format("%x%n", ~octet);  
> AA
```

14 - Les opérateurs (4)

- Sommaire des opérateurs arithmétiques
 - opérateurs arithmétiques usuels : +, -, *, /
 - Modulo : %
 - Affectations composées : +=, -=, *=, /=, ...
 - Opérateur de concaténation (arithmétique des chaînes) : + utilisé sur des types chaîne (ou assimilés).

15 - Les opérateurs (5)

- Sommaire des opérateurs logiques
 - opérateurs logique (conditionnels): `&&` (et), `||` (ou)
 - opérateurs relationnels :
`<`, `>`, `<=`, `>=`, `==`, `!=`
 - opérateur de comparaison de type objet : `instanceof`
 - `instanceof` ne compare pas "l'exactitude de type" mais la compatibilité de type.

16 - Les opérateurs (6)

- Opérateurs bit

$\&$, \wedge , $|$

Bit à bit AND, XOR, OR

- Opérateurs de décalage

- \ll , décalage à gauche (insertion de zéros) $\Leftrightarrow * 2$
- \gg , décalage à droite (insertion du signe) $\Leftrightarrow / 2$
- \ggg , décalage à droite (insertion de zéros)

17 - Expressions, instructions, blocs

- Expression
 - Une expression est une construction syntaxique qui utilise des règles de grammaire.
 - Une expression évalue une valeur finale.
 - La valeur finale est d'un certain type qui dépend des règles de grammaire.
 - Une expression est syntaxiquement équivalente à une variable du même type.

18 - Expressions, instructions, blocs (2)

- Instruction
 - Une instruction est une unité élémentaire d'exécution.
 - Une instruction peut être élémentaire (simple) ou complexe.
 - Une instruction simple n'a que quelques formes possibles :
 - Expressions d'affectation
 - Toute utilisation de ++ ou --
 - Une invocation de méthode
 - Une expression de création d'objet
 - Déclaration de variable
 - Instruction de contrôle de flux
 - Une instruction termine par un ";" (point-virgule)

19 - Expressions, instructions, blocs (3)

- Bloc
 - Un bloc rassemble des instructions.
 - Un bloc est à la base des instructions complexes.
 - Un bloc constitue souvent une frontière de « portée ».
 - Le bloc structure le programme en « chemins » d'exécution, choisis selon le résultat de certaines structures de contrôle. (contrôle d'exécution).

```
{  
    instruction1;  
    instruction2;  
    ...  
}
```

20 - Structures conditionnelles

- *If (cond) expr ; / bloc*
 - La condition doit avoir un type évalué `boolean` (\neq langage C)
 - 1 chemin optionnel d'exécution
- *If (cond) expr ; / bloc else expr ; / bloc*
 - 2 chemins alternatifs d'exécution
- **Pas d'instruction `elsif` ni `elseif`**

21 - Structures conditionnelles (2)

- Pourquoi on préfère un bloc :
 - écriture initiale :

```
if (uneCondition)
    ... une instruction ... ;
```

- écriture corrective fréquente :

```
if (uneCondition) {
    ... une instruction ... ;
    ... une instruction complémentaire ... ;
}
```

22 - Branchements

- L'instruction switch
 - Règles identiques au C :

```
switch (nomVariable) {  
    case valeur1:  
        ... instructions ...  
        break;  
    case valeur2: ...  
    default: ....  
}
```

- Avantageuse d'un point de vue performances.
- La variable « sélecteur » doit être à valeur entière ou énumérable (production d'un offset dans une table de branchement)

23 - Boucles indéterminées

- *while (cond) bloc*
 - Ne passe pas forcément par le bloc de boucle.
 - La condition doit avoir un type évalué `boolean`
- *do bloc while (cond) ;*
 - Passe forcément une fois par le bloc de boucle.
 - La condition doit avoir un type évalué `boolean`

24 - Boucles déterminées

- Forme standard :
 - *for (init ; cond ; iter) bloc*
 - S'utilise quand :
 - la boucle fonctionne sur un principe de "comptage"
 - on connaît bien la condition de fin de boucle

```
int[] t = {3, 2, 10, 25, 10};  
int i;  
  
for (i = 0 ; i < t.length ; i++) {  
    System.out.println(i + " - " + t[i]);  
}
```

24 - Boucles déterminées

- Exploration de collection :
 - *for (type variable : collection) bloc*
 - La variable explore la collection prenant la valeur de ses éléments

```
int[] t = {3, 2, 10, 25, 10};  
  
for (int element : t) {  
    System.out.println(element);  
}
```

- *collection = tableau, liste, ... = tout ce qui est « Iterable »*

25 - Déroutements

- Sortie de méthode :
 - Sans valeur : `return;`
 - Avec valeur : `return expr;`