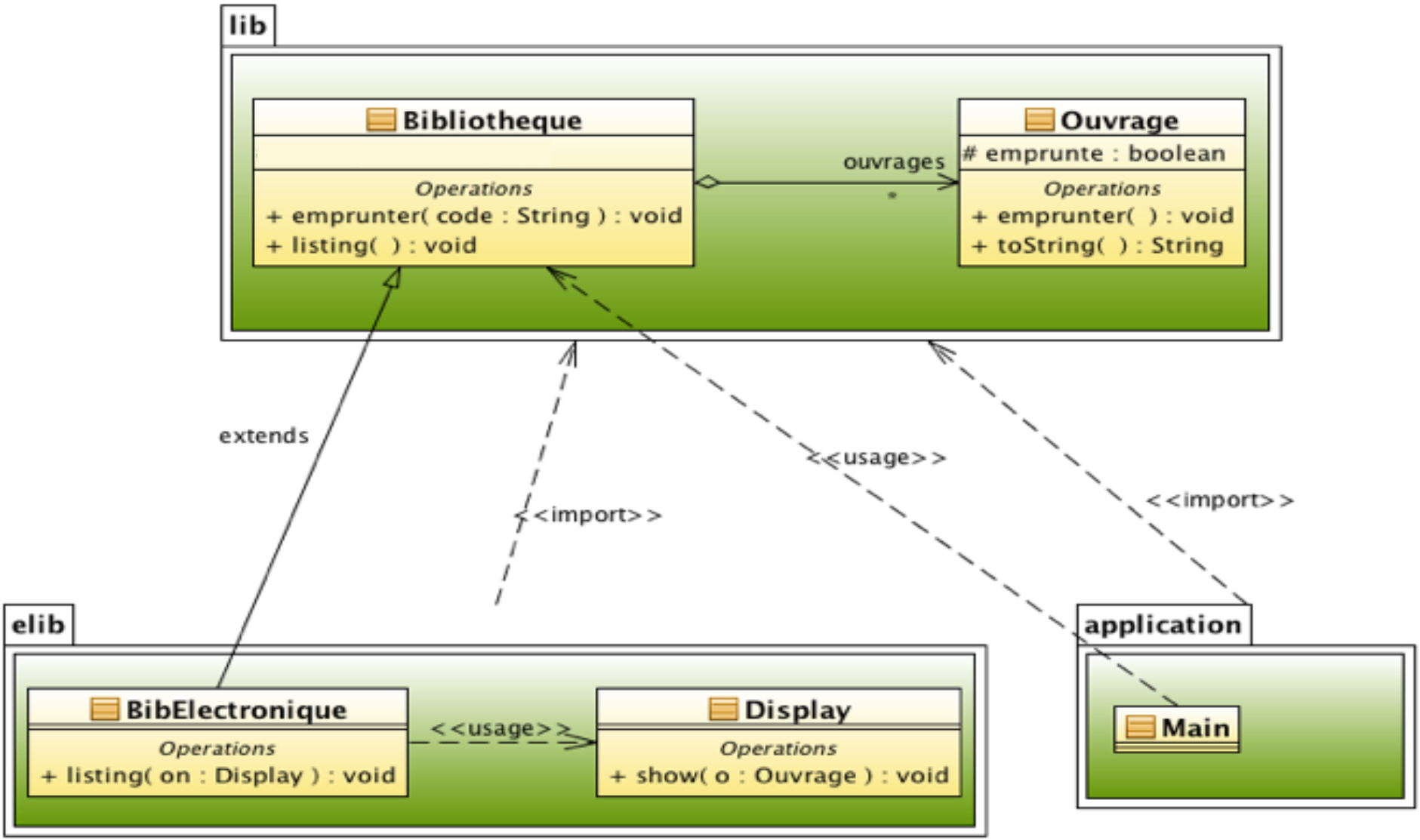
MODULARITÉ DE PACKAGES

packages et modifiers de visibilité

Notation UML



Packages : niveau logique

- Niveau logique
  - Regroupement de classes (et d'interfaces) reliées logiquement
  - Modules de niveau supérieur aux classes et aux objets
  - Structuration modulaire de logiciels de grande taille
  - Structuration des bibliothèques: `java.applet, java.awt, java.io, java.lang, java.math, java.net, java.util, ...`
- Niveau physique
  - A chaque package correspond un répertoire du système sous-jacent
  - Les chemins d'accès aux packages sont rangés dans la variable d'environnement `CLASSPATH`, semblable à `PATH` pour les commandes)
  - ou passés en paramètres aux commandes `javac` et `java` : options `-classpath` ou `-cp`

Création de packages

`package <nomDePackage>;`

- « range » logiquement les classes du fichier dans le package
  - d'où le nom complet d'une classe (nécessaire si ambiguité): `<nomDePackage>.<nomDeClasse>`
- Si aucun package n'est spécifié
  - les classes appartiennent à un package par défaut résidant sur le répertoire courant (.)
  - elles ne peuvent être importées ailleurs.
- Seules les classes déclarées public d'un package (autre que « . ») sont “import“-ables dans d'autres packages
- Les autres sont encapsulées dans le package (classes auxiliaires d'implantation).

Utilisation d'une classe d'un package

- Lorsqu'on fait référence à une classe,
  - le compilateur la recherche dans le package par défaut.
  - Si elle appartient à un autre package, il est nécessaire de fournir l'information
- En citant le nom du package avec le nom de la classe
  - `bib.Ouvrage o = new bib.Ouvrage();`
  - Evidemment, cette démarche devient fastidieuse dès que de nombreuses classes sont concernées.
- En important une classe
  - Instruction import au début du fichier (juste après l'instruction `package`)
  - `import <nomDePackage>.<NomDeClasse>;`
  - Utilisation de la classe sans mentionner son nom de package.
  - Démarche fastidieuse dès qu'un certain nombre de classes d'un même package sont concernées.
- En important un package
  - `import <nomDePackage>.*;`
  - Possibilité ensuite d'utiliser toutes les classes du package
  - physiquement les répertoires correspondants doivent être accessibles par l'un des chemins du `CLASSPATH`
- pas d'importation (inclusion) « physique » de code :
  - à la compilation : résolution de noms et typage
  - à l'exécution : chargement dynamique du bytecode

Import : exemples

- Applet

```
import java.applet.*;
import java.awt.*;
public class Salut extends Applet {//java.applet.Applet
  public void paint(Graphics g) {//java.awt.Graphics
        g.drawString("Salut!",20,20);
  }
}
```

- Collections

```
import java.util.*;
public class Bibliotheque {
 protected Map<String,Ouvrage> ouvrages
   = new TreeMap<String,Ouvrage>();
...}
```

- `java.lang`
  - Ce package est automatiquement importé par le compilateur
  - Possibilité d'utiliser des classes standards telles que `Math`, `System`, `Float`, ou `Integer` sans introduire d'instruction `import`.

Exemple : accès aux classes

- package lib : Ouvrage, Bibliotheque et Exception

```
//fichier ./lib/NonDisponibleException.java
package lib;
public class NonDisponibleException extends Exception{}

//fichier ./lib/Ouvrage.java
package lib;
public class Ouvrage { ...
 public void emprunter() throws NonDisponibleException …

//fichier ./lib/Bibliotheque.java
package lib;
import java.util.*;
public class Bibliotheque {
 protected Map<String,Ouvrage> ouvrages …
 public void emprunter(String code) throws
  NonDisponibleException…
```

Exemple

- Utilisations des packages

```
//fichier ./elib/BibElectronique.java
package elib;
import lib.*;
public class BibElectronique extends Bibliotheque {…}

//fichier ./Application.java : sans package => package .
import lib.*;
import elib.*;
public class Application {
 public static void main(String[] argv) {
  Ouvrage x; //<=> lib.Ouvrage x;
  Bibliotheque bib = new Bibliotheque(); //lib.Bibliotheque
  BibElectronique ebib;  //elib.BibElectronique
  …
```

Encapsulation

- régler le degré d'encapsulation/de visibilité entre classes
  - des variables d'instance : en général masquées pour cacher l'implantation
  - des méthodes (et constructeurs) :
    - internes : accessoires d'implantation
    - publiques : protocole ou interface de manipulation

Visibilité entre classes	d'un même package	de packages distincts
`public`	oui	oui
`protected` (« subclass limited »)	oui	restreint aux sous-classes
Aucun (« package limited »)	oui	non
`private`	non	non

Exemple : public/private

```
package complexes;
public class Complexe {
  private double re, im;
  public Complexe (double x, double y) {re=x; im=y;}
  public double re() {return re;}
  public Complexe add(Complexe c) {
        return new Complexe(re+c.re(),im+c.im());}
...}

// meme package
// ou non (moyennant: import complexes.*)
public class Test {
  public static void main(String argv[]) {
        Complexe c = new Complexe(10.0,20.0); // public => ok
        ... c.re ... // erreur : private => non visible
        ... c.re= ...  // encore moins!
        ... c.re() ... // public => ok
}
```

Exemple : public/protected

```
package elib;
import lib.*;
public class BibElectronique extends Bibliotheque {
  public void listing(Display display) {
   for(Ouvrage o: ouvrages.values())
      // protected dans sous-classe OK
      display.show(ouvrages.get(code));
    }
}
```

Accès au champ `protected`.

```
import lib.*;
public class Application { //utilisatrice non sous-classe
 public static void main(String[] argv) {
 Bibliotheque bib = new Bibliotheque();
 bib.listing(); // public OK
 // bib.ouvrages.get(code).emprunte = false;
 // Impossible de tricher! : protected's hors sous-classe
```

Pas accès au champ `protected`.

Quelques règles…

- Par défaut au sein d'un même package, tout est visible. Java part du principe qu'au sein d'un package, on est entre amis (`friend` C++ :-)
- on ne peut redéfinir une méthode «plus privée» dans une sous-classe
- les modifiers Java sont unitaires contrairement à C++ où l'on peut les faire porter sur un groupe de caractéristiques.
- la déclaration public d'une classe n'a aucune conséquence sur les modalités d'encapsulation de ses caractéristiques
- le modifier **final** appliqué à :
  - une classe : la rend non-extensible (ex. System)
  - une méthode : la rend non-redéfinissable
  - une variable initialisée : constante.
- dans les documentations utilisateurs de classes (API Java ou les vôtres), seules les caractéristiques accessibles (public ou protected) apparaissent.