Programmation Orientée Objet

Bertrand Estellon

Département d'Informatique de Luminy
Aix-Marseille Université

29 mai 2012

La programmation orientée objet (POO)

Les objectifs :

- Faciliter le développement et l'évolution des applications ;
- Permettre le travail en équipe ;
- Augmenter la qualité des logiciels (moins de bugs).

Solutions proposées :

- Découpler (séparer) les parties des projets ;
- Limiter (et localiser) les modifications lors des évolutions ;
- Réutiliser facilement du code.

Le langage Java (utilisé dans ce cours)

Le langage Java :

- est un langage de programmation orienté objet
- créé par James Gosling et Patrick Naughton (Sun)
- présenté officiellement le 23 mai 1995.

Les objectifs de Java :

- simple, orienté objet et familier ;
- robuste et sûr ;
- indépendant de la machine employée pour l'exécution ;
- très performant ;
- interprété, multi-tâches et dynamique.

Autres langages orienté objet

- **C++** : très utilisé
- **C#** : langage de Microsoft (appartient à .NET)
- **Objective C** : langage utilisé par Apple
- **PHP** : langage très utilisé sur le Web
- **Python**
- **Ruby**
- Eiffel
- Ada
- Smalltalk
- ...

La syntaxe change mais le concept objet est le même !

Mon premier programme Java

Le programme *HelloWorld.java* :

```
class HelloWorld {
  public static void main(String arg[]) {
    System.out.println("Hello world !");
  }
}
```

Compilation et exécution :

```
$ javac HelloWorld.java
$ ls
HelloWorld.java HelloWorld.class
$ java HelloWorld
Hello world !
```

Commentaires

```
class HelloWorld {
  public static void main(String arg[]) {
    /* Commentaire
       sur plusieurs lignes.
    */
    // sur une seule ligne.
    System.out.println("Hello world !");
  }
}
```

Les types primitifs

byte	entier	8 bits	-128 à 127	0
short	entier	16 bits	-32768 à 32767	0
int	entier	32 bits	-2^{31} à $2^{31}-1$	0
long	entier	64 bits	-2^{63} à $2^{63}-1$	0
float	flotant	32 bits		0.0
double	flotant	64 bits		0.0
char	caractère	16 bits	caractères Unicode	\u0000
boolean	boolean	1 bit	false ou true	false

```
int a = 12;
double b = 13.5;
```

Syntaxe

Très proche de celle du C :

```
class PetitProgramme {
  public static void main(String arg[]) {
    for (int i = 0; i < 100; i++)
        System.out.println(i);
    boolean b = true;
    int k = 1;
    while (b) {
        if (k%100==0) b = false;
        k++;
    }
  }
}
```

Les structures du C

En C, la déclaration d'une structure permet de définir ses champs :

```
struct DeuxEntiers {
    int x;
    int y;
}
typedef struct DeuxEntiers DeuxEntiers;
```

L'allocation de différentes zones mémoires utilisant cette structure :

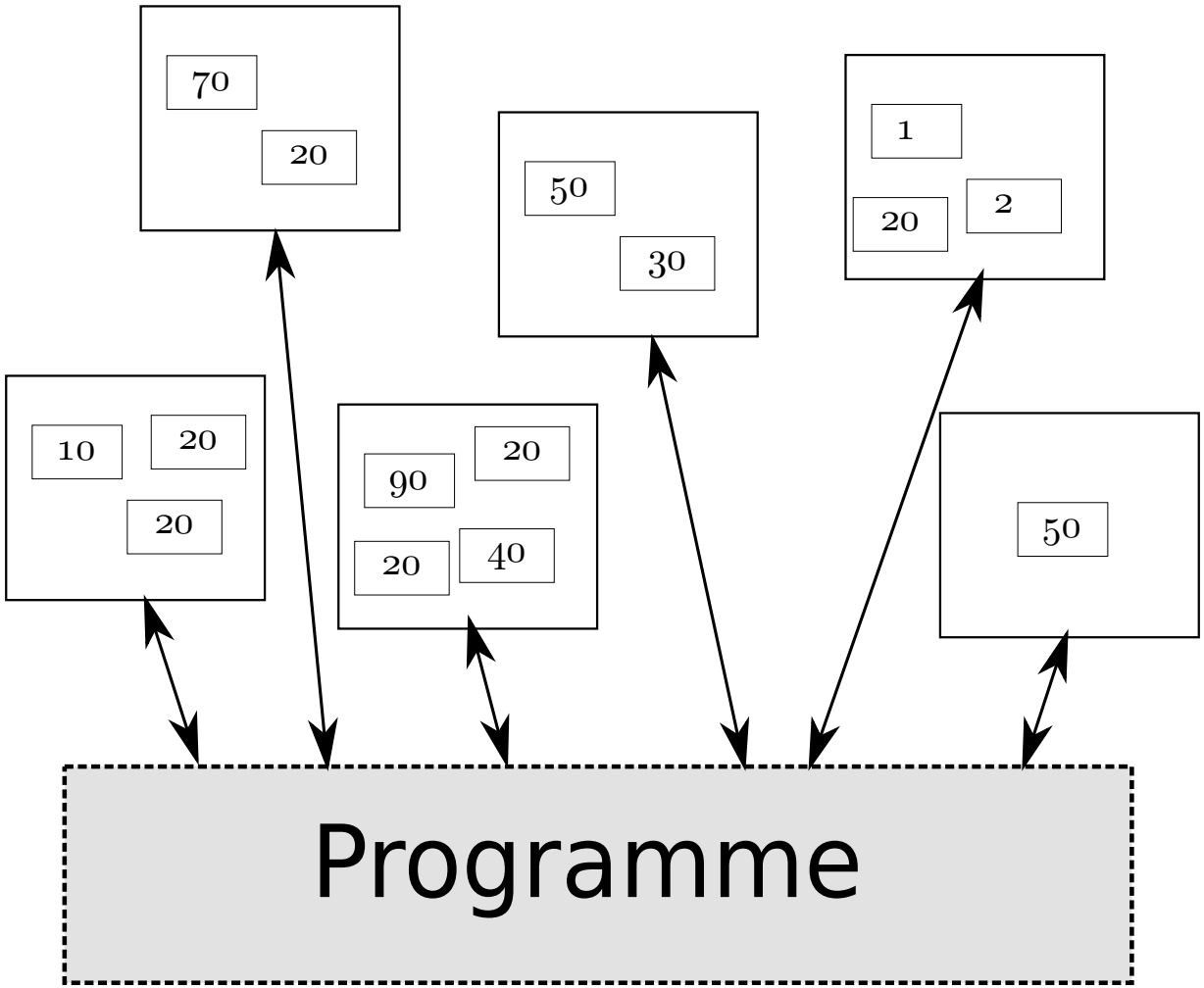
```
DeuxEntiers *p1 = malloc(sizeof *p1);
DeuxEntiers *p2 = malloc(sizeof *p2);
```

Accès aux données des structures :

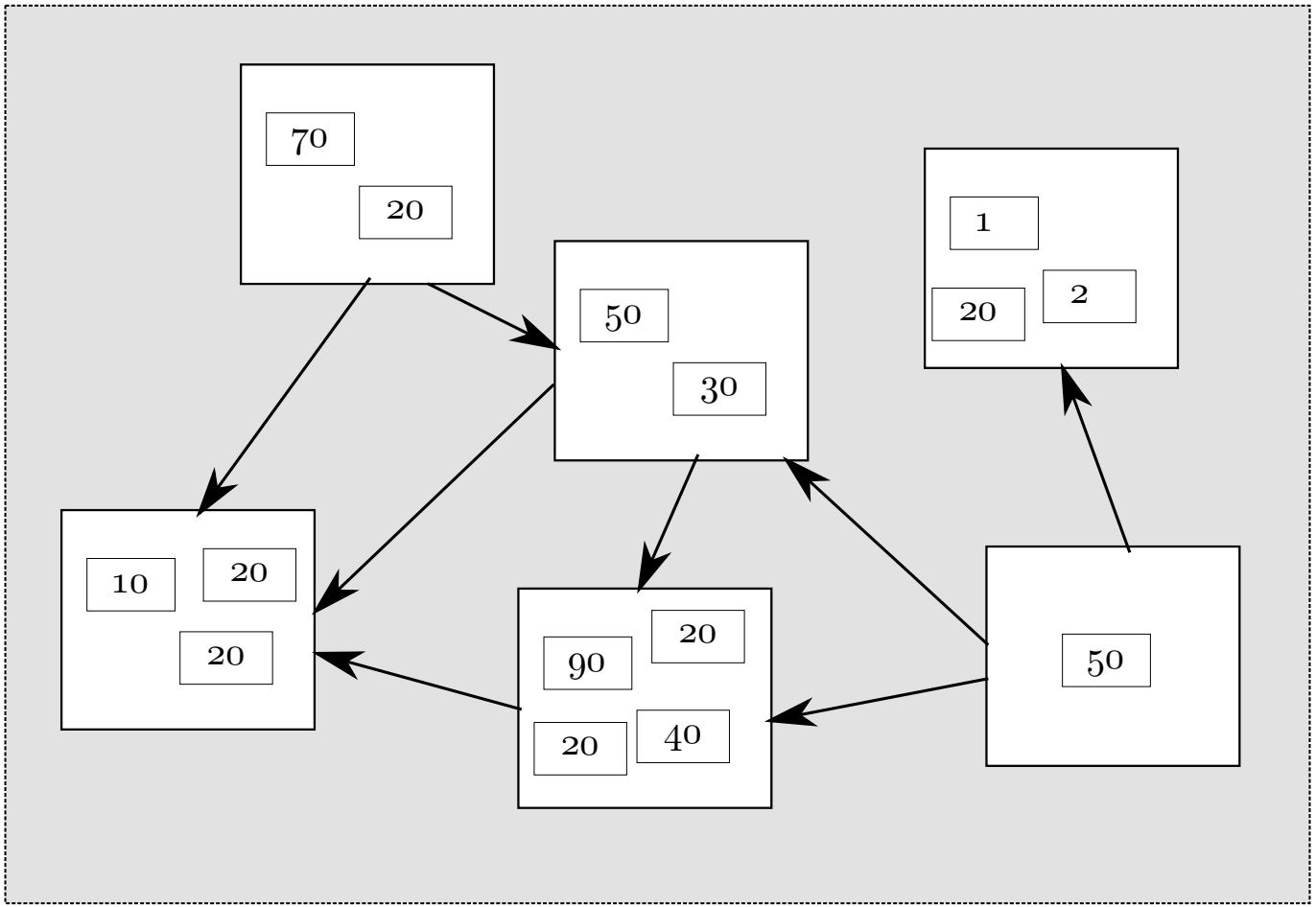
```
printf("%d %d", p1->x, p1->y);
```

Il n'y a aucun comportement ou service associé aux structures

Les structures du C



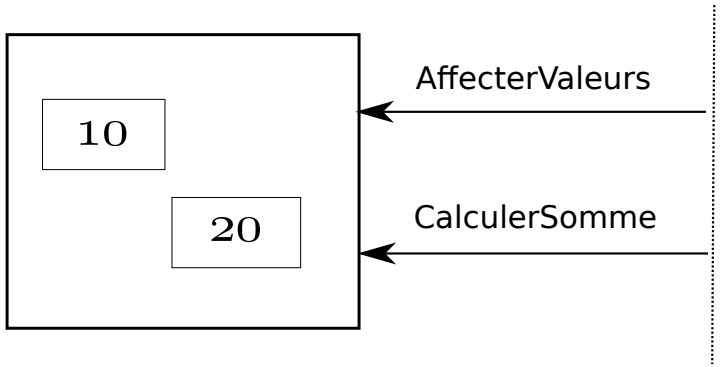
Programmation Orientée Objet (POO)



Programmation Orientée Objet (POO)

Un **objet** :

- rend un ensemble de services (*interface* de l'objet)
- contient des données (*état* de l'objet)



Une **classe** est un "moule" pour fabriquer des objets. Elle :

- définit les méthodes (c'est-à-dire les services)
- décrit la structure des données

Un objet créé à partir d'une classe A est une **instance** de la classe A.

Définir une classe

```
class Additionneur {

 int v1, v2;

 void affecterValeurs(int valeur1, int valeur2) {
  v1 = valeur1;
  v2 = valeur2;
 }

 int calculerSomme() {
  return v1+v2;
 }
}
```

Créer une instance

Création d'une instance avec le **mot-clé** `new` :

```
new Additionneur();
```

Cette expression renvoie une **référence** vers l'**instance** créée par `new`. Elle peut être affectée à une variable de type *"référence vers un Additionneur"*.

Déclaration d'une variable de type *"référence vers un Additionneur"* :

```
Additionneur add;
```

Affectation de la référence à la variable :

```
add = new Additionneur();
```

Utilisation des méthodes et accès aux données

On accède aux données et aux méthodes avec l'opérateur "." (point) :

```
class ProgrammePrincipal {
  public static void main(String arg[]) {
    Additionneur add = new Additionneur();
    add.affecterValeurs(10,30);
    int r = add.calculerSomme();
    System.out.println(r); // affiche 40.
    System.out.println(add.v1); // affiche 10.
    System.out.println(add.v2); // affiche 30.
  }
}
```

En Java, le compilateur vérifie que le membre (méthode ou attribut) existe en utilisant le type de la référence.

Référence “null”

Les variables de type référence contienne la valeur `null` par défaut. La valeur `null` signifie qu'elle ne pointe vers aucune instance.

Affectation de la valeur “null” à une variable :

```
Additionneur add = null;
```

L'utilisation d'une méthode (ou d'une donnée) à partir d'une variable de type référence à null provoque une erreur à l'**exécution** :

```
Additionneur add = null;
add.affecterValeurs(10,30);
```

```
Exception in thread "main" java.lang.NullPointerException
at ProgrammePrincipal.main(ProgrammePrincipal.java:4)
```

Comparaison de références

Il est possible de comparer deux *références* :

```
class ProgrammePrincipal {
  public static void main(String arg[]) {
    Additionneur add = new Additionneur();
    Additionneur add2 = add;

    if (add==add2) System.out.println("add==add2");
    else System.out.println("add!=add2");

    Additionneur add3 = new Additionneur();

    if (add==add3) System.out.println("add==add3");
    else System.out.println("add!=add3");
  }
}
```

Destruction des instances

En Java, la destruction des instances est assurée par le Garbage collector. Lorsqu'une instance n'est plus accessible à partir des variables, elle peut être détruite automatiquement par Java.

```
class ProgrammePrincipal {
  public static void main(String arg[]) {
    Additionneur add = new Additionneur();
    add.affecterValeurs(10,30);
    int r = add.calculerSomme();
    System.out.println(r);
    add = null; // L'instance créée par le new peut être détruite
    ....
  }
}
```

Objectif : une référence contient "null" ou désigne un objet

Destruction des instances (Autre exemple)

```
class Lien { Lien l; }

class ProgrammePrincipal {

 public static Lien creerCycle() {
  Lien lien1 = new Lien();
  Lien lien2 = new Lien();
  Lien lien3 = new Lien();
  lien1.l = lien2; lien2.l = lien3; lien3.l = lien1;
  return lien1;
 }

 public static void main(String arg[]) {
  Lien lien = creerCycle(); lien = null;
  ...   //Toutes les instances qui composent le cycle ne sont plus accessibles.
 }

}
```

Constructeur (Déclaration)

```
class Additionneur {

 int v1, v2;

 Additionneur(int valeur1, int valeur2) {
   v1 = valeur1; v2 = valeur2;
 }

 Additionneur(int valeur) {
   v1 = valeur; v2 = valeur;
 }

 int calculerSomme() {
  return v1+v2;
 }
}
```

Constructeur par défaut

Si aucun constructeur n'est défini, la classe a un **constructeur par défaut**

```
public class A {
        int a = 1;
        int b = 2;
}
```

est équivalent à

```
public class A {
        int a;
        int b;

        public A() {
                a = 1;
                a = 2;
        }
}
```

Conseil : ne pas initialiser les champs en dehors du constructeur

Constructeur (Utilisation)

```
class ProgrammePrincipal {

 public static void main(String arg[]) {
  Additionneur add1 = new Additionneur(10,20);
  Additionneur add2 = new Additionneur(20);
  int r1 = add1.calculerSomme();
  int r2 = add2.calculerSomme();
  System.out.println(r1); // affiche 30
  System.out.println(r2); // affiche 40
 }
}
```

Mot-clé **this**

```
class Additionneur {

 int valeur1, valeur2;

 Additionneur(int valeur1, int valeur2) {
   this.valeur1 = valeur1; this.valeur2 = valeur2;
 }

 Additionneur(int valeur) {
   valeur1 = valeur; valeur2 = valeur;
 }

 int calculerSomme() {
  return valeur1+valeur2;
 }
}
```

Mot-clé **this**

```
class Additionneur {

 int valeur1, valeur2;

 Additionneur(int valeur1, int valeur2) {
   this.valeur1 = valeur1; this.valeur2 = valeur2;
 }

 Additionneur(int valeur) {
   this(valeur, valeur);
 }

 int calculerSomme() {
  return valeur1+valeur2;
 }
}
```

Données et méthodes statiques

Les méthodes et des données **statiques** sont directement associées à la classe (et non aux instances de la classe) :

```
class Additionneur {

 static int c;
 int v1, v2;

 Additionneur(int valeur1, int valeur2) {
  v1 = valeur1; v2 = valeur2;
 }

 public static void setConstant(int constant) {
  c = constant;
 }

 public int calculerSomme() {
  return v1+v2+c;
 }
}
```

Données et méthodes statiques

Comme ces données et méthodes sont directement associées à la classe, il n'est pas nécessaire d'instancier la classe pour les utiliser :

```
class ProgrammePrincipal {

 public static void main(String[] arg) {
  Additionneur.setConstant(1);
  Additionneur add1 = new Additionneur(10,10);
  Additionneur add2 = new Additionneur(20,20);
  System.out.println(add1.calculerSomme()); // affiche 21
  System.out.println(add2.calculerSomme()); // affiche 41
  Additionneur.c = 2;
  System.out.println(add1.calculerSomme()); // affiche 22
  System.out.println(add2.calculerSomme()); // affiche 42
 }

}
```

Données et méthodes statiques

Une méthode statique ne peut utiliser que :

- des données statiques à la classe ;
- des méthodes statiques à la classe.

afin de garantir (par transitivité) l'utilisation exclusive de données statiques.

L'utilisation de `this` n'a aucun sens dans une méthode statique :

```
class Additionneur {

  static int c;
  ...
  public static void setConstant(int c) {
    Additionneur.c = c;
  }
}
```

Les tableaux

Déclaration d'une variable de type "référence vers un tableau de" :

```
int[] tableauDEntiers;
Additionneur[] tableauDAdds;
```

Création d'un tableau :

```
tableauDEntiers = new int[10];
tableauDAdds = new Additionneur[10];
```

Utilisation d'un tableau :

```
for (int i = 0; i < tableauDAdds.length; i++)
        tableauDAdds[i] = new Additionneur(i,10-i);
int r = 1;
for (int i = tableauDAdds.length-1; i >= 0; i--)
        r = r*tableauDAdds[i].calculerSomme();
```

Les tableaux à plusieurs indices

Déclaration :

```
int[][] matriceDEntiers;
Additionneur[][] matriceDAdds;
```

Création :

```
matriceDAdds = new Additionneur[10][];
for (int i = 0; i < matriceDAdds.length; i++)
  matriceDAdds[i] = new Additionneur[5];
```

ou

```
matriceDAdds = new Additionneur[10][5];
```

Tableaux “non rectangulaires” :

```
matriceDAdds = new Additionneur[10][];
for (int i = 0; i < matriceDAdds.length; i++)
  matriceDAdds[i] = new Additionneur[i+1];
```

Les exceptions et les tableaux

```
int[] tab = new int[12];
for (int i = 0; i < tab.length; i++) tab[i] = i;
System.out.println(tab[12]);
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 12
at ProgrammePrincipal.main(ProgrammePrincipal.java:5)
```

```
int[] tab = null;
System.out.println(tab[12]);
```

```
Exception in thread "main" java.lang.NullPointerException
at ProgrammePrincipal.main(ProgrammePrincipal.java:4)
```

Chaînes de caractères

Trois classes permettent de gérer les chaînes de caractères :

- la classe **String** : chaîne invariable ;
- la classe **StringBuffer** : chaîne modifiable (multi-thread).
- la classe **StringBuilder** : chaîne modifiable (mono-thread).

Déclaration et création :

```
String h = "Hello";
String w = "World";
```

Concaténation :

```
String hw = h + " "+ w + " ! ";
int c = 13;
String hw12c = h + " "+ w + " " + 12 + " " + c;
```

La conversion est effectée en utilisant l'une des méthodes statiques `valueOf` de la classe `String`.

Chaînes de caractères

Affichage :

```
System.out.print(hw);          // affiche "Hello World ! "
System.out.println(hw12c);     // affiche "Hello World 12 13"
                               // avec retour à la ligne
```

Comparaison :

```
String a1 = "a";
String a2 = "a";
String a3 = new String("a");
System.out.println(a1==a2);          // affiche "true"
System.out.println(a1==a3);          // affiche "false"
System.out.println(a1.equals(a3));   // affiche "true"
```

Exemple : formulaire avec boutons

```
class ProgrammePrincipal {

 public static void main(String[] arg) {
  Formulaire f = new Formulaire(2);
  f.ajouterBouton(new Bouton("ouvrir", 10, 10, 20, 20));
  f.ajouterBouton(new Bouton("annuler", 30, 10, 40, 20));

  Bouton b;
  b = f.detecterClic(15, 15);
  if (b!=null) b.afficherTexte(); // affiche "ouvrir"
  b = f.detecterClic(35, 15);
  if (b!=null) b.afficherTexte(); // affiche "annuler"
  b = f.detecterClic(50, 15);
  if (b!=null) b.afficherTexte(); // n'affiche rien
 }

}
```

Exemple : formulaire avec boutons

```
class Formulaire {

 int nbBoutons;
 Bouton[] boutons;

 Formulaire(int maxNbBoutons) {
  boutons = new Bouton[maxNbBoutons]; nbBoutons = 0;
 }

 void ajouterBouton(Bouton bouton) {
  boutons[nbBoutons] = bouton; nbBoutons++;
 }

 Bouton detecterClic(int x, int y) {
  for (int i = 0; i < nbBoutons; ++i)
        if (boutons[i].detecterClic(x,y))
              return boutons[i];
  return null;
 }
}
```

Exemple : formulaire avec boutons

```
class Bouton {

 String texte;
 int x1, y1, x2, y2;

 Bouton(String texte, int x1, int y1, int x2, int y2) {
   this.texte = texte;
   this.x1 = x1; this.y1 = y1;
   this.x2 = x2; this.y2 = y2;
 }

 boolean detecterClic(int x, int y) {
   return x>=x1 && x<=x2 && y>=y1 && y <= y2;
 }

 void afficherTexte() {
   System.out.println(texte);
 }

}
```

Packages

- Un **package** est un ensemble de classes.
- Deux classes peuvent avoir le même nom si elles appartiennent à deux paquets différents.
- Une ligne "**package** `nomPaquet` ;" (placée **au début** du fichier source) permet de placer les classes décrites dans le fichier dans le paquet `nomPaquet`.
- Par défaut, les classes sont placées dans le paquet sans nom.
- Un nom de paquet est une suite d'identificateurs séparés de points.
  Ex : `monPaquet`, `mon.Paquet`, `java.awt`
- Les noms de paquet qui commencent par `java.` sont réservés à Java.

Noms des classes et mot-clé **import**

- Une classe peut être désignée en préfixant le nom de la classe par celui du paquet (`java.awt.List` = classe `List` du paquet `java.awt`).
- Une classe `C` du paquet `nomPaquet` peut être désignée par son nom :
  - depuis les classes du paquet nomPaquet
  - depuis les fichiers qui contiennent "`import nomPaquet.C`"
  - depuis les fichiers qui contiennent "`import nomPaquet.*`"
- Les classes du paquet `java.lang` sont toujours importées : ce paquet contient les classes fondamentales de Java (`Object`, `String`...).

Packages et modificateur **public**

- Par défaut, une classe ou une méthode est **non-publique** : elle n'est accessible que depuis les classes du **même** paquet.
- Une classe ou un membre public est accessible de n'importe où.
- Pour rendre une classe ou un membre **public** :

```
public class ClassePublique {
  public int proprietePublique;
  public void methodePublique() { }
}
```

- Si un fichier contient une classe publique, le nom du fichier doit être formé du nom de la classe suivi de ".java".
- Un fichier ne peut contenir qu'une seule classe publique.

Modificateur **private** et encapsulation

- Un membre **privé** n'est accessible que par les méthodes de la classe qui le contient.
- Pour rendre un membre privé, on utilise le modificateur **private** :

```
public class ClassePublique {
  private int proprietePrivee;
  private void methodePrivee() { }
}
```

- Encapsulation : Tout ce qui participe à l'implémentation des services doit être privé (afin de permettre la modification de l'implémentation des services sans risquer d'impacter les autres classes)

Point d'entrée d'un programme

Le programme *HelloWorld.java* :

```
public class HelloWorld {
   public static void main(String arg[]) {
     System.out.println("Hello world ! ");
     for (int i = 0; i < arg.length; i++)
      System.out.println(arg[i]);
   }
}
```

```
$ javac HelloWorld.java
$ ls
HelloWorld.java HelloWorld.class
$ java HelloWorld toto aaa
Hello world !
toto
aaa
```

Convention de nommage (respectée par Java)

- Première lettre en majuscule pour les noms des classes
- Première lettre en minuscule pour les noms des membres
- Ensuite, première lettre de chaque mot en majuscule
- Noms simples et descriptifs
- N'utiliser que des lettres et des chiffres

```
class MaPile {

        private int maPile[] = new int[100];
        private int taille = 0;

        public void empilerEntier(int entier) {
            maPile[taille] = entier; taille++;
        }

        public int depilerEntier() {
            taille--; return maPile[taille];
        }
}
```

Résumé du cours

Les points abordés pendant ce cours :

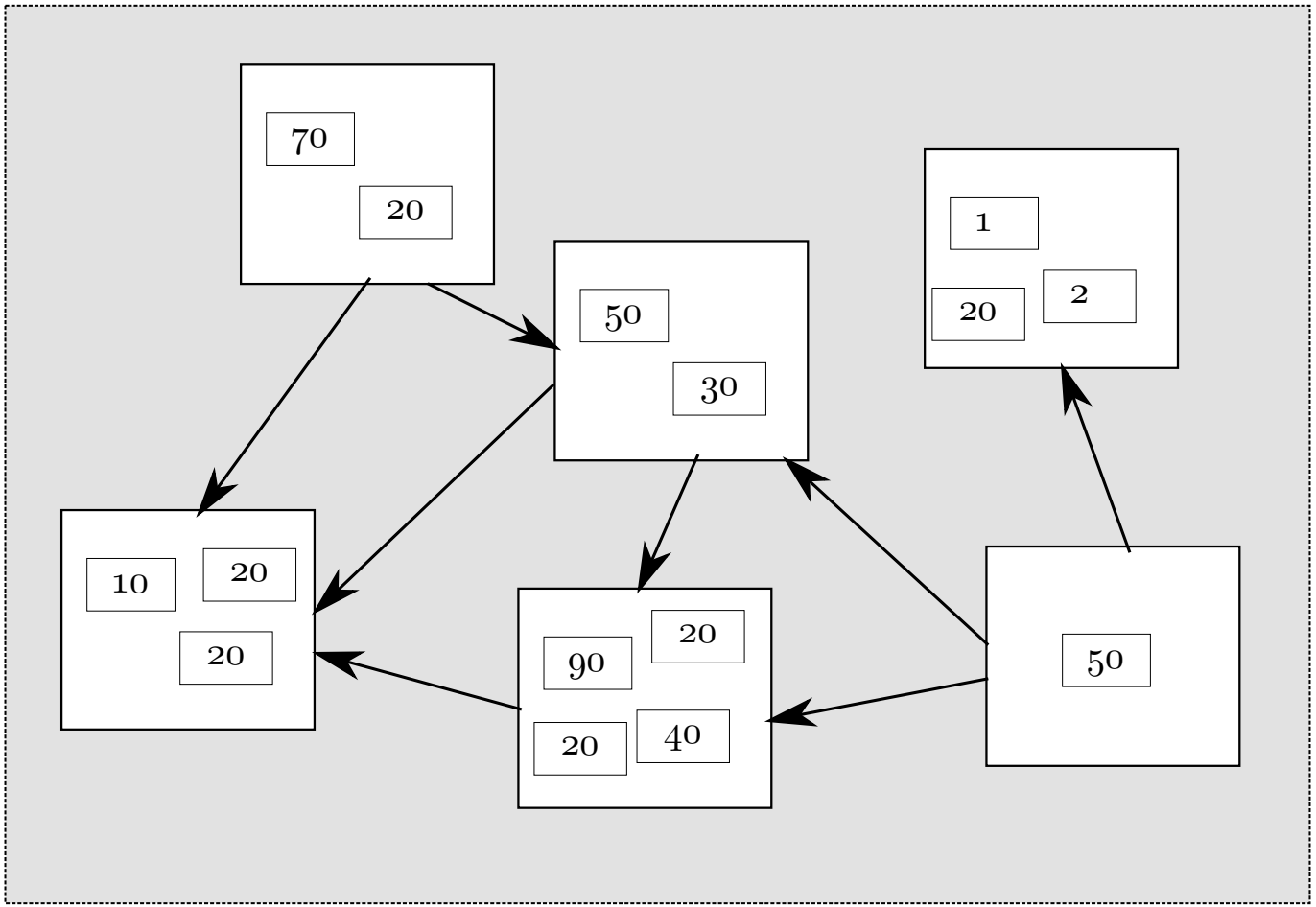
- Objets, classes et instances
- Variables de type “référence”
- Définir une classe
- Instancier une classe
- Définir et utiliser des constructeurs
- Destruction des instances (Garbage collector)
- Mot-clé **this**
- Membres statiques
- Les tableaux
- Les chaînes de caractères
- Organisation en packages
- Convention de nommage

Résumé du cours précédent

Les points abordés pendant ce cours :

- Objets, classes et instances
- Variables de type “référence”
- Définir une classe
- Instancier une classe
- Définir et utiliser des constructeurs
- Destruction des instances (Garbage collector)
- Mot-clé **this**
- Membres statiques
- Les tableaux
- Les chaînes de caractères
- Organisation en packages
- Convention de nommage

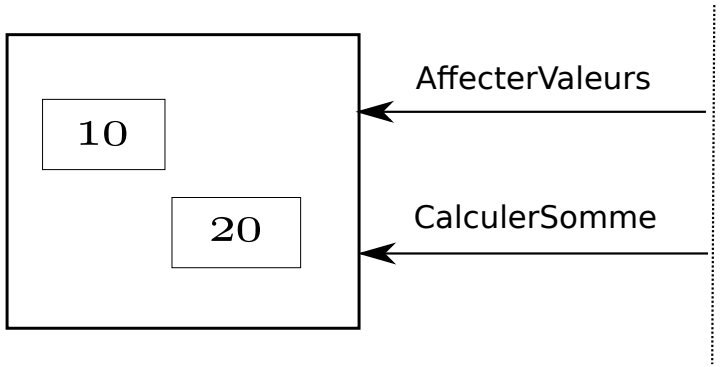
Rappel : Programmation Orientée Objet (POO)



Rappel : Programmation Orientée Objet (POO)

Un **objet** :

- rend un ensemble de services (*interface* de l'objet)
- contient des données (*état* de l'objet)



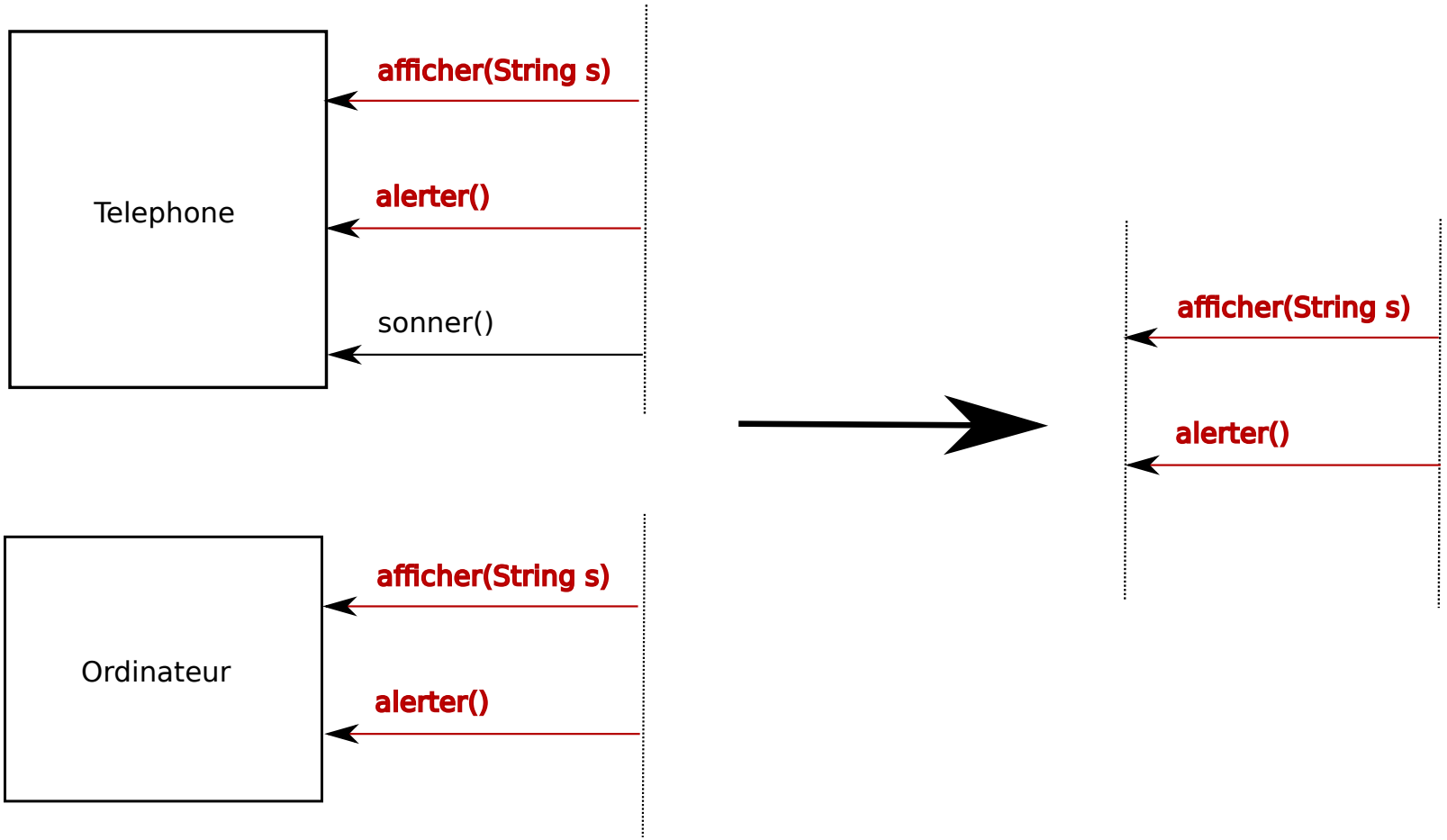
Une **classe** est un "moule" pour fabriquer des objets. Elle :

- définit les méthodes (c'est-à-dire les services)
- décrit la structure des données

Un objet créé à partir d'une classe A est une **instance** de la classe A.

Abstraction

Des objets implémentant des services identiques de façons différentes :



Les objets implémentent les méthodes **afficher(String s)** *et* **alerter()**

Abstraction

Objectifs :

- Traiter des objets *en utilisant les services qu'ils partagent* :

```
for (int i = 0; i < afficheurs.length; i++)
  afficheurs[i].afficher(s);
}
```

- Écrire un programme en supposant que les objets manipulés implémentent certains services :

```
boolean estOrdonne(Comparable tab[]) {
  for (int i = 0; i < tab.length - 1; i++)
    if (tab[i].compareTo(tab[i+1]) > 0) return false
  return true;
}
```

Si le tableau contient des objets "comparables" alors cette méthode détermine si le tableau est ordonné

Description d'une interface

Description d'une Interface en Java :

```
public interface Afficheur {
    /**
     * Affiche la chaine de caracteres s.
     * @param s la chaine a afficher.
     */
    public void afficher(String s);

    /**
     * Alerte l'utilisateur.
     */
    public void alerter();
}
```

Une interface peut être vue comme un **contrat** entre celui qui utilise les objets et celui qui les implémente.

Implémentation d'une interface

```
class Telephone implements Afficheur {

 public void afficher(String s) {
  System.out.println(s);
 }

 void alerter() {
  afficher("Alerte !!!");
 }

 void sonner() {
  System.out.println("bip !!");
 }
}
```

Implémentation d'une interface

```
class Ordinateur implements Afficheur {

 String nom;

 public Ordinateur(String nom) {
  this.nom = nom;
 }

 public void afficher(String s) {
  System.out.println(nom+" : "+s);
 }

 void alerter() {
  System.out.println("Alerte de "+nom);
 }
```

Références et interfaces

Déclaration d'une variable de type référence vers "**une instance d'une classe qui implémente l'interface Afficheur**" :

```
Afficheur aff;
```

Il n'est pas possible d'instancier une interface :

```
Afficheur aff = new Afficheur(); // interdit !
aff.afficher("toto"); // que faire ?
```

Il est possible d'instancier une classe qui implémente une interface :

```
Telephone t = new Telephone();
```

et de mettre la référence dans une variable de type "Afficheur" :

```
Afficheur aff1 = t;
Afficheur aff2 = new Ordinateur("truc");
Afficheur aff3 = new Integer(2) // impossible ! !
```

Transtypage vers le haut (upcasting)

Références et interfaces

Utilisation d'objets qui implémentent l'interface :

```
class ProgrammePrincipal {

 void afficherChaine(Afficheur[] afficheurs, String s) {
  for (unsigned int i = 0; i < afficheurs.length; i++)
        afficheurs[i].afficher(s);
 }

 void afficherTableau(String[] tab, Afficheur a) {
  for (unsigned int i = 0; i < tab.length; i++)
        a.afficher(tab[i]);
 }
}
```

Vérification à la compilation :

```
Afficheur aff = new Telephone();
aff.sonner(); // impossible (ne fait pas partie de l'interface)
```

Polymorphisme

Le choix de la méthode à exécuter ne peut être fait qu'à l'exécution :

```
Afficheur[] afficheurs = new Afficheur[2];
afficheurs[0] = new Telephone();
afficheurs[1] = new Ordinateur("pc");
Random r = new Random(); // instanciation d'un générateur aléatoire.
int i = r.nextInt(2); // tirage d'un nombre en 0 et 1.
afficheurs[i].afficher("mon message");
```

- Si `i=0`, le programme affiche `mon message`.
- Si `i=1`, le programme affiche `pc : mon message`.

(En C, l'**éditeur de liens** détermine à la compilation les appels à effectuer)

Résumé

- Une **interface** est un ensemble de signatures de méthodes.
- Une classe peut **implémenter** une interface : elle **doit** préciser le comportement de chacune des méthodes de l'interface.
- Il est possible de déclarer une variable pouvant contenir des références vers des instances de classes **qui implémentent l'interface**.
- Java vérifie **à la compilation** que toutes les affectations et les appels de méthodes sont corrects.
- Le choix du code qui va être exécuté est décidé **à l'exécution** (en fonction de l'instance pointée par la référence).

Implémentations multiples

```
interface Visualisable {
        /**
        * Affiche une chaîne de caractères qui represente l'objet.
        */
        public void visualiser();
}

interface Pile {
        /**
        * Empile la valeur v sur la pile.
        * @param v la valeur à empiler.
        */
        public void empiler(int v);

        /**
        * Dépile une valeur sur la pile.
        * @return la valeur dépilée.
        */
        public int depiler();
}
```

Implémentations multiples

Implémentation des deux interfaces précédentes :

```
class PileVisualisable implements Visualisable, Pile {

        int[] tab; int size;

        public PileVisualisable(int capacite) {
                tab = new int[capacite]; size = 0;
        }

        public void empiler(int v) { tab[size] = v; size++; }

        public int depiler() { size--; return tab[size]; }

        public void visualiser() {
                for (int i = 0; i < size; i++)
                        System.out.print(tab[i]+" ");
                System.out.println();
        }
}
```

Implémentations multiples

Implémentation d'une des deux interfaces :

```
class StringVisualisable implements Visualisable {

        String s;

        public StringVisualisable(String s) {
                this.s = s;
        }

        public void visualisable() {
                System.out.println(s);
        }

}
```

Implémentations multiples

Exemple :

```
Visualisable[] tab = new Visualisable[3];
tab[0] = new StringVisualisable("bonjour");
PileVisualisable p = new PileVisualisable(10);
tab[1] = p;
tab[2] = new StringVisualisable("salut");
p.empiler(10);
p.empiler(30);
System.out.println(p.depiler());
p.empiler(12);
for (int i = 0; i < tab.length; i++)
        tab[i].visualiser();
```

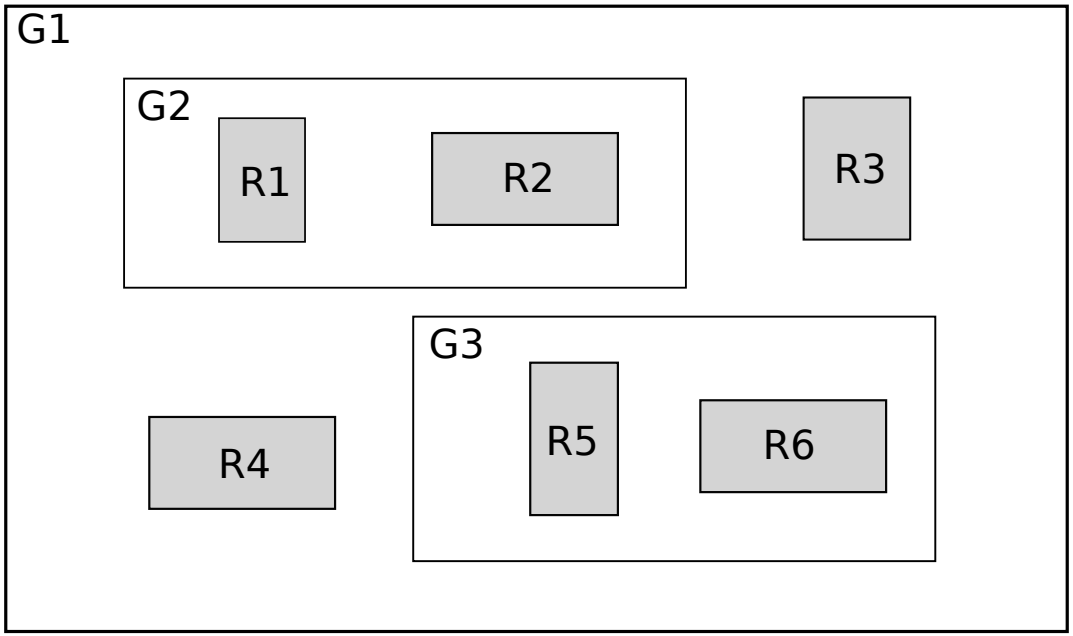
Qu'affiche ce programme à l'écran ?

Implémentations multiples

Vérification des types à la compilation :

```
Visualisable[] tab = new Visualisable[3];
tab[0] = new StringVisualisable("bonjour");
Pile p = new PileVisualisable(10);
tab[1] = p; Erreur à la compilation (Une pile n'est pas visualisable) !
tab[2] = new StringVisualisable("salut");
p.empiler(10);
p.empiler(30);
System.out.println(p.depiler());
p.empiler(12);
for (int i = 0; i < tab.length; i++)
        tab[i].visualiser();
```

Exemple : Formes cliquables



- Les formes sont regroupées de façon hiérarchique.
- Seuls les formes grises captent les clics.
- Lorsque l'utilisateur clique sur une forme grise, le programme doit afficher les noms des formes traversés par le clic.
  Exemple : si on clique dans le rectangle `R5`, on traverse `R5 G3 G1`.

Exemple : Formes cliquables

Toutes les formes vont implémenter l'interface suivante :

```
interface Cliquable {

        /**
         * Propage les conséquences d'un clic
         * @param x coordonnée x du clic
         * @param y coordonnée y du clic
         * @return true si le point touche un point gris de la forme,
                   false sinon
         */
        public boolean traiterClic(int x, int y);

}
```

Exemple : Formes cliquables

```
class Rectangle implements Cliquable {

 int x1, y1, x2, y2, numero;

 Rectangle(int x1, int y1, int x2, int y2, int numero) {
        this.x1 = x1; this.x2 = x2;
        this.y1 = y1; this.y2 = y2;
        this.numero = numero;
 }

 public boolean traiterClic(int x, int y) {
  if (x>=x1 && x<=x2 && y>=y1 && y<=y2) {
        System.out.println("R"+numero);
        return true;
  }
  return false;
 }

}
```

Exemple : Formes cliquables

```
class Groupe implements Cliquable {

 int numero, size;
 Cliquable[] elements;

 Groupe(int capacite, int numero) {
        elements = new Cliquable[capacite]; size = 0;
        this.numero = numero;
 }

 void ajouterElement(Cliquable c) {elements[size]=c; size++;}

 public boolean traiterClic(int x, int y) {
  boolean res = false;
  for (int i = 0; i < size; i++)
        if (elements[i].traiterClic(x,y)) res = true;
  if (res) System.out.println("G"+numero);
  return res;
 }
}
```

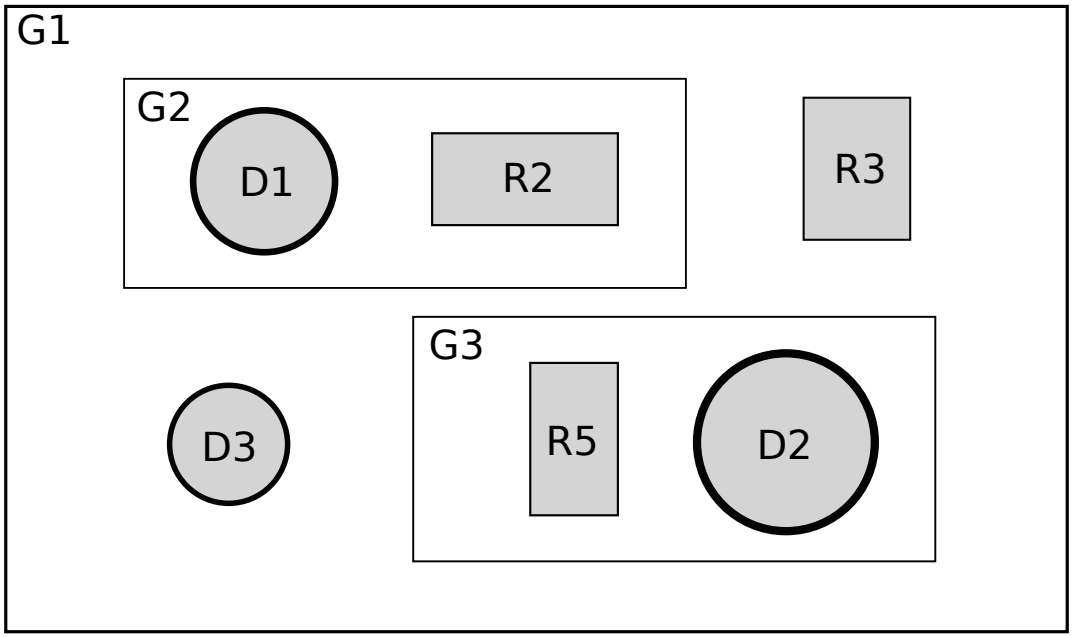
Exemple : Formes cliquables

Exemple d'utilisation :

```
Groupe g1 = new Groupe(4,1);
Groupe g2 = new Groupe(2,2);
Groupe g3 = new Groupe(2,3);
g1.ajouterElement(g2);
g1.ajouterElement(g3);
g2.ajouterElement(new Rectangle(10,10,40,40,1));
g2.ajouterElement(new Rectangle(50,20,70,30,2));
g1.ajouterElement(new Rectangle(80,10,90,30,3));
g1.ajouterElement(new Rectangle(20,60,50,80,4));
g3.ajouterElement(new Rectangle(60,40,70,80,5));
g3.ajouterElement(new Rectangle(80,50,100,70,6));

g1.traiterClic(65,60);
```

Qu'affiche ce programme ?

Exemple : Formes cliquables – Ajout des disques



Que faut-il modifier pour gérer les disques ?

Exemple : Formes cliquables – Ajout des disques

```
public class Disque implements Cliquable {

    int x, y, r, numero;

    public Disque(int x, int y, int r, int numero) {
        this.x = x; this.y = y; this.r = r;
        this.numero = numero;
    }

    public boolean traiterClic(int x, int y) {
        int dx = x - this.x, dy = y - this.y;
        if (Math.sqrt(dx*dx+dy*dy)<=r) {
            System.out.println("D"+numero);
            return true;
        }
        return false;
    }

}
```

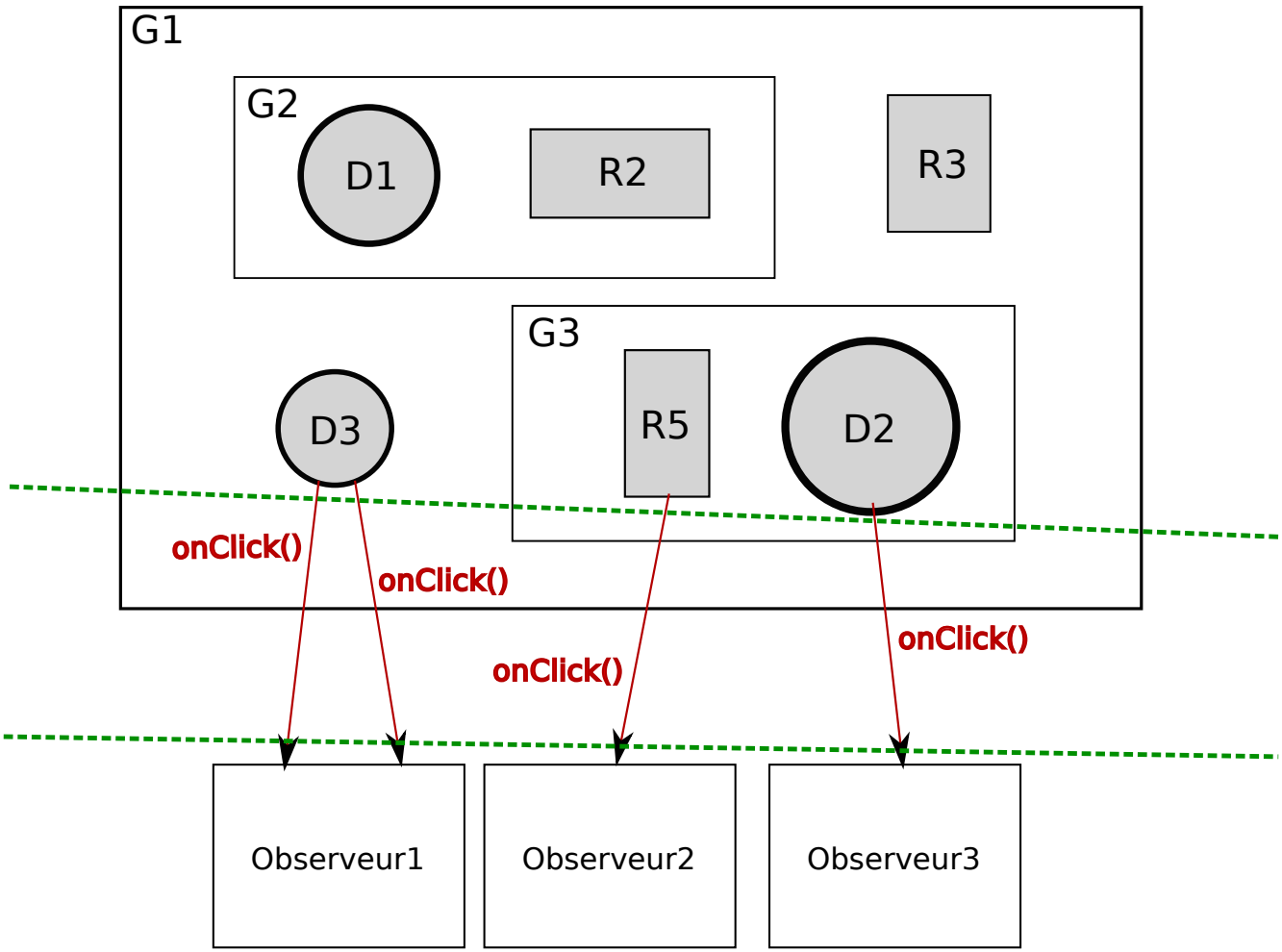
Exemple : Formes cliquables – Ajout des disques

Exemple d'utilisation :

```
Groupe g1 = new Groupe(4, 1);
Groupe g2 = new Groupe(2, 2);
Groupe g3 = new Groupe(2, 3);
g1.ajouterElement(g2);
g1.ajouterElement(g3);
g2.ajouterElement(new Disque(25, 25, 25, 1));
g2.ajouterElement(new Rectangle(50, 20, 70, 30, 2));
g1.ajouterElement(new Rectangle(80, 10, 90, 30, 3));
g1.ajouterElement(new Disque(35, 70, 20, 3));
g3.ajouterElement(new Rectangle(60, 40, 70, 80, 5));
g3.ajouterElement(new Disque(90, 60, 20, 2));

g1.traiterClic(92, 65);
```

Qu'affiche ce programme ?

Exemple : Formes cliquables – Observateurs



Exemple : Formes cliquables – Observateurs

Interface pour les échanges entre les formes et les observateurs :

```
public interface ClickListener {

    /**
     * Invoquée quand l'objet est cliqué.
     */
    public void onClick(Cliquable c);

}
```

Exemple : Formes cliquables – Observateurs

```
class Rectangle implements Cliquable {
 ....
 ClickListener[] listeners = new ClickListener[100];
 int nbListeners = 0;
 ...
 void addClickListener(ClickListener l) {
     listeners[nbListeners] = l; nbListeners++;
 }

 public boolean traiterClic(int x, int y) {
  if (x>=x1 && x<=x2 && y>=y1 && y<=y2) {
        for (int i = 0; i < nbListeners; i++)
              listeners[i].onClick(this);
        return true;
  }
  return false;
 }
}
```

Exemple : Formes cliquables – Observateurs

```
class Disque implements Cliquable {
 ....
 ClickListener[] listeners = new ClickListener[100];
 int nbListeners = 0;
 ...
 void addClickListener(ClickListener l) {
     listeners[nbListeners] = l; nbListeners++;
 }

 public boolean traiterClic(int x, int y) {
     int dx = x - this.x, dy = y - this.y;
     if (Math.sqrt(dx * dx + dy * dy) <= r) {
         for (int i = 0; i < nbListeners; i++)
                 listeners[i].onClick(this);
         return true;
     }
     return false;
 }
}
```

Exemple : Formes cliquables – Observateurs

Première implémentation de l'interface ClickListener :

```
public class SimpleClickListener implements ClickListener {

    String msg;

    public SimpleClickListener(String msg) {
        this.msg = msg;
    }

    public void onClick(Cliquable c) {
        System.out.println("Clic : " + msg);
    }

}
```

Exemple : Formes cliquables – Observateurs

Exemple d'utilisation :

```
Groupe g1 = new Groupe(4, 1);
Rectangle r1 = new Rectangle(10,10, 30, 30, 1);
Disque d1 = new Disque(20,20, 5, 1);
g1.ajouterElement(r1);
g1.ajouterElement(d1);
r1.addClickListener(new SimpleClickListener("Rectangle1"));
d1.addClickListener(new SimpleClickListener("Disque1"));
SimpleClickListener scl = new SimpleClickListener("cool !!");
r1.addClickListener(scl);
d1.addClickListener(scl);

g1.traiterClic(20, 20);

g1.traiterClic(10, 10);
```

Qu'affiche ce programme ?

Exemple : Formes cliquables – Observateurs

Classe anonyme :

```
Rectangle r = new Rectangle(10, 10, 40, 40, 1);
ClickListener cl = new ClickListener() {
    public void onClick(Cliquable c) {
        System.out.println("super !!");
    }
};
r.addClickListener(cl);
r.traiterClic(20, 20);
```

Sans variable intermédiaire (moins lisible) :

```
Rectangle r = new Rectangle(10, 10, 40, 40, 1);
r.addClickListener(new ClickListener() {
    public void onClick(Cliquable c) {
        System.out.println("super !!");
    }
});
r.traiterClic(20, 20);
```

Exemple : Formes cliquables – Observateurs

Transtypage vers le bas (downcasting) :

```
public class RectangleClickListener implements ClickListener {

    public void onClick(Cliquable c) {
        Rectangle r = (Rectangle)c;
        System.out.println("Rectangle : " + r.numero);
    }

}
```

Exemple :

```
Rectangle r = new Rectangle(10, 10, 40, 40, 6);
r.addClickListener(new RectangleClickListener());
r.traiterClic(20, 20);
```

Qu'affiche ce programme ?

Exemple : Formes cliquables – Observateurs

Transtypage vers le bas (downcasting) :

```
public class RectangleClickListener implements ClickListener {

    public void onClick(Cliquable c) {
        Rectangle r = (Rectangle)c;
        System.out.println("Rectangle : " + r.numero);
    }

}

Disque d = new Disque(20, 20, 20, 4);
d.addClickListener(new RectangleClickListener());
d.traiterClic(20, 20);
```

Durant l'exécution :

```
Exception in thread "main" java.lang.ClassCastException: Disque cannot be cast to Rectangle
        at RectangleClickListener.onClick(RectangleClickListener.java:5)
        at Disque.traiterClic(Disque.java:23)
        at Test.main(Test.java:26)
Java Result: 1
```

Résumé

- Description d'une interface
- Implémentation d'une interface
- Référence vers une classe implémentant une interface
- Implémentation de plusieurs interfaces
- Polymorphisme d'objet
- Classe anonyme
- Transtypage (vers le haut et vers le bas)

Résumé des cours précédents

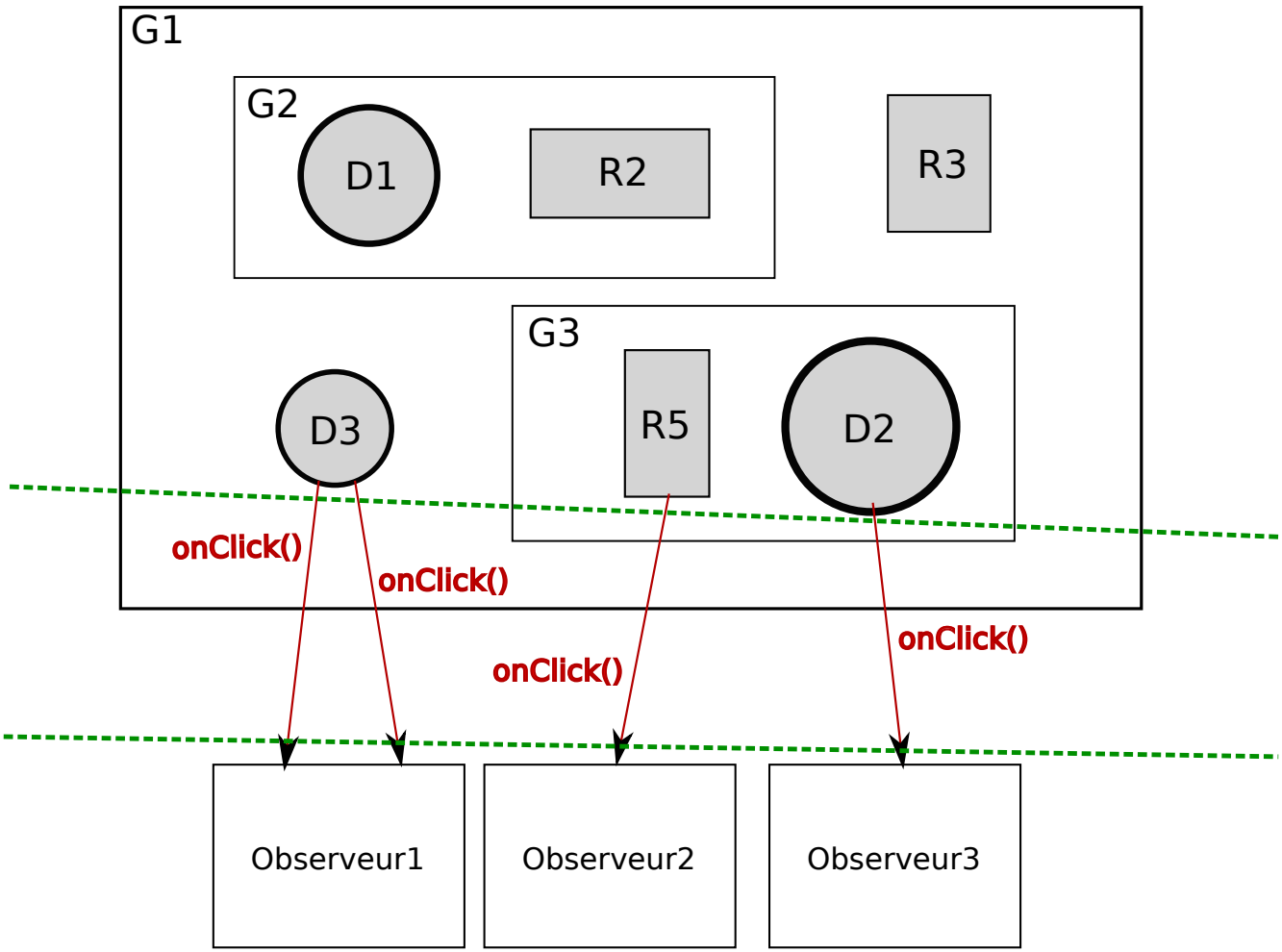
Premier cours :

- Objets, classes et instances
- Variables de type “référence”
- Définir et instancier une classe
- Tableaux, chaînes de caractères
- Packages

Deuxième cours :

- Description et implémentation d'interfaces
- Polymorphisme d'objet
- Transtypage (vers le haut et vers le bas)

Rappel : Interfaces et formes cliquables



Rappel : Interfaces et formes cliquables

Interface pour les échanges entre les formes et les observateurs :

```
public interface ClickListener {

    /**
     * Invoquée quand l'objet est cliqué.
     */
    public void onClick(Cliquable c);

}
```

Rappel : Interfaces et formes cliquables

```
class Rectangle implements Cliquable {
 ....
 ClickListener[] listeners = new ClickListener[100];
 int nbListeners = 0;
 ...
 void addClickListener(ClickListener l) {
     listeners[nbListeners] = l; nbListeners++;
 }

 public boolean traiterClic(int x, int y) {
  if (x>=x1 && x<=x2 && y>=y1 && y<=y2) {
        for (int i = 0; i < nbListeners; i++)
              listeners[i].onClick(this);
        return true;
  }
  return false;
 }
}
```

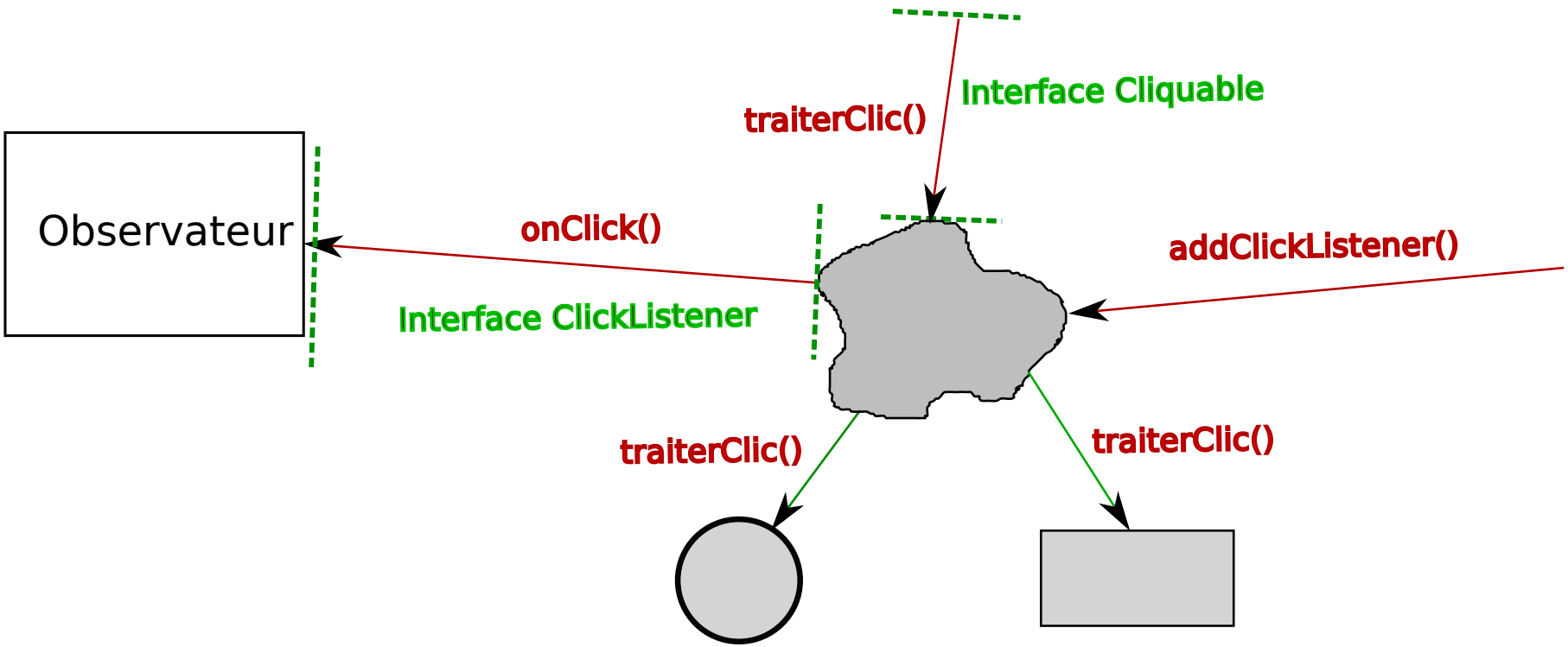
Rappel : Interfaces et formes cliquables

```
class Disque implements Cliquable {
 ....
 ClickListener[] listeners = new ClickListener[100];
 int nbListeners = 0;
 ...
 void addClickListener(ClickListener l) {
     listeners[nbListeners] = l; nbListeners++;
 }

 public boolean traiterClic(int x, int y) {
     int dx = x - this.x, dy = y - this.y;
     if (Math.sqrt(dx * dx + dy * dy) <= r) {
         for (int i = 0; i < nbListeners; i++)
                 listeners[i].onClick(this);
         return true;
     }
     return false;
 }
}
```

Abstraction et extension

Le service associé aux listeners est identique. On aimerait :

- définir une classe FormeCliquable qui implémente ce service ;
- sans perdre les spécificités d'un disque ou d'un rectangle.



Extension

L'extension permet de créer une classe qui :

- conserve les services (propriétés et méthodes) d'une autre classe ;
- ajouter ses propres services (propriétés et méthodes).

En Java :

- On utilise le mot-clé **extends** pour étendre une classe ;
- Une classe ne peut étendre qu'une seule classe.

Ne pas étendre quand une implémentation d'interface suffit !

La classe FormeCliquable

```
public class FormeCliquable implements Cliquable {

    private ClickListener[] listeners;
    private int nbListeners;

    public FormeCliquable() {
        listeners = new ClickListener[100];
        nbListeners = 0;
    }

    public void addClickListener(ClickListener l) {
     listeners[nbListeners] = l; nbListeners++;
    }

    public boolean traiterClic(int x, int y) {
        for (int i = 0; i < nbListeners; i++)
                listeners[i].onClick(this);
        return true;
    }
}
```

Utilisation de l'extension – La classe Rectangle

```
public class Rectangle extends FormeCliquable {

    public int x1, y1, x2, y2, numero;

    Rectangle(int x1, int y1, int x2, int y2, int numero) {
        this.x1 = x1; this.x2 = x2;
        this.y1 = y1; this.y2 = y2;
        this.numero = numero;
    }

    public boolean traiterClic(int x, int y) {
        if (x >= x1 && x <= x2 && y >= y1 && y <= y2) {
            super.traiterClic(x, y);
            return true;
        }
        return false;
    }
}
```

Utilisation de l'extension – La classe Disque

```
public class Disque extends FormeCliquable {

    public int x, y, r, numero;

    public Disque(int x, int y, int r, int numero) {
        this.x = x; this.y = y;
        this.r = r;
        this.numero = numero;
    }

    public boolean traiterClic(int x, int y) {
        int dx = x - this.x, dy = y - this.y;
        if (Math.sqrt(dx * dx + dy * dy) <= r) {
            return super.traiterClic(x,y);
        }
        return false;
    }

}
```

Transtypage vers le haut (upcasting)

L'upcasting est toujours possible :

Si la classe B étend la classe A, l'affectation d'une référence de type "B" dans une variable de type "A" est **toujours possible**.

(car tout ce qu'une instance de A sait faire, une instance de B sait le faire)

Exemple :

```
public class A {
        public void method1() { System.out.println("m1"); }
}

public class B extends A {
        public void method2() { System.out.println("m2"); }
}

B b = new B();
A a = b; //upcasting
a.method1(); // la méthode est implémenté
a.method2(); // erreur à la compilation : la classe A ne définit pas cette méthode
```

Transtypage vers le bas (downcasting)

L'upcasting n'est pas toujours possible :

Si la classe B étend la classe A, l'affectation d'une référence de type "A" dans une variable de type "B" **n'est pas toujours possible**.

Exemple :

```
public class A {
        public void method1() { System.out.println("m1"); }
}

public class B extends A {
        public void method2() { System.out.println("m2"); }
}

A a1 = new A();
B b1 = new B();
A a2 = b1; // upcasting !
B b2 = (B)a2; // downcasting possible !
B b3 = (B)a1; // downcasting impossible (erreur à l'exécution)
```

Redéfinition de méthodes et polymorphisme

Dans la classe Rectangle, nous avons redéfini la méthode `traiterClic` :

- La méthode existe dans la classe FormeCliquable ;
- Une nouvelle implémentation est donnée dans la classe Rectangle.

```
public class A {
        public void afficherNom() { System.out.println("A"); }
}

public class B extends A {
        public void afficherNom() { System.out.println("B"); }
}

B b = new B(); b.afficherNom(); // affiche B
A a = new A(); a.afficherNom(); // affiche A
a = b; a.afficherNom(); // affiche B
```

Mot-clé **super**

Le mot-clé **super** permet d'utiliser une méthode définie au dessus :

```
public class A  { public String getName() { return "A"; } }

public class B1 extends A {
        public String test() { return getName();   /*A*/}
}

public class B2 extends A {
        public String getName() { return "B"; }
        public String test() { return getName(); /*B*/}
}

public class B3 extends A {
        public String getName() { return "B"; }
        public String test() { return super.getName(); /*A*/}
}
```

Mot-clé **super**

Le mot-clé **super** permet d'utiliser une propriété définie au dessus :

```
public class A  { public String name = "A"; }

public class B1 extends A {
        public String test() { return name;   /*A*/}
}

public class B2 extends A {
        public String name = "B";
        public String test() { return name; /*B*/}
}

public class B3 extends A {
        public String name = "B";
        public String test() { return super.name; /*A*/}
}
```

Rappel : Constructeur par défaut

Si aucun constructeur n'est défini, la classe a un **constructeur par défaut**

```
public class A {
        int a = 1;
        int b = 2;
}
```

est équivalent à

```
public class A {
        int a;
        int b;

        public A() {
                a = 1;
                a = 2;
        }
}
```

Construction des instances et extension

Si une classe B etend une classe A, la construction d'une instance de A est nécessaire lors de la construction d'une instance de B

Un appel au constructeur de A est effectué au début du constructeur de B :

```
public class Rectangle extends FormeCliquable {

    public int x1, y1, x2, y2, numero;

    Rectangle(int x1, int y1, int x2, int y2, int numero) {
        /* Appel du constructeur sans paramètre de la classe FormeCliquable */
        this.x1 = x1; this.x2 = x2;
        this.y1 = y1; this.y2 = y2;
        this.numero = numero;
    }
    ...
}
```

Mot-clé **super** lors de la construction

S'il n'y a pas de constructeur vide dans la classe A, il faut préciser les paramètres du constructeur de A en utilisant le mot-clé **super** :

```
public class ClasseAvecNom  {
        String nom;
        public ClasseAvecNom(String nom) { this.nom = nom; }
}

public class MaClasse extends ClasseAvecNom {

        public MaClasse() {
                super("MaClasse");
                ...
        }
}
```

Remarque : la classe `ClasseAvecNom` n'a pas de constructeur vide car un constructeur avec un paramètre est défini.

Modificateur **final**

Le modificateur final permet de bloquer l'extension d'une classe ou la redéfinition d'une méthode.

Exemple 1 :

```
final public class A  { }

public class B extends A { } /*Impossible car la classe A est finale*/
```

Exemple 2 :

```
public class A  { final public void method() { } }

public class B extends A {
        public void method() { }
        /*Impossible car la methode est finale dans A  */
}
```

Classes abstraites

- Une classe dont l'implémentation est incomplète est dite abstraite.
- Les méthodes non-implémentées sont dites abstaites.
- Une classe abstraite n'est pas instanciable.

Exemple :

```
public abstract class SansNom {
 void afficherNom() { System.out.println(obtenirNom()); }
 abstract String obtenirNom();
}

public class A extends SansNom {
 String obtenirNom() { return "A"; }
}

A a = new A();
a.afficherNom(); // affiche "A"
```

Utilisation des classes abstraites

```
public abstract class FormeCliquable implements Cliquable {
    ClickListener[] listeners;
    int nbListeners;

    public FormeCliquable() {
        listeners = new ClickListener[100];
        nbListeners = 0;
    }

    void addClickListener(ClickListener l) {
     listeners[nbListeners] = l; nbListeners++;
    }

    public void processListener() {
        for (int i = 0; i < nbListeners; i++)
                listeners[i].onClick(this);
    }

    public abstract boolean traiterClic(int x, int y);
}
```

Utilisation des classes abstraites

```
public class Disque extends FormeCliquable {

    int x, y, r, numero;

    public Disque(int x, int y, int r, int numero) {
        this.x = x;
        this.y = y;
        this.r = r;
        this.numero = numero;
    }

    public boolean traiterClic(int x, int y) {
        int dx = x - this.x, dy = y - this.y;
        if (Math.sqrt(dx * dx + dy * dy) <= r) {
            processListener();
            return true;
        }
        return false;
    }
}
```

Rappel : Packages et modificateur **public**

- Par défaut, une classe ou une méthode est **non-publique** : elle n'est accessible que depuis les classes du **même** paquet.
- Une classe ou un membre publics est accessible de n'importe où.
- Pour rendre une classe ou un membre **public** :

```
public class ClassePublique {
  public int proprietePublique;
  public void methodePublique() { }
}
```

- Si fichier contient une classe publique, le nom du fichier doit être formé du nom de la classe suivi de ".java".
- Un fichier ne peut contenir qu'une seule classe publique.

Rappel : Modificateur **private** et encapsulation

- Un membre **privé** n'est accessible que par les méthodes de la classe qui le contient.
- Pour rendre un membre privé, on utilise le modificateur **private** :

```
public class ClassePublique {
  private int proprietePrivee;
  private void methodePrivee() { }
}
```

- Encapsulation : Tout ce qui participe à l'implémentation des services doit être privé (afin de permettre la modification de l'implémentation des services sans risquer d'impacter les autres classes)

Modificateur **protected**

- Un membre **protégé** est accessible depuis :
  - les méthodes de la classe qui le contient ;
  - des méthodes des classes qui étendent la classe qui le contient.

- Pour rendre un membre protégé, on utilise le modificateur **protected** :

```
public class ClassePublique {
 protected int proprieteProtegee;
 protected void methodeProtegee() { }
}
```

- Utilisation possible : Commencer l'implémentation d'un service dans une classe et la terminer dans les classes qui l'étendent .
- Possible représentation : Les membres protégés forment une interface entre une classe et les classes qui l'étendent.

Utilisation du modificateur **protected**

```
public abstract class FormeCliquable implements Cliquable {
    ClickListener[] listeners;
    int nbListeners;

    public FormeCliquable() {
        listeners = new ClickListener[100];
        nbListeners = 0;
    }

    void addClickListener(ClickListener l) {
     listeners[nbListeners] = l; nbListeners++;
    }

    protected void processListener() {
        for (int i = 0; i < nbListeners; i++)
                listeners[i].onClick(this);
    }

    public abstract boolean traiterClic(int x, int y);
}
```

La classe **Object**

Par défaut, les classes étendent la classe Object

Consequence : le upcasting vers Object est toujours possible

```
MaClasse c = new MaClasse();
Object o = c;
Object[] t = new Object[10];
for (int i = 0; i < t; i++) {
        if (i%2==0) t[i] = new UneClasse();
        else t[i] = new UneDeuxiemeClasse();
}
```

Rappel : Chaînes de caractères

Deux classes permettent de gérer les chaînes de caractères :

- la classe **String** : chaîne invariable ;
- la classe **StringBuffer** : chaîne destinée à être modifiée (voir API).

Déclaration et création :

```
String h = "Hello";
String w = "World";
```

Concaténation :

```
String hw = h + " "+ w + " ! ";
int c = 13;
String hw12c = h + " "+ w + " " + 12 + " " + c;
```

La conversion est effectée en utilisant l'une des méthodes statiques **valueOf** de la classe **String**.

La méthode **toString()**

Une implémentation possible des `String.valueOf(...)`

```
class String {
    ...
    public static String valueOf(Object obj) {
        if (obj==null) return "null";
        else return obj.toString();
    }

    public static String valueOf(boolean b) {
        if (b) return "true";
        else return "false";
    }
    ...
}
```

Une implémentation possible des `System.out.print(...)` :

```
public void print(String s) { écrire "s" sur la sortie; }
public void print(boolean b) { print(String.valueOf(b)); }
public void print(Object obj) { print(String.valueOf(obj)); }
```

Redéfinir la méthode **toString()**

Redéfinition de la méthode **toString()** :

```
public class MaClasse {

    private int numero;

    public MaClasse(int numero) { this.numero = numero; }

    public String toString() {  return "MaClasse"+numero; }

}
```

Exemple d'utilisation :

```
MaClasse c1 = new MaClasse(1);
MaClasse c2 = new MaClasse(2);
System.out.println(c1.toString() + " " + c2.toString());
System.out.println(c1 + " " + c2);
```

Extension d'interfaces

Il est également possible d'étendre une interface :

```
interface Iterator {
        boolean hasNext();
        Object next();
}
```

```
interface ListIterator extends Iterator {
        boolean hasPrevious();
        Object previous();
}
```

Extension d'interfaces

```
class VectorIterator implements ListIterator {

    final private Vector v;
    private int p;

    public VectorIterator(Vector v) {
        this.v = v; p = 0;
    }

    public boolean hasPrevious() { return (p > 0); }

    public Object previous() { p--; return v.get(p); }

    public boolean hasNext() { return p < v.size(); }

    public Object next() {
        Object o = v.get(p);  p++; return o;
    }
}
```

Extension d'interfaces

Exemples d'utilisation :

```
Vector v = new Vector();
...
ListIterator iterator = new VectorIterator(v);
while (iterator.hasNext())
        System.out.println(iterator.next());
while (iterator.hasPrevious())
        System.out.println(iterator.previous());
...
Iterator iterator = new VectorIterator(v);
while (iterator.hasNext())
        System.out.println(iterator.next());
```

Résumé

- Abstraction
- Etendre une classe
- Transtypage (vers le haut et vers le bas)
- Redéfinition de méthodes et polymorphisme
- mot-clé **super**
- Construction des instances
- Modificateurs **final** et **protected**
- Classes et méthodes abstraites
- La classe **Object**
- La méthode **toString()**

Résumé des cours précédents

Premier cours :

- Objets, classes, instances, références
- Définir et instancier une classe

Deuxième cours :

- Description et implémentation d'interfaces
- Transtypage (vers le haut et vers le bas)

Troisième cours :

- Extension de classe
- Redéfinition de méthodes et polymorphisme
- Classes et méthodes abstraites
- La classe **Object**

Classes internes

- Il est possible de définir une classe à l'intérieur d'une autre classe.
- Une telle classe est dite **interne**.

```
public class ListeChainee {

    private Maillon tete;

    public ListeChainee() { tete = null; }

    public void add(int v) { tete = new Maillon(v, tete); }

    private static class Maillon {
        final int valeur;
        final Maillon suivant;
        public Maillon(int valeur, Maillon suivant) {
            this.valeur = valeur; this.suivant = suivant;
        }
    }
}
```

Classes internes

```
public class Vector {
    private Object[] array;
    private int size;
    ...
    public Iterator iterator() { return new VectorIterator(this); }
}

public class VectorIterator implements Iterator {
    private int position = 0;
    private Vector v;

    public VectorIterator(Vector v) { this.v = v; }

    public boolean hasNext() { return position < v.size; }

    public Object next() {
        Object o = v.array[position]; position++;
        return o;
    }

    public void remove() { }
    }
}
```

Classes internes

Une classe interne peut utiliser les membres de la classe qui la contient :

```
public class Vector {

    private Object[] array;
    private int size;
    ...
    public Iterator iterator() { return new VectorIterator(); }

    private class VectorIterator implements Iterator {
        int position = 0;

        public boolean hasNext() { return position < size; }

        public Object next() {
            Object o = array[position]; position++;
            return o;
        }

        public void remove() { }
    }
}
```

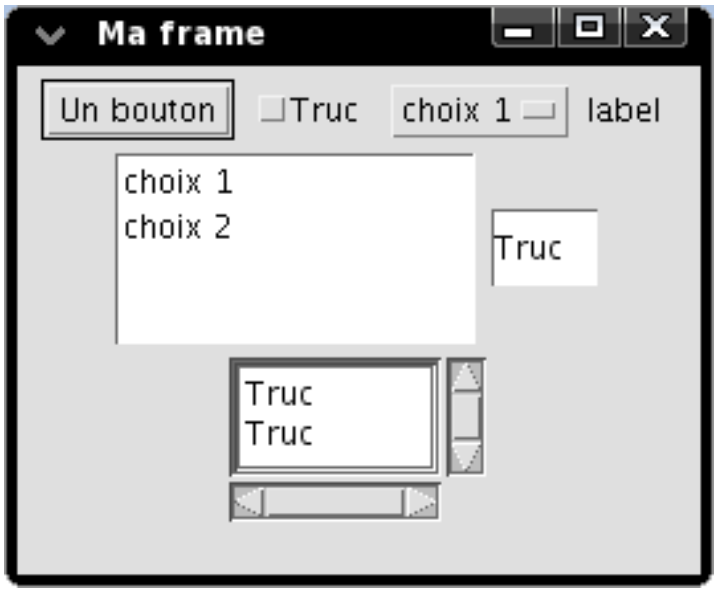
Classes anonymes

```
public class Vector {

    ...

    public Iterator iterator() {
        return new Iterator() {
            int position = 0;

            public boolean hasNext() { return position < size; }

            public Object next() {
                Object o = array[position]; position++;
                return o;
            }

            public void remove() { }

        };
    }
}
```

Syntaxe : `new ClasseOuInterface() { implémentation }`

Introduction à AWT

- **Abstract Window Toolkit** (**AWT**) = bibliothèque graphique Java ;
- **AWT** a été introduite dès les premières versions de Java ;
- **AWT** sert de fondement à **Swing**
- **Swing** = bibliothèque de gestion de fenêtre de JAVA.



Composants

- Une interface graphique est construite à partir de **composants** ;
- Un Composant est un élément visible ;
- Un Composant étend la classe abstraite **Component**.

Les composants de AWT :

- Button
- Canvas
- Checkbox
- Choice
- Label
- TextComponent
  - TextArea
  - TextField
- List
- Scrollbar
- Container

Conteneurs

- Un **conteneur** est un composant ;
- Un conteneur peut contenir plusieurs composants ;
- Un conteneur étend la classe **Container**.

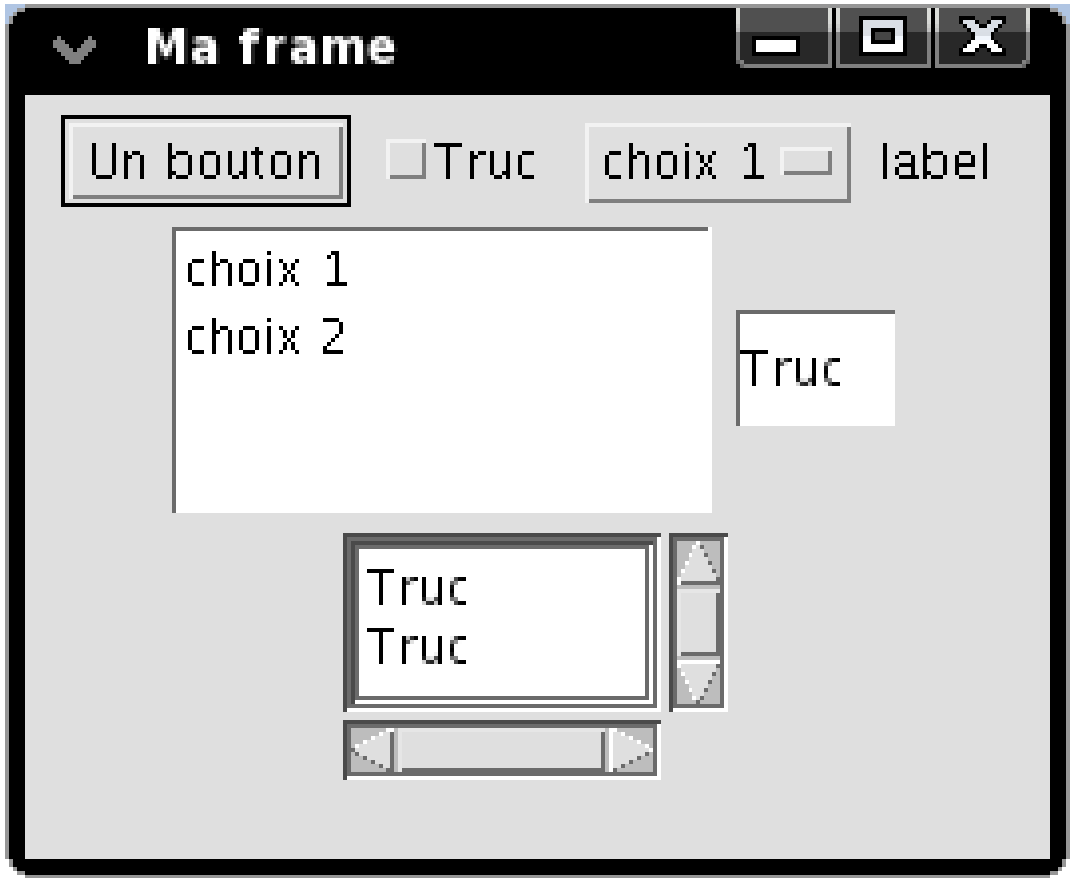
Les principaux conteneurs de AWT :

- Panel
  - Applet
- Window
  - Dialog
    - FileDialog
    - JDialog (Swing)
  - Frame
    - JFrame (Swing)
  - JWindow (Swing)
- JComponent (Swing)

Exemple

```
Frame frame = new Frame("Ma frame");
frame.setPreferredSize(new Dimension(200,200));
Panel panel = new Panel();
panel.add(new Button("Un bouton"));
panel.add(new Checkbox("Truc", false));
Choice choice = new Choice();
choice.add("choix 1"); choice.add("choix 2");
panel.add(choice);
panel.add(new Label("label"));
List list = new List();
list.add("choix 1"); list.add("choix 2");
panel.add(list);
panel.add(new TextField("Truc"));
TextArea textArea = new TextArea("Truc\nTruc");
textArea.setColumns(10); textArea.setRows(3);
panel.add(textArea);
frame.add(panel);
frame.pack();
frame.setVisible(true);
```

Exemple


Ma frame
Un bouton
Truc
choix 1
label
choix 1
choix 2
Truc
Truc
Truc

Gestionnaires de présentation

- Un gestionnaire de présentation implémente **LayoutManager** ;
- Chaque conteneur a un gestionnaire de présentation ;
- Gère la position et la taille des composants présents dans le conteneur.

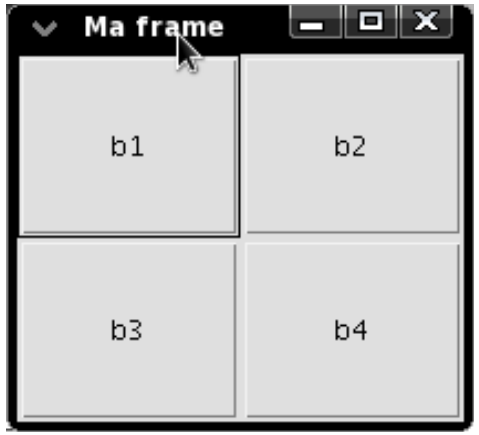
Les principaux gestionnaires de présentation de AWT :

- BorderLayout
- BoxLayout
- CardLayout
- FlowLayout
- GridBagLayout
- GridLayout
- ...

Gestionnaires de présentation

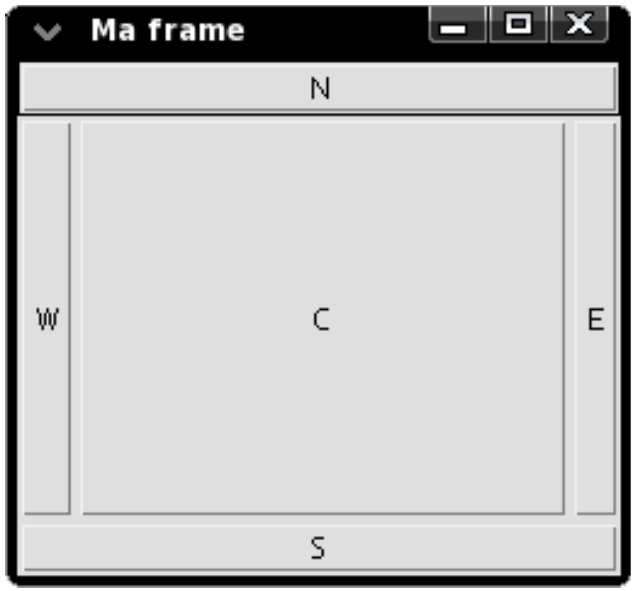
- Chaque conteneur a un gestionnaire de présentation par défaut :
  - Panel → FlowLayout
  - Window → BorderLayout
- Il est possible de changer le gestionnaire de présentation d'un conteneur en utilisant la méthode **setLayout** de la classe **Container**.

```
Frame frame = new Frame("Ma frame");
Panel panel = new Panel();
panel.setLayout(new GridLayout(2,2));
panel.add(new Button("b1"));
panel.add(new Button("b2"));
panel.add(new Button("b3"));
panel.add(new Button("b4"));
frame.add(panel);
frame.pack();
frame.setVisible(true);
```



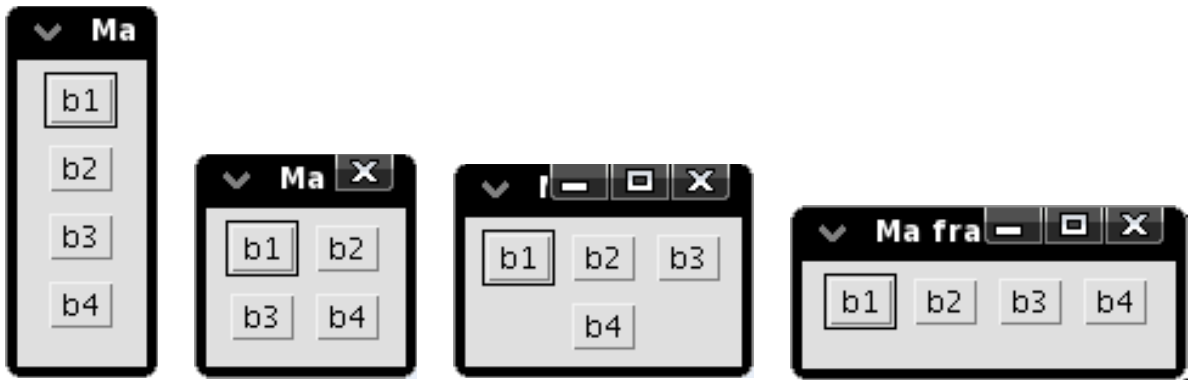
Gestionnaires de présentation (BorderLayout)

```
Frame frame = new Frame("Ma frame");
Panel panel = new Panel();
panel.setLayout(new BorderLayout());
panel.add(new Button("N"), BorderLayout.NORTH);
panel.add(new Button("S"), BorderLayout.SOUTH);
panel.add(new Button("C"), BorderLayout.CENTER);
panel.add(new Button("W"), BorderLayout.WEST);
panel.add(new Button("E"), BorderLayout.EAST);
frame.add(panel);
frame.pack();
frame.setVisible(true);
```



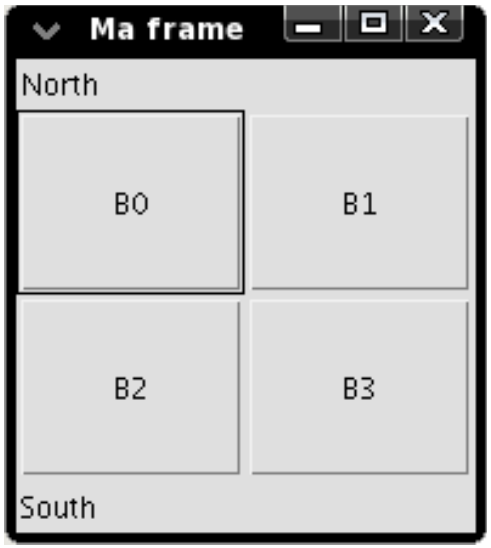
Gestionnaires de présentation (FlowLayout)

```
Frame frame = new Frame("Ma frame");
Panel panel = new Panel();
//panel.setLayout(new FlowLayout());
panel.add(new Button("b1"));
panel.add(new Button("b2"));
panel.add(new Button("b3"));
panel.add(new Button("b4"));
frame.add(panel);
frame.pack();
frame.setVisible(true);
```



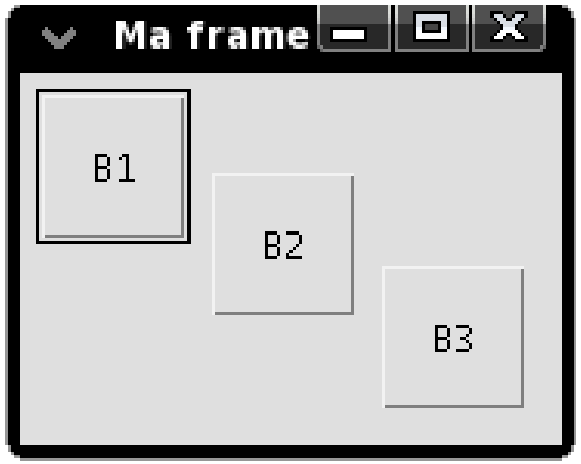
Gestionnaires de présentation (Composition)

```
Frame frame = new Frame("Ma frame");
Panel panel = new Panel();
 panel.setLayout(new BorderLayout());
panel.add(new Label("North"), BorderLayout.NORTH);
panel.add(new Label("South"), BorderLayout.SOUTH);
Panel panelCenter = new Panel();
panelCenter.setLayout(new GridLayout(2,2));
for (int i = 0; i < 4; i++)
    panelCenter.add(new Button("B"+i));
panel.add(panelCenter, BorderLayout.CENTER);
frame.add(panel);
frame.pack();
frame.setVisible(true);
```



Sans gestionnaire de présentation

```
Frame frame = new Frame("Ma frame");
Panel pane = new Panel();
pane.setLayout(null);
Button b1 = new Button("B1");
Button b2 = new Button("B2");
Button b3 = new Button("B3");
pane.add(b1);
pane.add(b2);
pane.add(b3);
b1.setBounds(5, 5, 50, 50);
b2.setBounds(60, 30, 50, 50);
b3.setBounds(115, 60, 50, 50);
frame.setPreferredSize(new Dimension(185,150));
frame.add(pane);
frame.pack();
frame.setVisible(true);
```



Évènements

- Un évènement a lieu quand l'utilisateur agit sur l'interface graphique (clic de souris, bouton pressé, sélection dans une liste, ...) ;
- Un évènement est une instance d'une classe qui étend **AWTEvent** ;
- Les évènements sont émis par les composants ;
- Les composants transmettent les évènements à des observateurs ;
- Un observateur implémente une extension de **EventListener** ;
- Les méthodes de l'interface permettent de recevoir les évènements ;
- Les méthodes **add*Listener** d'un composant permettent d'abonner un observateur aux évènements émis par ce composant.

Évènements

Exemple :

```
public class MyActionListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Clic !!");
    }
}

Button b = new Button("B");
b.addActionListener(new MyActionListener());
panel.add(b);
```

Remarque :

- **ActionListener** est une extension de l'interface **EventListener**
- **ActionEvent** est une extension de l'interface **AWTEvent**

Évènements – Source

```
public class MyActionListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.out.println(e.getSource() /* Soit b1, soit b2 */);
    }
}

ActionListener listener = new new MyActionListener();
Button b1 = new Button("B1");
b1.addActionListener(listener);
panel.add(b1);
Button b2 = new Button("B2");
b2.addActionListener(listener);
```

Exemple de sortie :

```
java.awt.Button[button0,39,5,29x23,label=B2]
java.awt.Button[button1,5,5,29x23,label=B1]
java.awt.Button[button0,39,5,29x23,label=B2]
java.awt.Button[button1,5,5,29x23,label=B1]
```

Évènements – Souris

```
public interface MouseListener extends EventListener {
    public void mouseClicked(MouseEvent e);
    public void mousePressed(MouseEvent e);
    public void mouseReleased(MouseEvent e);
    public void mouseEntered(MouseEvent e);
    public void mouseExited(MouseEvent e);
}

public interface MouseMotionListener extends EventListener {
    public void mouseDragged(MouseEvent e);
    public void mouseMoved(MouseEvent e);
}

public interface MouseWheelListener extends EventListener {
    public void mouseWheelMoved(MouseWheelEvent e);
}
```

Les évènements souris sont observables sur tous les composants

Évènements – Souris

Implémentation de l'interface **MouseListener** :

```
public class SimpleMouseListener implements MouseListener {

    public void mouseClicked(MouseEvent e) {
        System.out.println(e.getX()+" "+e.getY());
    }

    public void mousePressed(MouseEvent e) {    }
    public void mouseReleased(MouseEvent e) {    }
    public void mouseEntered(MouseEvent e) {    }
    public void mouseExited(MouseEvent e) {    }

}
```

Abonnement de l'observateur au composant :

```
Frame frame = new Frame("Ma frame");
frame.addMouseListener(new SimpleMouseListener());
frame.setVisible(true);
```

Évènements – Clavier

```
public interface KeyListener extends EventListener {
    public void keyTyped(KeyEvent e);
    public void keyPressed(KeyEvent e);
    public void keyReleased(KeyEvent e);
}
```

Exemple d'implémentation :

```
public class SimpleKeyListener implements KeyListener {

    public void keyTyped(KeyEvent e) { }

    public void keyPressed(KeyEvent e) {
        System.out.println(e.getKeyChar());
    }

    public void keyReleased(KeyEvent e) { }
}
```

Évènements – Fenêtre

```
public interface WindowListener extends EventListener {
    public void windowOpened(WindowEvent e);
    public void windowClosing(WindowEvent e);
    public void windowClosed(WindowEvent e);
    public void windowIconified(WindowEvent e);
    public void windowDeiconified(WindowEvent e);
    public void windowActivated(WindowEvent e);
    public void windowDeactivated(WindowEvent e);
}

public interface WindowStateListener extends EventListener {
    public void windowStateChanged(WindowEvent e);
}

public interface WindowFocusListener extends EventListener {
    public void windowGainedFocus(WindowEvent e);
    public void windowLostFocus(WindowEvent e);
}
```

Évènements – Adapteurs

- Un **adapteur** est une classe qui implémente un observateur en associant un corps vide à chacune des méthodes.
- Pour éviter d'implémenter toutes les méthodes d'un observateur, on peut étendre l'adapteur et redéfinir certaines de ses méthodes.

```
public class SimpleWindowListener implements WindowListener {
    public void windowOpened(WindowEvent e) { }
    public void windowClosing(WindowEvent e) { System.exit(0); }
    public void windowClosed(WindowEvent e) { }
    public void windowIconified(WindowEvent e) { }
    public void windowDeiconified(WindowEvent e) { }
    public void windowActivated(WindowEvent e) { }
    public void windowDeactivated(WindowEvent e) { }
}
```

est équivalent à

```
public class SimpleWindowListener extends WindowAdapter {
    public void windowClosing(WindowEvent e) { System.exit(0); }
}
```

Évènements – Adapteurs et classes anonymes

Extension d'un adapteur en classe anonyme :

```
Frame frame = new Frame("Ma frame");
frame.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
});
frame.setVisible(true);
```

Exemple – Calculette

```
public class Calculette extends Frame {

    private TextField number1, number2;
    private Label result;

    public Calculette() {
        setPreferredSize(new Dimension(200, 80));
        Panel p = new Panel();
        number1 = new TextField(5);
        number2 = new TextField(5);
        result = new Label("0");
        Button b = new Button("Add");
        b.addActionListener(new BoutonListener());
        p.add(number1);
        p.add(number2);
        p.add(result);
        p.add(b);
        add(p);
        pack();
    }
```

Exemple – Calculette

Suite de la classe Calculette :

```
public class Calculette extends Frame {

    private TextField number1, number2;
    private Label result;

    ....

    private class BoutonListener implements ActionListener {

        public void actionPerformed(ActionEvent e) {
            int v1 = Integer.parseInt(number1.getText());
            int v2 = Integer.parseInt(number2.getText());
            result.setText(""+(v1+v2));
        }
    }
}
```

Exemple – Calculette

La méthode **main** :

```
public static void main(String[] args) {
Calculette calculette = new Calculette();
calculette.setVisible(true);
calculette.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
});
}
```

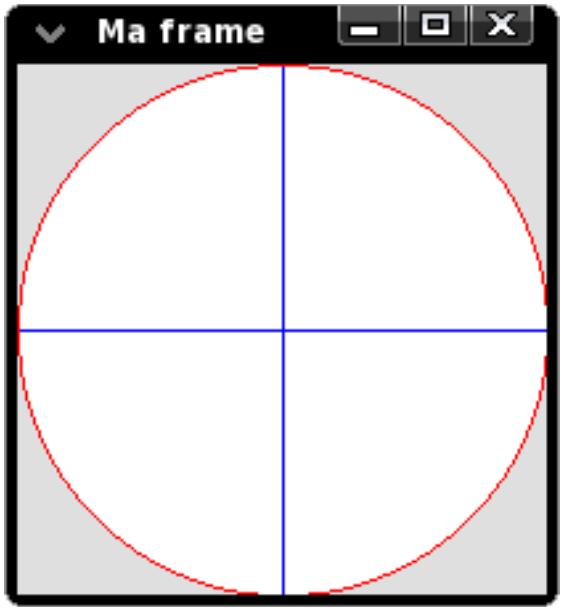
Création de son composant

Il suffit d'étendre la classe **Component**

```
public class MyComponent extends Component {

  public MyComponent() {
    setPreferredSize(new Dimension(200,200));
  }

  public void paint(Graphics g) {
    Dimension size = getSize();
    g.setColor(Color.white);
    g.fillOval(0, 0, size.width, size.height);
    g.setColor(Color.blue);
    g.drawLine(0, size.height/2, size.width, size.height/2);
    g.drawLine(size.width/2, 0, size.width/2, size.height);
    g.setColor(Color.red);
    g.drawOval(0, 0, size.width, size.height);
  }
}
```

Création de son composant

```
Frame frame = new Frame("Ma frame");
frame.add(new MyComponent());
frame.pack();
frame.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
});
frame.setVisible(true);
```



Création de son composant

Changement de la couleur du disque quand on clique sur le composant :

```
public class MyComponent extends Component {

    private Color color = Color.white;

    public MyComponent() {
        setPreferredSize(new Dimension(200,200));
        addMouseListener(new MyMouseAdapter());
    }

    public void paint(Graphics g) {
        g.setColor(color);
        g.fillOval(0, 0, size.width, size.height);
        ...
    }

  ...
}
```

Création de son composant

Changement de la couleur du disque quand on clique sur le composant :

```
public class MyComponent extends Component {

    private Color color = Color.white;

    ...

    private class MyMouseAdapter extends MouseAdapter {

        public void mouseClicked(MouseEvent e) {
            if (color == Color.white)
                color = Color.gray;
            else color = Color.white;
            repaint();
        }
    }

}
```

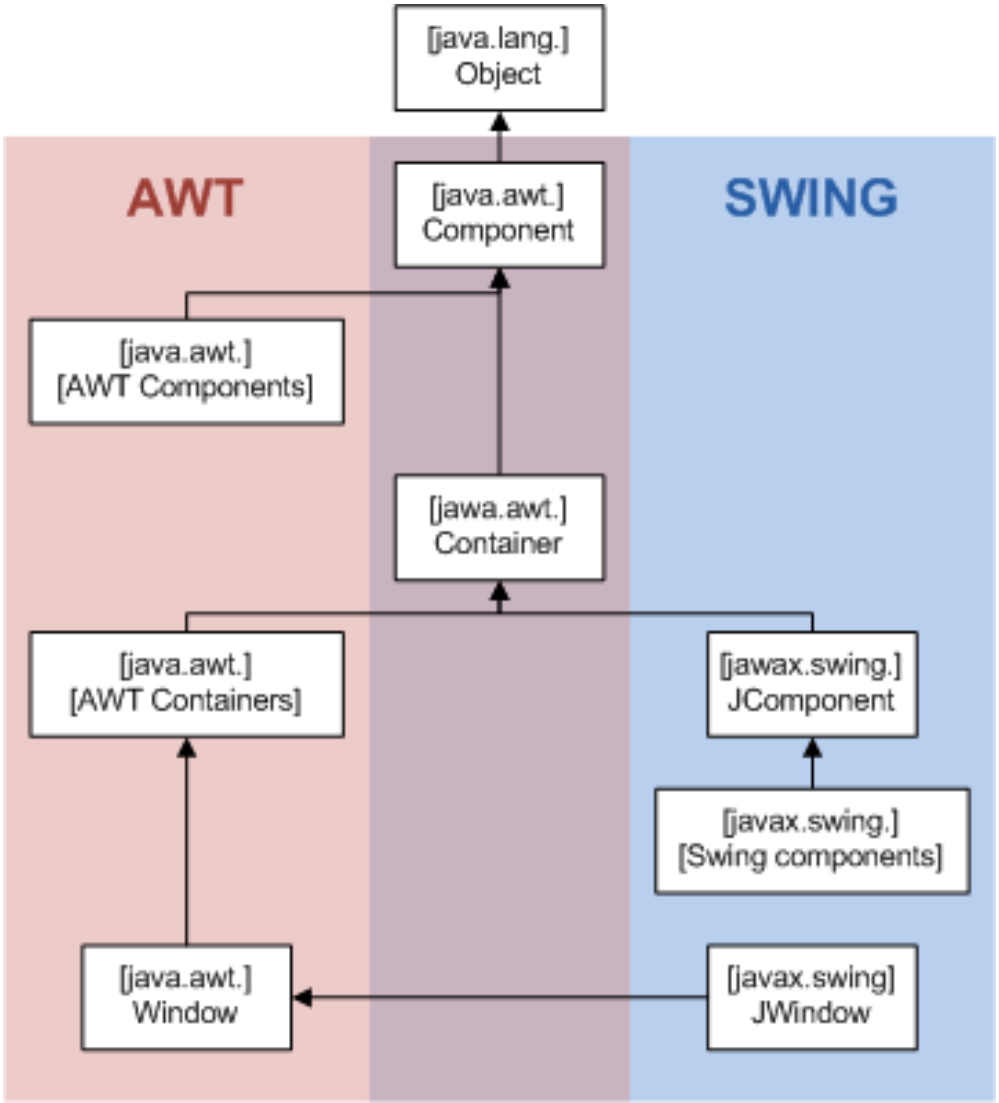
Swing

- **Swing** est plus riche en composants que AWT
- **Swing** propose des composants plus jolis (look and feel)
- **Swing** respecte l'architecture Modèle-Vue-Contrôleur (MVC).

Exemple d'utilisation de Swing :

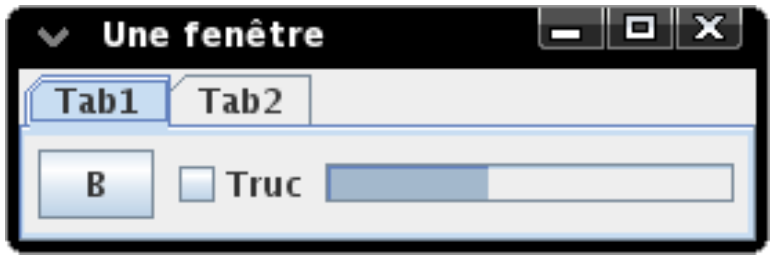
```
public class HelloWorld {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Hello World!");
        frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        JLabel label = new JLabel("Hello, World!");
        frame.getContentPane().add(label);
        frame.pack();
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
    }
}
```

Relations entre Swing et AWT

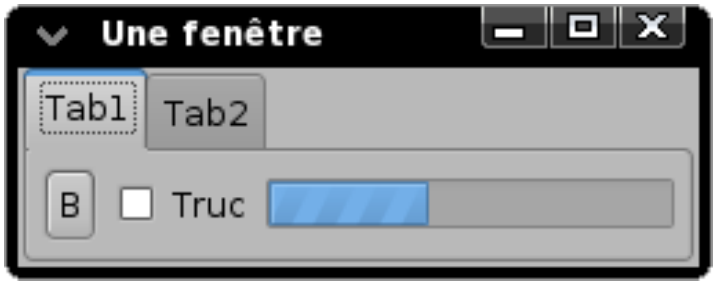


Les composants de Swing

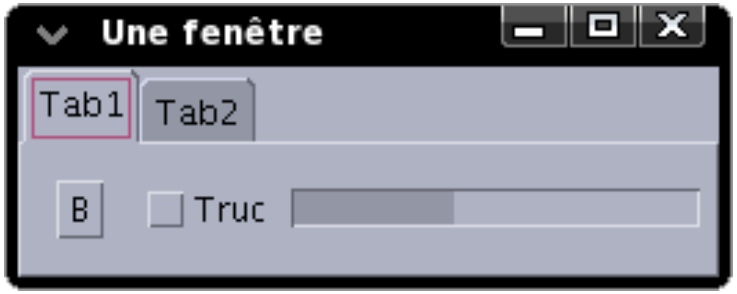
- Eléments de base : JButton, JCheckBox, JComboBox, JMenu, JList, JRadioButton, JSlider, JSpinner, JTextField, JPasswordField, JColorChooser, JEditorPane, JTextPane, JFileChooser, JTable, JTextArea, JTree, JLabel, JProgressBar, JSeparator, JToolTip...
- Conteneurs de haut niveau : JApplet, JDialog, JFrame...
- Conteneurs : JPanel, JScrollPane, JSplitPane, JToolBar, JTabbedPane, JInternalFrame, JLayeredPane...



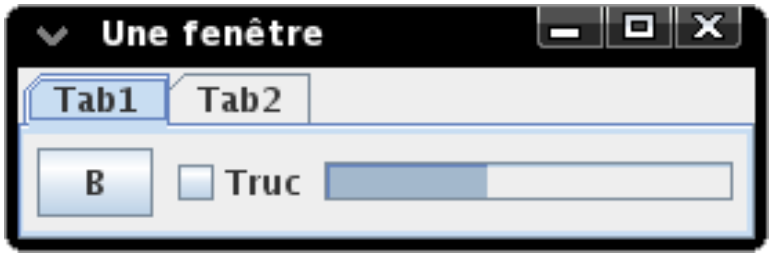
Look and feel



GTKLookAndFeel



MotifLookAndFeel

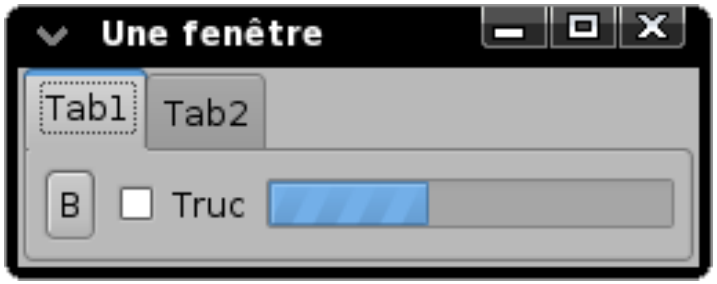


WindowsLookAndFeel

Look and feel

Au début du programme :

```
try {
    UIManager.setLookAndFeel(
        "com.sun.java.swing.plaf.gtk.GTKLookAndFeel");
} catch (Exception e) {
    System.out.println("Error setting LookAndFeel: " + e);
}
```



Création de son composant Swing

```
public class MySwingComponent extends JPanel {

   public MyComponentSwing() {
        setPreferredSize(new Dimension(200,200));
   }

   public void paintComponent(Graphics g) {
      super.paintComponent(g);
      Dimension size = getSize();
      g.setColor(Color.white);
      g.fillOval(0, 0, size.width, size.height);
      g.setColor(Color.blue);
      g.drawLine(0, size.height/2, size.width, size.height/2);
      g.drawLine(size.width/2, 0, size.width/2, size.height);
      g.setColor(Color.red);
      g.drawOval(0, 0, size.width, size.height);
   }

}
```

Remarques sur Swing

- **paint** appelle **paintComponent**, **paintBorder**, **paintChildren**
- Redéfinir **paintComponent** pour changer l'affichage d'un composant
- Pour redessiner le composant, appeler **repaint** (pas **paint**)
- Dans **paintComponent**, commencer par **super.paintComponent(g)**
- Ne pas mélanger des composants de **Swing** et de **AWT**

Résumé des cours précédents

Premier cours :

- Objets, classes, instances, références
- Définir et instancier une classe

Deuxième cours :

- Description et implémentation d'interfaces
- Transtypage (vers le haut et vers le bas)

Troisième cours :

- Extension de classe
- Redéfinition de méthodes et polymorphisme
- Classes et méthodes abstraites
- La classe **Object**

Surcharge de méthodes

- Dans une classe, plusieurs méthodes peuvent avoir le même nom.
- La méthode est choisie par le **compilateur** de la façon suivante :
  - Le nombre de paramètres doit correspondre ;
  - Les affectations des paramètres doivent être valides ;
  - Parmi ces méthodes, le **compilateur** choisit la plus spécialisée.

```
class Additionneur {
    public static int additionner(int a, int b) {
        System.out.println("entier"); return a+b;
    }
    public static double additionner(double a, double b) {
        System.out.println("flottant"); return a+b;
    }
}

int i = 1; double d = 2.2;
double r1 = Additionneur.additionner(d, d); // flottant
double r2 = Additionneur.additionner(i, d); // flottant
int r3 = Additionneur.additionner(i, i); // entier
```

Surcharge de méthodes

- Dans une classe, plusieurs méthodes peuvent avoir le même nom.
- La méthode est choisie par le **compilateur** de la façon suivante :
  - Le nombre de paramètres doit correspondre ;
  - Les affectations des paramètres doivent être valides ;
  - Parmi ces méthodes, le **compilateur** choisit la plus spécialisée.

```
class Afficheur {
    static void Afficher(Object o) {
        System.out.println("Object : "+o);
    }
    static void Afficher(String s) {
        System.out.println("String : "+s);
    }
}

String s = "message";
Object o = s;
Afficheur.Afficher(s); //String : message
Afficheur.Afficher(o); //Object : message
```

Surcharge de méthodes et extension

Attention :

```
class C {}
class D extends C {}

class A {
    public void method(D d) { System.out.println("D"); }
}

class B extends A {
    public void method(C c) { System.out.println("C"); }
}

B b = new B();
b.method(new C()); //affiche "C"
b.method(new D()); //affiche "D"
```

En Java, il n'y a pas de **contravariance** lors de l'extension
(la méthode de B ne redéfinit pas la méthode de A
même si ses arguments sont moins spécialisés)

Covariance

```
class A {
        public Object get() { return "Object"; }
}

class B extends A {
        public String get() { return "String"; }
}

B b = new B(); A a = b;
String s = b.get();
System.out.println(s); // Affiche "String"
Object o = a.get();    // Le type de retour est Object (et pas String)
System.out.println(o); // Affiche "String"
a = new A();
o = a.get();
System.out.println(o); // Affiche "Object"
```

Covariance : on peut modifier le type de la valeur retournée par une méthode que l'on redéfinit si le nouveau type étend l'ancien

Types paramétrés – Pourquoi ?

La classe **Pile** :

```
public class Pile {

    private Object[] pile;
    private int taille;

    public Pile() { pile = new Object[100]; taille = 0; }

    public void empiler(Object o) {
        pile[taille] = o; taille++;
    }

    public Object depiler() {
        taille--; Object o = pile[taille]; pile[taille]=null;
        return o;
    }
}
```

Types paramétrés – Pourquoi ?

Premier problème :

```
Pile p = new Pile();
String s = "truc";
p.empiler(s);     // Ok car String étend Object
s = (String)p.depiler(); // Transtypage obligatoire
```

Deuxième problème :

```
Pile p = new Pile();
Integer i = new Integer(2);
p.empiler(i);     // Ok car Integer étend Object
String s = (String)p.depiler(); // Erreur à l'exécution (String != Integer)
```

Solution : **préciser le type des éléments autorisés dans la pile**

```
Pile<String> p = new Pile<String>();
String s = "truc";
p.empiler(s);     // Ok car s est de type String
String s = p.depiler(); // Ok car la pile ne contient que des String
```

Types paramétrés – Classes paramétrées

```
public class Pile<T> {

    private Object[] pile;   (En Java, impossible de définir de tableau de T)
    public int taille;

    public Pile() { pile = new Object[100]; taille = 0; }

    public void empiler(T o) {
        pile[taille] = o; taille++;
    }

    public T depiler() {
        taille--; T e = pile[taille]; pile[taille]=null;
        return (T)e;
    }
}
```

En Java, les types paramétrés ajoutent des vérifications de type et des transtypages automatiques lors de la compilation mais ne modifient pas le bytecode généré.

Boxing et unboxing

Comment obtenir une pile d'entiers ?

```
Pile<int> p = new Pile<int>();
```
Impossible : int n'est pas le nom d'une classe

Il faut utiliser la classe d'emballage **Integer** associé au type simple **int** :

```
Pile<Integer> p = new Pile<Integer>();
int i = 2;
Integer j = new Integer(i);  (empaquetage du int dans un Integer)
p.empiler(j);
Integer k = p.depiler();
int l = j.intValue();  (déballage du int présent dans le Integer)
```

Depuis Java 5, autoboxing et auto-unboxing :

```
Pile<Integer> p = new Pile<Integer>();
int i = 2;
p.empiler(i);  (empaquetage du int dans un Integer)
int l = p.depiler();  (déballage du int présent dans le Integer)
```

(Attention : des allocations sont effectuées sans new dans le code)

Rappel : Les types primitifs

byte	entier	8 bits	-128 à 127	0
short	entier	16 bits	-32768 à 32767	0
int	entier	32 bits	-2^{31} à $2^{31}-1$	0
long	entier	64 bits	-2^{63} à $2^{63}-1$	0
float	flotant	32 bits		0.0
double	flotant	64 bits		0.0
char	caractère	16 bits	caractères Unicode	\u0000
boolean	boolean	1 bit	false ou true	false

Classes d'emballage

La classe **Number** :

- public abstract int intValue ()
- public abstract long longValue ()
- public abstract float floatValue ()
- public abstract double doubleValue ()
- public byte byteValue()
- public short shortValue()

Les classes d'emballage qui étendent Number :

- **Byte** : public Byte(byte b)
- **Short** : public Short(short s)
- **Integer** : public Integer(int i)
- **Long** : public Long(long l)
- **Float** : public Float(float f)
- **Double** : public Double(double d)

Classes d'emballage

La classe **Boolean** :

- public Boolean(bool b)
- public boolean booleanValue()

La classe **Character** :

- public Character(char c)
- public char charValue()
- public static boolean isLowerCase (char ch)
- public static boolean isUpperCase (char ch)
- public static boolean isTitleCase (char ch)
- public static boolean isDefined (char ch)
- public static boolean isDigit (char ch)
- public static boolean isLetter (char ch)
- public static boolean isLetterOrDigit (char ch)
- public static char toLowerCase (char ch)
- public static char toUpperCase (char ch)
- public static char toTitleCase (char ch)

Types paramétrés – Implémentation

```
public interface Comparable<T> { (Interface Java)
    public int compareTo(T o);
}

class Carte implements Comparable<Carte> {

    public int coul;
    public int val;

    public Carte(int c, int v) { coul = c; val = v; }

    public int compareTo(Carte c) {
        int cc = coul - c.coul;
        if (cc!=0) return cc;
        else return val - c.val;
    }

    public String toString() { return "["+coul+","+val+"]"; }
}
```

Types paramétrés – Condition sur les paramètres

```
class SortedArray<T extends Comparable<T>> {

    private Object[] tab;

    public int taille;

    public SortedArray() { tab = new Object[100]; taille = 0; }

    public T get(int p) { return (T)tab[p]; }

    public void add(T e) {
        int p = 0;
        while (p < taille && e.compareTo(get(p))>0) p++;
        for (int i = taille; i > p; i--) tab[i] = tab[i-1];
        tab[p] = e;
        taille++;
    }
}
```

Méthodes paramétrées et surcharge

```
class Tools {

    static boolean isSorted(String[] t) {
        for (int i = 0; i < t.length-1; i++)
            if (t[i].length() > t[i+1].length()) return false;
        return true;
    }

    static <T extends Comparable<T>> boolean isSorted(T[] t) {
        for (int i = 0; i < t.length-1; i++)
            if (t[i].compareTo(t[i+1]) > 0) return false;
        return true;
    }
}
```

Exemple 1 :

```
Carte[] tab = new Carte[3]; tab[0] = new Carte(1,2);
tab[1] = new Carte(2,3); tab[2] = new Carte(2,4);
System.out.println(Tools.isSorted(tab));
```

Méthodes paramétrées et surcharge

```
class Tools {

    static boolean isSorted(String[] t) {
        for (int i = 0; i < t.length-1; i++)
            if (t[i].length() > t[i+1].length()) return false;
        return true;
    }

    static <T extends Comparable<T>> boolean isSorted(T[] t) {
        for (int i = 0; i < t.length-1; i++)
            if (t[i].compareTo(t[i+1]) > 0) return false;
        return true;
    }
}
```

Exemple 2 :

```
String[] tab = new String[3]; tab[0] = "toto";
tab[1] = "truc"; tab[2] = "abc";
System.out.println(Tools.isSorted(tab));
```

Méthodes paramétrées et surcharge

```
class Tools {

    static boolean isSorted(String[] t) {
        for (int i = 0; i < t.length-1; i++)
            if (t[i].length() > t[i+1].length()) return false;
        return true;
    }

    static <T extends Comparable<T>> boolean isSorted(T[] t) {
        for (int i = 0; i < t.length-1; i++)
            if (t[i].compareTo(t[i+1]) > 0) return false;
        return true;
    }
}
```

Exemple 3 :

```
Object[] tab = new String[3]; tab[0] = "toto";
tab[1] = "truc"; tab[2] = "abc";
System.out.println(Tools.isSorted(tab)); Erreur de compilation
```

Les itérateurs

```
public interface Iterable<T> { (Interface Java)
        public Iterator<T> iterator();
}

public interface Iterator<T> { (Interface Java)
        public boolean hasNext();
        public T next();
        public void remove();
}
```

Exemple d'utilisation :

```
static <T> void printCollection(Iterable<T> t) {
        Iterator<T> it = t.iterator();
        while (it.hasNext()) {
            T e = it.next();
            System.out.println(e);
        }
}
```

Les itérateurs – implémentation

```
class SortedArrayIterator<T extends Comparable<T>>
                                    implements Iterator<T> {

    int pos;
    SortedArray<T> sa;

    SortedArrayIterator(SortedArray<T> sa) {
        pos = 0; this.sa = sa;
    }

    public boolean hasNext() { return pos < sa.taille; }

    public T next() {
        T e = sa.get(pos); pos++;
        return e;
    }

    public void remove() { ... }
}
```

Les itérateurs – implémentation

```
class SortedArray<T extends Comparable<T>>
                                    implements Iterable<T>  {

    private Object[] tab;

    public int taille;

    public SortedArray() { tab = new Object[100]; taille = 0; }

    public T get(int p) { return (T)tab[p]; }

    ...

    public SortedArrayIterator<T> iterator() {
        return new SortedArrayIterator(this);
    }

}
```

Les itérateurs – utilisation

Exemple de méthode générique :

```
static <T> void printCollection(Iterable<T> t) {
        Iterator<T> it = t.iterator();
        while (it.hasNext()) {
            T e = it.next();
            System.out.println(e);
        }
}
```

Exemple d'utilisation :

```
SortedArray<Carte> a = new SortedArray<Carte>();
a.add(new Carte(1, 2));
a.add(new Carte(2, 1));
a.add(new Carte(1, 4));
Tools.printCollection(a);
```

Java (5 et plus) et les itérateurs :

```
for (Carte c : a) { System.out.println(c); }
```

Types paramétrés – Plusieurs paramètres

Il est possible d'avoir plusieurs paramètres :

```
public class Pair<A, B> {

    public A first;
    public B second;

    public Pair(A f, B s) { first = f; second = s; }

    public static <A, B> Pair<A,B> makePair(A f, B s) {
        return new Pair<A,B>(f,s);
    }

}
```

Exemple d'utilisation de la classe **Pair** :

```
Pair<Integer, Integer> p = new Pair<Integer, Integer>(2,3);
Pair<Integer, Integer> p = Pair.makePair(2, 3);
Pair<Integer, Integer> p = Pair.makePair(2.5, 3);
```

Types incompatibles : Pair<Integer, Integer> != Pair<Double, Integer>

Types paramétrés – ? super

Problème :

```
class Carte implements Comparable<Carte> { ... }
class JolieCarte extends Carte { ... }

class SortedArray<T extends Comparable<T>> { ... }
SortedArray<JolieCarte> s = new SortedArray<JolieCarte>();
// Erreur : JolieCarte n'implémente pas Comparable<JolieCarte>
```

Solution :

```
class SortedArray<T extends Comparable<? super T>> { ... }

SortedArray<JolieCarte> s = new SortedArray<JolieCarte>();
// Ok : JolieCarte implémente Comparable<Carte>
```

Types paramétrés – ? super

Rien ne change dans la classe **SortedArray** :

```
class SortedArray<T extends Comparable<? super T>> {
    ...
    public T get(int p) { return (T)tab[p]; }

    public void add(T e) {
        int p = 0;
        while (p < taille && e.compareTo(get(p))>0) p++;
        for (int i = taille; i > p; i--) tab[i] = tab[i-1];
        tab[p] = e;
        taille++;
    }
    ...
}
```

- Le type du paramètre de **e.compareTo** est **“? super T”**
- La méthode **get(p)** retourne un paramètre de type **T**
- **T** est compatible avec **“? super T”** => Les types sont corrects.

Types paramétrés – ? extends

Problématique :

```
class Carte implements Comparable<Carte> { ... }
class JolieCarte extends Carte { ... }

class SortedArray<T extends Comparable<? super T>>
                                    implements Iterable<T> {
   ...
   public void addAll(Iterable<T> c) {
    for (T e : c) add(e);
   }
   ...
}

SortedArray<Carte> c = new SortedArray<Carte>();
SortedArray<JolieCarte> jc = new SortedArray<JolieCarte>();

c.addAll(jc);
// Erreur : SortedArray<JolieCarte> n'implémente pas Iterable<Carte>
```

Types paramétrés – ? extends

Solution :

```
class Carte implements Comparable<Carte> { ... }
class JolieCarte extends Carte { ... }

class SortedArray<T extends Comparable<? super T>>
                                    implements Iterable<T> {
  ...
  public void addAll(Iterable<? extends T> c) {
   for (T e : c) add(e);
  }
  ...
}

SortedArray<Carte> c = new SortedArray<Carte>();
SortedArray<JolieCarte> jc = new SortedArray<JolieCarte>();

c.addAll(jc); // Ok : SortedArray<JolieCarte> implémente Iterable<JolieCarte>
```

Types paramétrés – ? extends

Explication :

```
class SortedArray<T extends Comparable<? super T>>
                                        implements Iterable<T> {
   ...
 public void addAll(Iterable<? extends T> c) {
        Iterator<? extends T> it = c.iterator();
        while (it.hasNext()) {
            T e = it.next();
            add(e);
        }
   }
   ...
}
```

- Le retour de **it.next()** est de type **“ ? extends T”**
- Il peut donc être transtypé (vers le haut) en **T**
- **“ ? extends T”** est compatible avec **T** => Les types sont corrects.

Résumé

- Surcharge de méthodes
- Covariance
- Boxing, unboxing et classes d'emballage
- Classes paramétrées
- Condition sur les paramètres
- Méthodes paramétrées
- Itérateurs
- "**? super**" et "**? extends**"

Résumé des cours précédents

Premier cours :

- Objets, classes, instances, références
- Définir et instancier une classe

Deuxième cours :

- Description et implémentation d'interfaces
- Transtypage (vers le haut et vers le bas)

Troisième cours :

- Extension de classes, redéfinition de méthodes
- Classes et méthodes abstraites

Cinquième cours :

- Surcharge de méthodes
- Types paramétrés

Structures de données Java

Des interfaces :

- **Collection<V>** : Groupe d'éléments
  - **List<V>** : Liste d'éléments ordonnés et accessibles via leur indice
  - **Set<V>** : Ensemble d'éléments uniques
  - **Queue<V>** : Une file d'éléments (FIFO)
  - **Deque<V>** : Une file à deux bouts (FIFO-LIFO)
- **Map<K,V>** : Ensemble de couples clé-valeur.

Il est préférable d'utiliser les interfaces pour typer les variables :

```
List<Integer> l = new ArrayList<Integer>();
```
(code qui utilise l)

car on peut changer la structure de données facilement :

```
List<Integer> l = new LinkedList<Integer>();
```
(code qui utilise l et qui n'a pas à être modifié)

Les collections

Les méthodes de l'interface **Collection<V>** :

- `boolean add(V e)`
- `boolean addAll(Collection<? extends V> c)`
- `void clear()`
- `boolean contains(Object o)`
- `boolean containsAll(Collection<?> c)`
- `boolean isEmpty()`
- `boolean remove(Object o)`
- `boolean removeAll(Collection<?> c)`
- `boolean retainAll(Collection<?> c)`
- `int size()`
- `Object[] toArray()`
- `<T> T[] toArray(T[] a)`

Les listes

Les méthodes de l'interface **List<V>** :

- `void add(int index, V element)`
- `boolean addAll(int index, Collection<? extends V> c)`
- `V get(int index)`
- `int indexOf(Object o)`
- `int lastIndexOf(Object o)`
- `V remove(int index)`
- `int indexOf(Object o)`
- `V set(int index, V element)`
- `List<V> subList(int fromIndex, int toIndex)`
- + les méthodes de **Collection<V>**

Les listes

Quelques implémentations de l'interface **List<V>** :

- ArrayList<V> : Tableau dont la taille varie dynamiquement.
- LinkedList<V> : Liste chaînée.
- ~~Vector<V> : Comme ArrayList mais synchronisé.~~
- Stack<V> : Pile (mais une implémentation de Deque est préférable).

Les maps

Les méthodes de l'interface **Map<K,V>** :

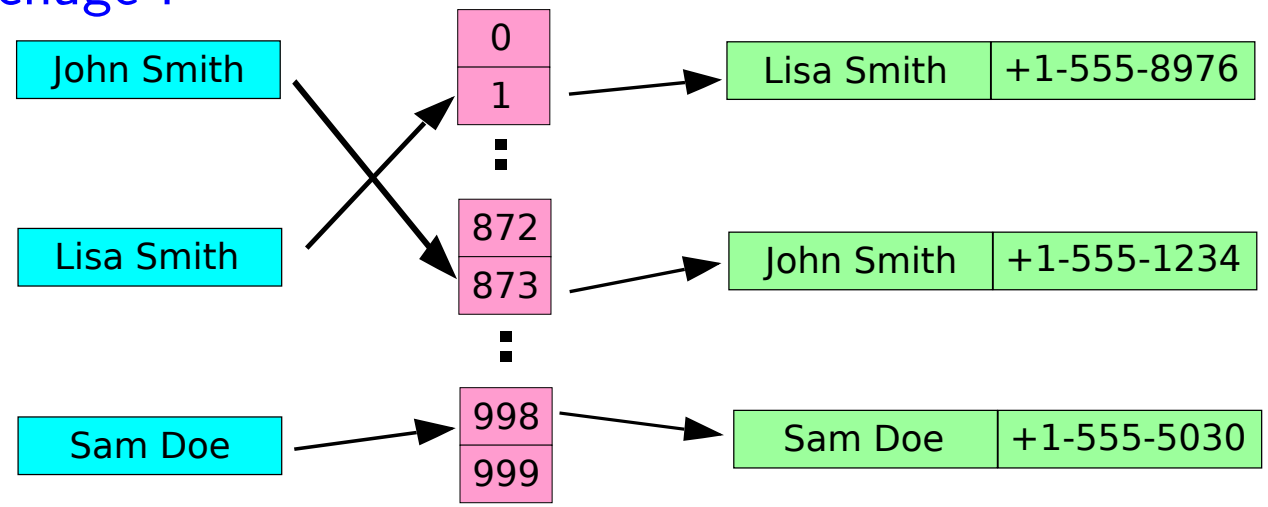
- `clear()`
- `boolean containsKey(Object key)`
- `boolean containsValue(Object value)`
- `Set<Map.Entry<K,V>> entrySet()`
- `V get(Object key)`
- `boolean isEmpty()`
- `Set<K> keySet()`
- `V put(K key, V value)`
- `void putAll(Map<? extends K,? extends V> m)`
- `V remove(Object key)`
- `int size()`
- `Collection<V> values()`

Les maps

Quelques implémentations de l'interface **Map<K,V>** :

- HashMap : table de hachage
- LinkedHashMap : table de hachage + listes chainées
- TreeMap : arbre rouge-noir (Les éléments doivent être **comparables**)

Table de hachage :



Calcul de l'indice : int indice = key.hashcode() & (taille - 1) ;
Calcul du hashcode : **int hashcode()** (méthode de la classe **Object**)

Les maps

Quelques implémentations de l'interface **Map<K,V>** :

- HashMap : table de hachage
- LinkedHashMap : table de hachage + listes chainées
- TreeMap : arbre rouge-noir (Les éléments doivent être **comparables**)

Arbre rouge-noir :

	Complexité
Rechercher	$O(\log n)$
Insérer	$O(\log n)$
Supprimer	$O(\log n)$

Les maps

Quelques implémentations de l'interface **Map<K,V>** :

- HashMap : table de hachage
- LinkedHashMap : table de hachage + listes chainées
- TreeMap : arbre rouge-noir (Les éléments doivent être **comparables**)

Arbre rouge-noir et interface **Comparator** :

```
Map<Carte, Integer> m = new TreeMap<Carte, Integer>(
        new Comparator<Carte>() {
                public int compare(Carte o1, Carte o2) {
                        ...
                }
```

Les ensembles

Set<V> ne contient que les méthodes de **Collection<V>**

Quelques implémentations de l'interface **Set<V>** :

- HashSet<V> : avec une HashMap.
- LinkedHashSet<V> : avec une LinkedHashMap.
- TreeSet<V> : avec une TreeMap.

Les files

Les méthodes de l'interface **Queue<V>** :

- `V element()`
- `boolean offer(V e)`
- `V peek()`
- `V poll()`
- `V remove()`
- + les méthodes de **Collection<V>**

Les files à deux bouts (Deque)

Les méthodes de l'interface **Deque<V>** :

	Premier élément		**Dernier élément**	
	avec exception	*avec null*	*avec exception*	*avec null*
Insérer	addFirst(e)	offerFirst(e)	addLast(e)	offerLast(e)
Supprimer	removeFirst()	pollFirst()	removeLast()	pollLast()
Consulter	getFirst()	peekFirst()	getLast()	peekLast()

Correspondance avec les méthodes de **Queue<V>** :

Dans Queue	**Dans Deque**
add(e)	addLast(e)
offer(e)	offerLast(e)
remove()	emoveFirst()
poll()	pollFirst()
element()	getFirst()
peek()	peekFirst()

Les files à deux bouts (Deque)

Quelques implémentations de l'interface **Deque<V>** :

- ArrayDeque<V> : avec un tableau dynamique.
- LinkedList<V> : avec une liste chaînée.

Les iterateurs

L'interface **Collection<V>** étend **Iterable<V>** :

```
Collection<Integer> l = new ArrayList<Integer>();
l.add(1); l.add(2); l.add(1);
for (Integer i : l)
    System.out.print(i+" ");
System.out.println();
```

Sortie : 1 2 1

```
Collection<Integer> l = new HashSet<Integer>();
l.add(1); l.add(2); l.add(1);
for (Integer i : l)
    System.out.print(i+" ");
System.out.println();
```

Sortie : 1 2

Les iterateurs

Exemple avec une HashMap :

```
Map<String, Integer> m = new HashMap<String, Integer>();
m.put("toto", 4);
m.put("aaa", 3);
m.put("bb", 2);
for (Integer i : m.values())
    System.out.print(i+" ");
System.out.println();
for (String k : m.keySet())
    System.out.print(k+" ");
System.out.println();
```

Sortie :

```
3 4 2
aaa toto bb
```

Exceptions

- Tout programme peut être confronté à une condition exceptionnelle (ou *exception*) durant son exécution.
- Une exception est une situation qui empêche l'exécution normale du programme (elle n'est pas un *bug*).

Exemple :

- Un fichier nécessaire à l'exécution du programme n'existe pas.
- Division par zéro
- Débordement dans un tableau
- etc.

Mécanisme de gestion des exceptions

- Java propose un mécanisme de gestion des exceptions afin de distinguer l'exécution normale du traitement des erreurs et faciliter la gestion des exceptions.
- En Java, une exception est une instance d'une classe qui **étend** la classe **Exception**
- Pour lever (déclencher) une exception, on utilise le mot-clé **throw** :

```
if (problem) throw new MyException("erreur de type 2");
```

- Pour capturer une exception, on utilise la syntaxe **try/catch** :

```
try {
  //Problème possible
} catch (MyException e) {
  TraiterException(e);
}
```

Définir son propre type d'exception

Il suffit d'étendre la classe **Exception** (ou une qui étend **Exception**) :

```
public class MyException extends Exception {

    int number;

    public MyException(int number) {
        this.number = number;
    }

    public String getMessage() {
        return "Erreur numero "+number;
    }

}
```

Convention de nommage : *quelquechose*Exception

La syntaxe **try/catch**

```
public static void test(int i) {
  System.out.print("A ");
  try {
     System.out.println("B ");
     if (i > 12) throw new MyException(i);
     System.out.print("C ");
  } catch (MyException e) {
     System.out.println(e);
  }
  System.out.println("D");
}
```

test(11) :

```
A B
C D
```

test(13) :

```
A B
MyException: Erreur numéro 13
D
```

Pile d'appels et exceptions

```
public class Test {

    public static void method1(int i) throws MyException {
        method2(i);
    }

    public static void method2(int i) throws MyException {
        if (i>12) throw new MyException(i);
    }

    public static void main(String arg[]) {
        try { method1(i); }
        catch (MyException e) { e.printStackTrace(); }
    }
}
```

Une méthode doit indiquer toutes les exceptions qu'elle peut générer et qu'elle n'a pas traitées avec un bloc try/catch

(Partiellement vrai => voir plus loin)

Pile d'appels et exceptions

```
public class Test {

    public static void method1(int i) throws MyException {
        method2(i);
    }

    public static void method2(int i) throws MyException {
        if (i>12) throw new MyException(i);
    }

    public static void main(String arg[]) {
        try { method1(i); }
        catch (MyException e) { e.printStackTrace(); }
    }
}
```

```
MyException: Erreur numéro 13
        at Test.method2(Test.java:12)
        at Test.method1(Test.java:8)
        at Test.test(Test.java:17)
        at Test.main(Test.java:24)
```

La classe **RuntimeException**

Une méthode doit indiquer toutes les exceptions qu'elle peut générer
sauf si l'exception étend la classe **RuntimeException**

Bien évidemment, la classe **RuntimeException** étend **Exception**

Quelques classes Java qui étendent **RuntimeException** :

- ArithmeticException
- ClassCastException
- IllegalArgumentException
- IndexOutOfBoundsException
- NegativeArraySizeException
- NullPointerException

Capturer plusieurs types exceptions

```
public static int diviser(Integer a, Integer b) {
        try {
            return a/b;
        } catch (ArithmeticException e) {
            e.printStackTrace();
            return Integer.MAX_VALUE;
        } catch (NullPointerException e) {
            e.printStackTrace();
            return 0;
        }
    }

}
```

diviser(12,0) :

```
java.lang.ArithmeticException: / by zero
        at Test.diviser(Test.java:17)
        at Test.main(Test.java:28)
2147483647
```

diviser(null,12) :

```
java.lang.NullPointerException
        at Test.diviser(Test.java:17)
        at Test.main(Test.java:28)
0
```

Le mot-clé **finally**

```
public static void readFile(String file) {
    try {
        FileReader f = new FileReader(file);
        (le constructeur de FileReader peut déclencher une FileNotFoundException)

        try {
            int ch = f.read(); (peut déclencher une IOException)
            while (ch!=-1) {
                System.out.println(ch);
                ch = f.read(); (peut déclencher une IOException)
            }
        } finally { (à faire dans tous les cas)
            f.close();
        }

    } catch (IOException e) { e.printStackTrace(); }
}
```

FileNotFoundException étend **IOException** donc elle est capturée

Le mot-clé **finally**

```
public static void readFile(String file) {
    try {
        FileReader f = new FileReader(file);
        (le constructeur de FileReader peut déclencher une FileNotFoundException)

        try {
            int ch = f.read(); (peut déclencher une IOException)
            while (ch!=-1) {
                System.out.println(ch);
                ch = f.read(); (peut déclencher une IOException)
            }
        } finally { (à faire dans tous les cas)
            f.close();
        }
    }
    catch (FileNotFoundException e) {
        System.out.println("Fichier "+file+" introuvable");
    }
    catch (IOException e) { e.printStackTrace(); }
}
```

Exemple – La pile

```
public class Pile<T> {
    private Object[] pile;
    private int taille;

    public Pile(int capacity) {
        pile = new Object[capacity]; taille = 0;
    }

    public void empiler(T o) throws PilePleineException {
        if (taille == pile.length)
                throw new PilePleineException();
        pile[taille] = o; taille++;
    }

    public T depiler() throws PileVideException {
        if (taille == 0) throw new PileVideException();
        taille--; T e = (T)pile[taille]; pile[taille]=null;
        return (T)pile[taille];
    }
}
```

Exemple – La pile

Définition des exceptions :

```
public class PileException extends Exception {
    public PileException(String msg) { super(msg); }
}

public class PilePleineException extends PileException {
    public PilePleineException() { super("Pile pleine"); }
}

public class PileVideException extends PileException {
    public PileVideException() { super("Pile vide"); }
}
```

Exemple – La pile

Exemples d'utilisation :

```
Pile<Integer> p = new Pile<Integer>(2);
```

```
try {
 p.empiler(1);
 p.empiler(2);
 p.empiler(3);
} catch (PileException e) {
 e.printStackTrace();
}
```

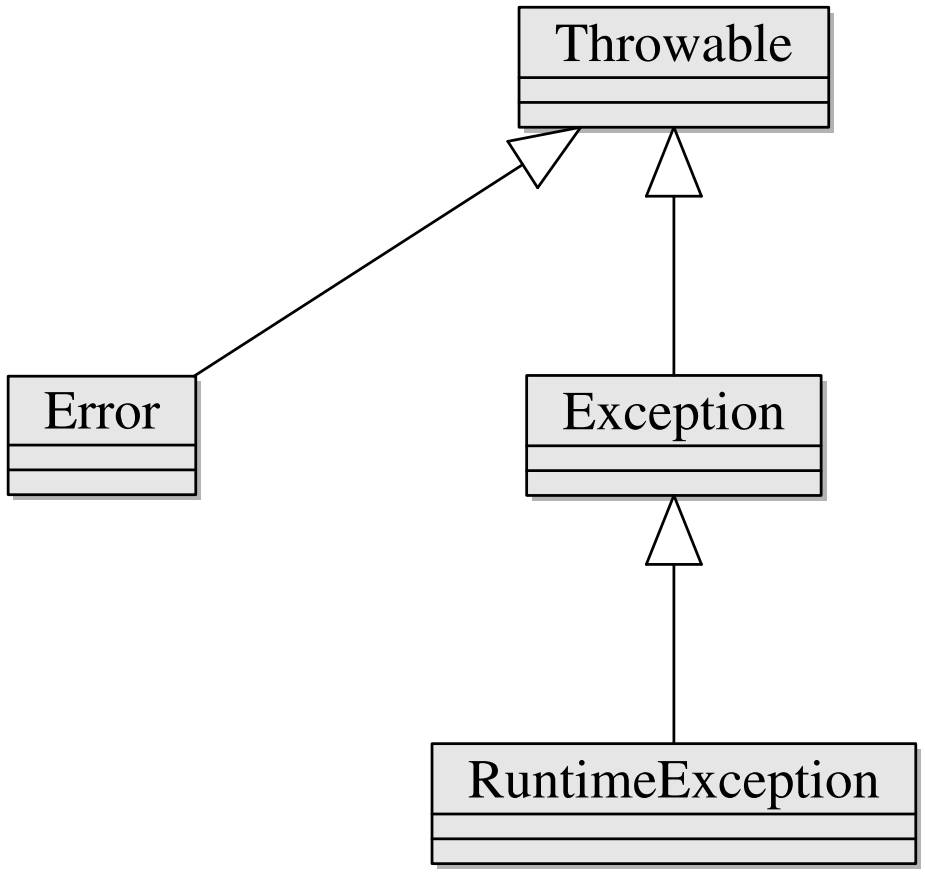
```
PilePleineException: Pile pleine
        at Pile.empiler(Pile.java:14)
        at Test.main(Test.java:58)
```

```
try {
 p.empiler(1);
 p.depiler();
 p.depiler();
} catch (PileException e) {
 e.printStackTrace();
}
```

```
PileVideException: Pile vide
        at Pile.depiler(Pile.java:19)
        at Test.main(Test.java:67)
```

La classe **Throwable**

En Java, toutes les instances des classes
qui étendent **Throwable** peuvent être jetées et capturées



La classe **Throwable**

La classe **Throwable** fournit les méthodes suivantes :

Throwable
Throwable() Throwable(message:String) Throwable(message:String, cause:Throwable) Throwable(cause:Throwable) fillInStackTrace():Throwable getCause():Throwable getLocalizedMessage():String getMessage():String getStackTrace():StackTraceElement[] initCause(cause:Throwable):Throwable printStackTrace():void printStackTrace(s:PrintStream):void printStackTrace(s:PrintWriter):void setStackTrace(stackTrace:StackTraceElement[]):void toString():String

Quelques erreurs et exceptions “standards”

- Error : indique un problème sérieux
  - VirtualMachineError
    - InternalError
    - OutOfMemoryError
    - StackOverflowError
    - UnknownError
  - ThreadDeath
- RuntimeException : exception qui n'a pas besoin d'être capturée
  - ArithmeticException
  - ClassCastException
  - IllegalArgumentException
  - IndexOutOfBoundsException
  - NegativeArraySizeException
  - NullPointerException
- Exception :
  - IOException
    - FileNotFoundException
    - SocketException

Résumé

- Structures de données
- Mécanisme de gestion des exceptions
- La classe **Exception**
- Définir un type d'exception
- Mot-clé **throw**
- Utilisations de **try**, **catch** et **finally**
- Mot-clé **throws**
- Les classes **RuntimeException**, **Throwable** et **Error**
- Les exceptions de Java

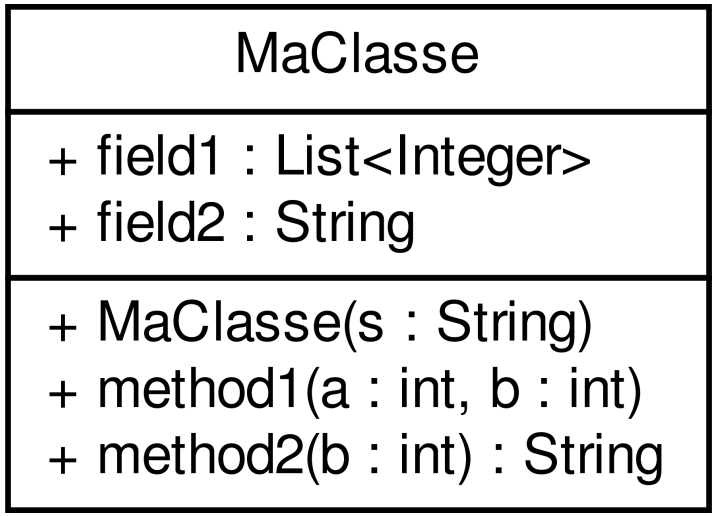
UML

- **UML** = Unified Modeling Language.
- **UML** est un langage de modélisation graphique.
- **UML** est apparu dans le cadre de la “conception orientée objet”.
- **UML** propose 13 types de diagrammes qui permettent la modélisation d’un projet durant tout son cycle de vie.
- Nous allons nous intéresser aux **diagrammes de classes**.

Diagramme de classes

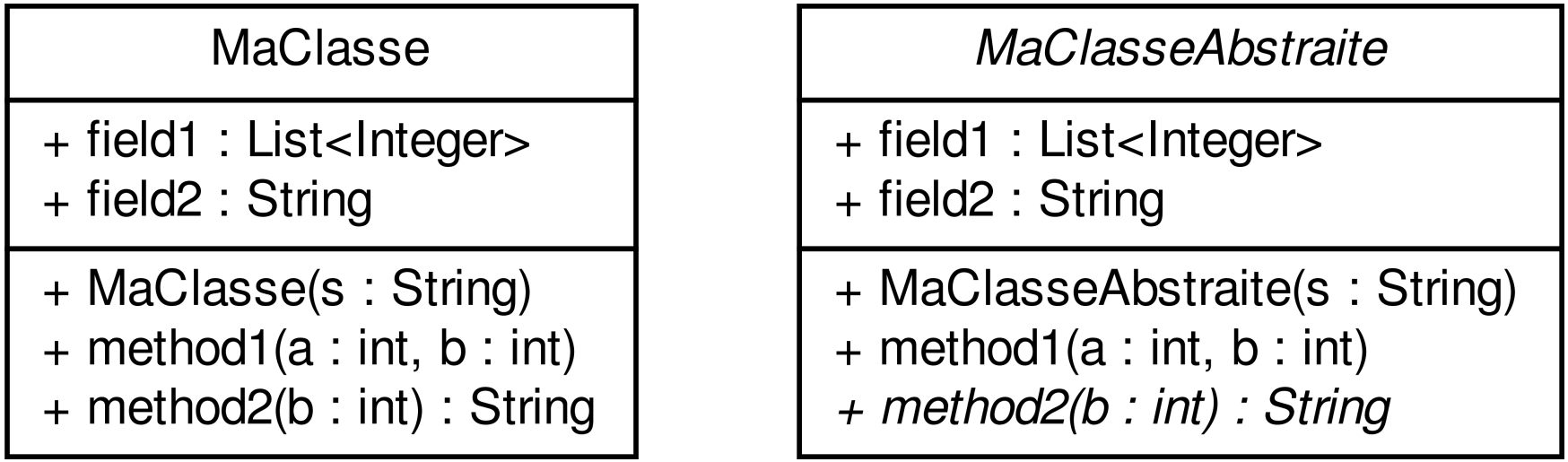
- Schéma utilisé pour représenter les classes et les interfaces.
- On représente également les intéractions entre les classes.
- Il est statique : on fait abstraction des aspects temporels.
- On peut écrire notre programme à partir du diagramme de classes.
- Il permet de réfléchir à la structure du programme.
- Il permet de décrire la structure du programme.

Schéma d'une classe



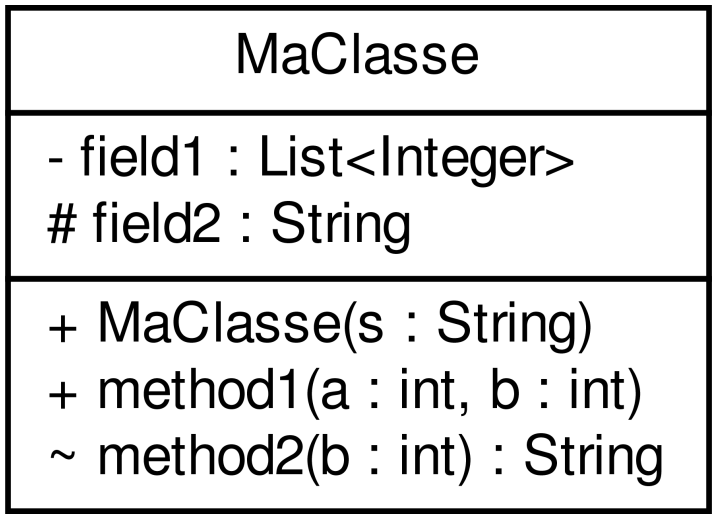
- **Champ** → nom ' :' type
- **Méthode** → nom '(' arguments ')' ' :' type
- **Arguments** → argument (',' argument) *
- **Argument** → nom ' :' type

Classes et méthodes abstraites



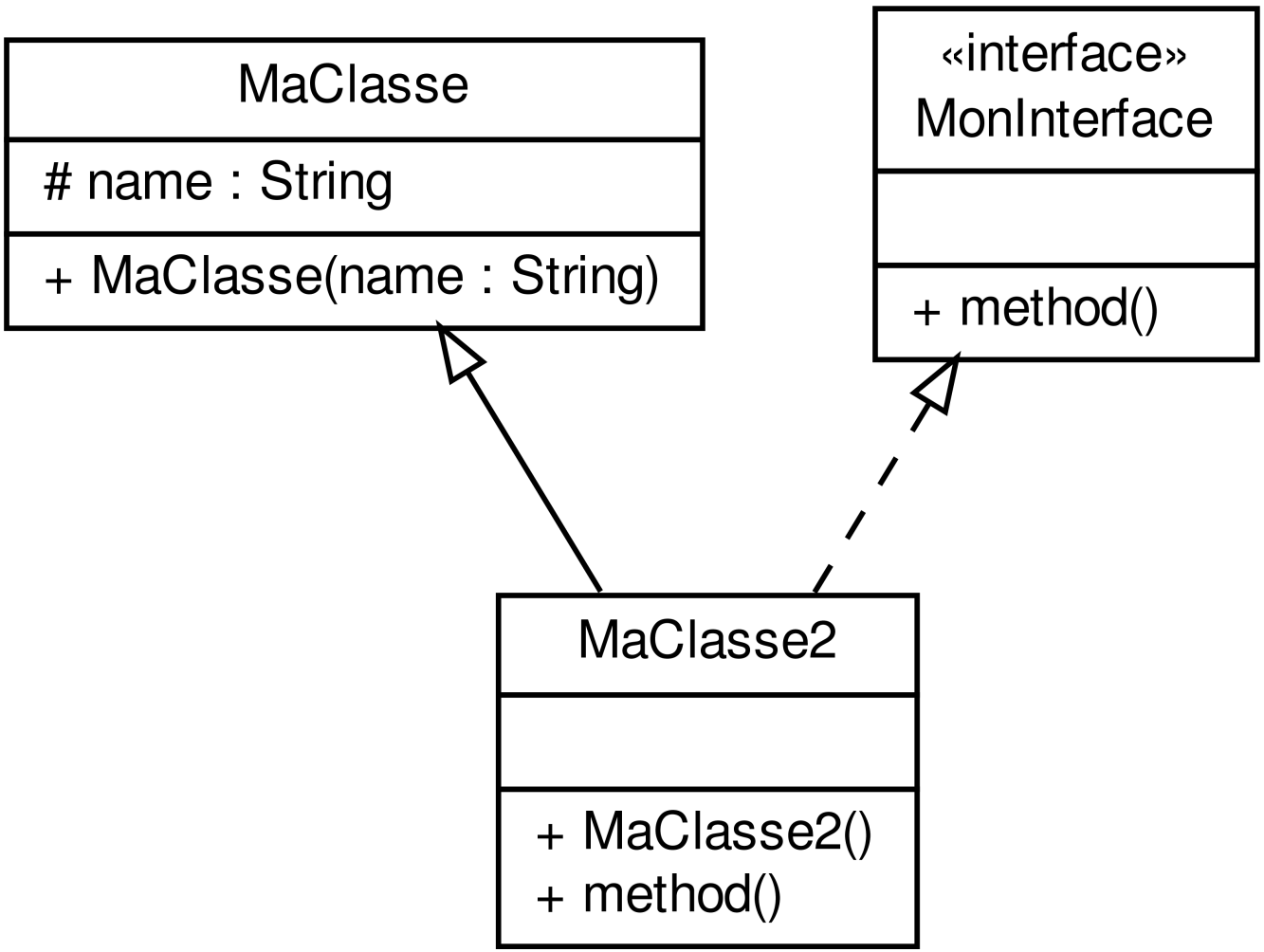
- A droite, la méthode **method2** est abstraite.
- Donc, à droite, la classe est abstraite.

Visibilité

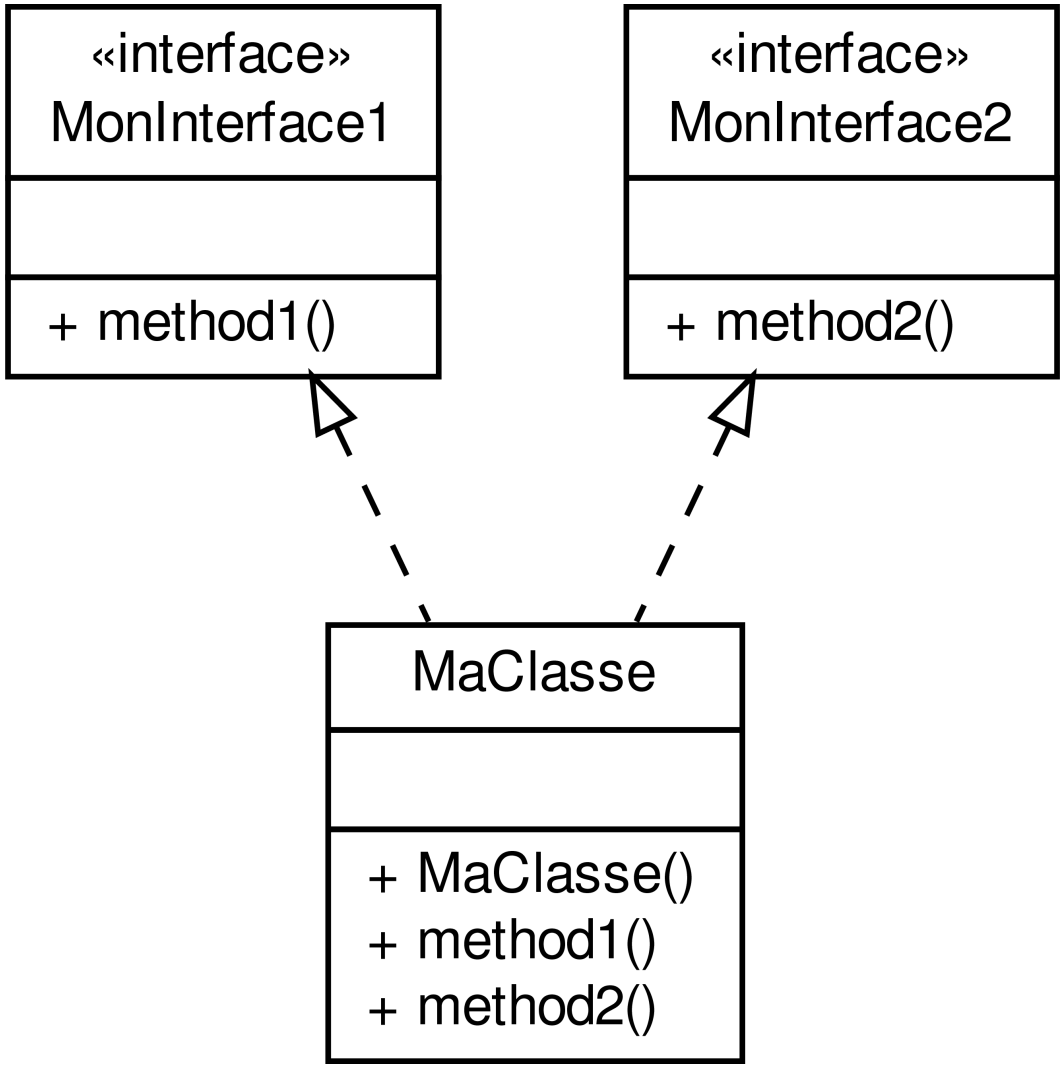


-	:	privé (*private*)
#	:	protégé (*protected*)
+	:	publique (*public*)
~	:	non-publique (*default*)

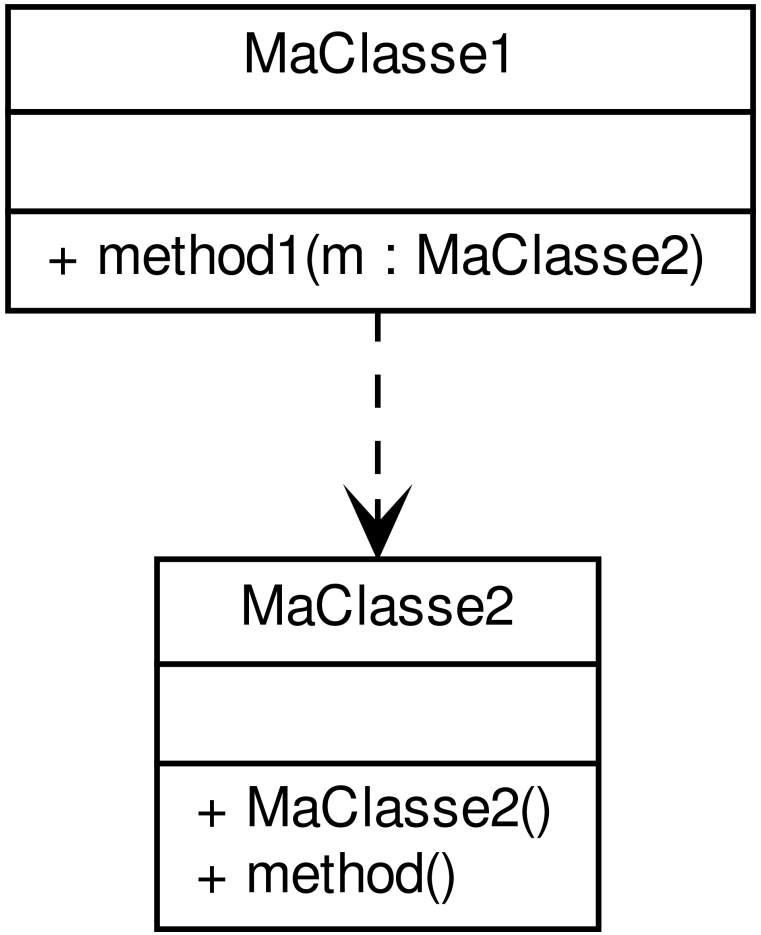
Extensions et implémentations



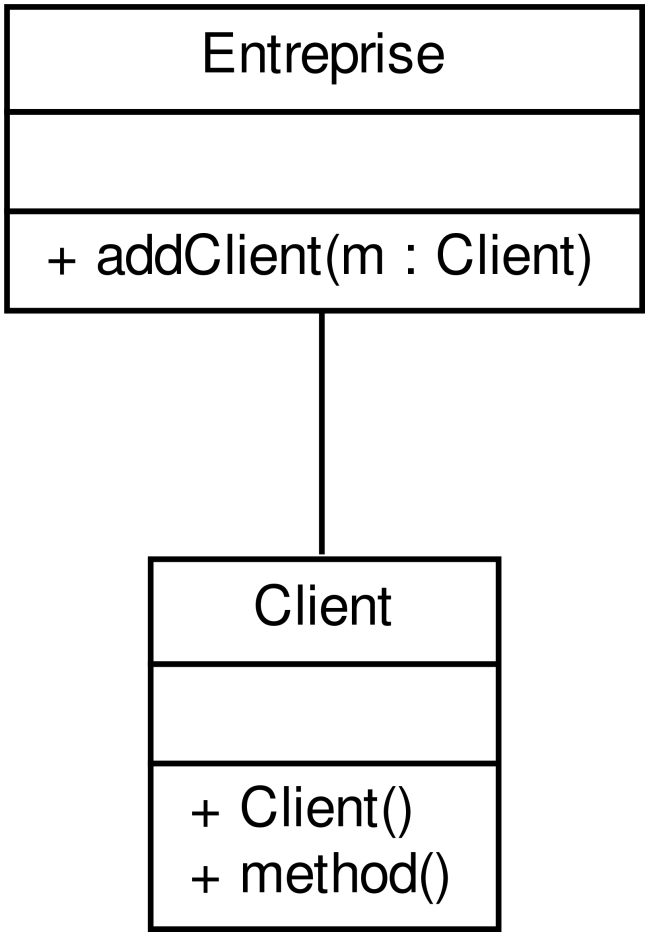
Implémentation multiples



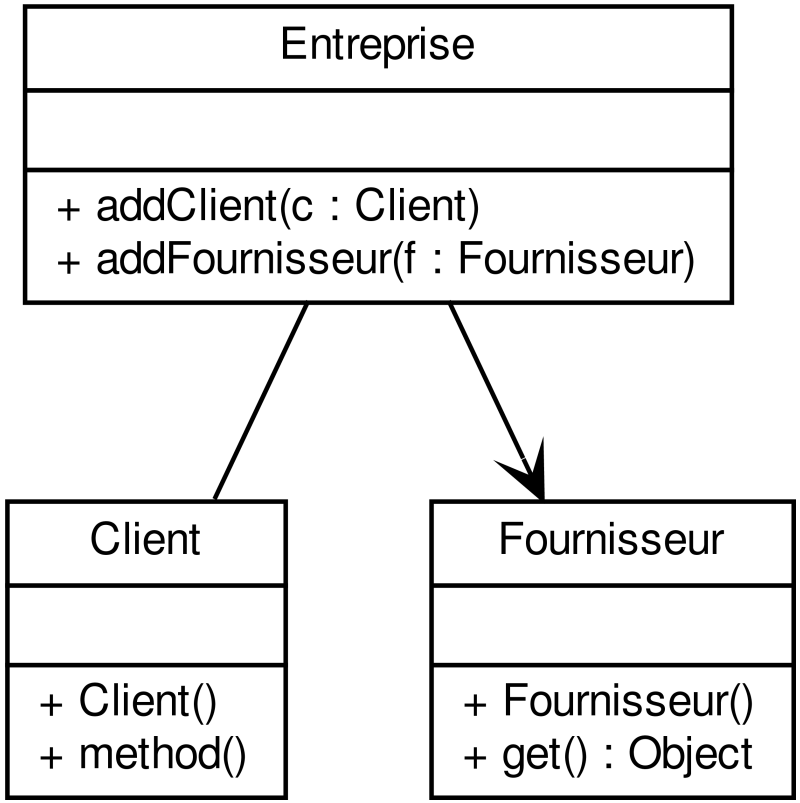
Dépendances



Associations

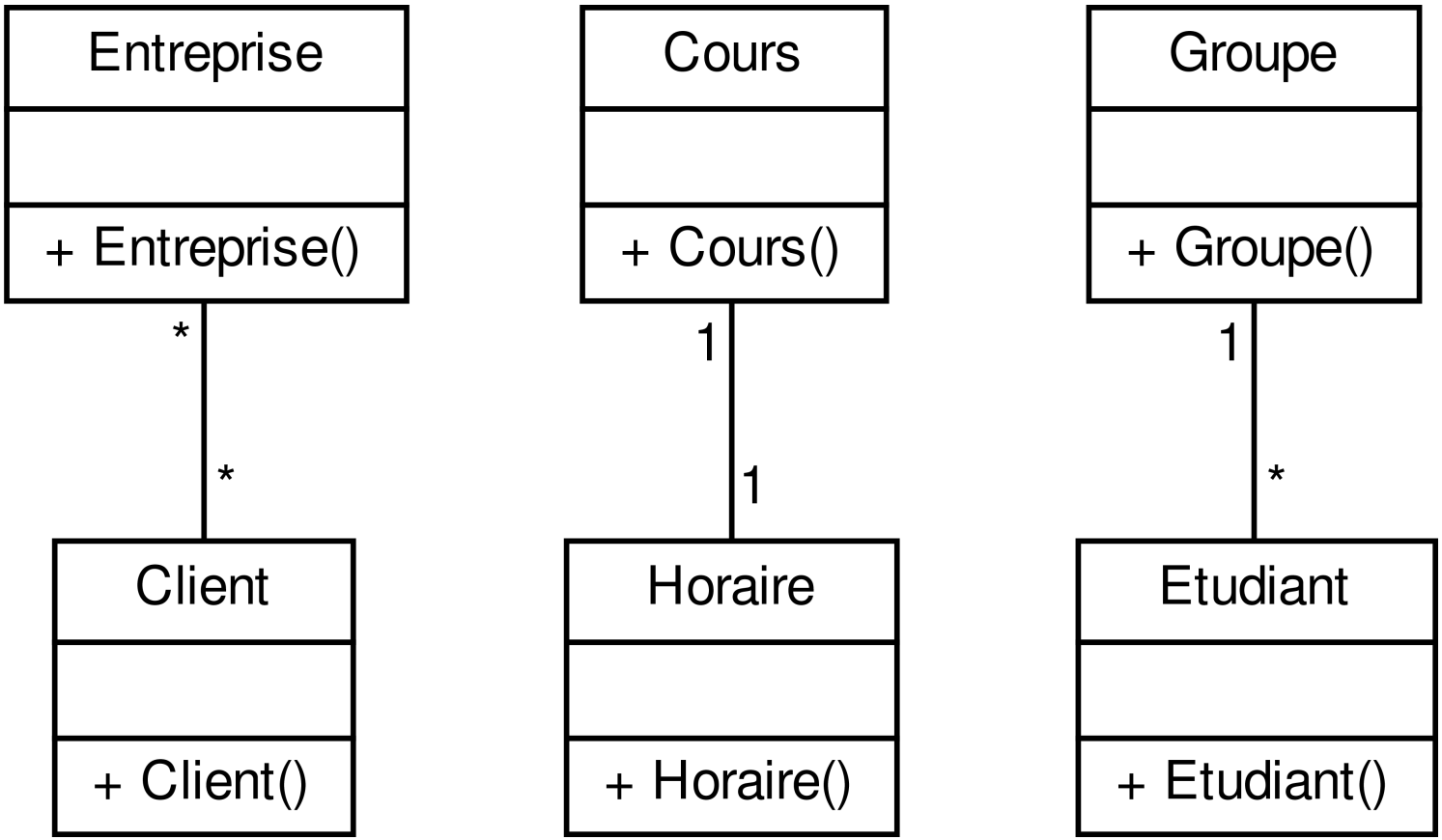


Navigation



- *Entreprise* utilise des méthodes de *Fournisseur*
- *Fournisseur* ne connait pas (et n'utilise pas) *Entreprise*
- Sans flèche, une association est navigable dans les deux sens

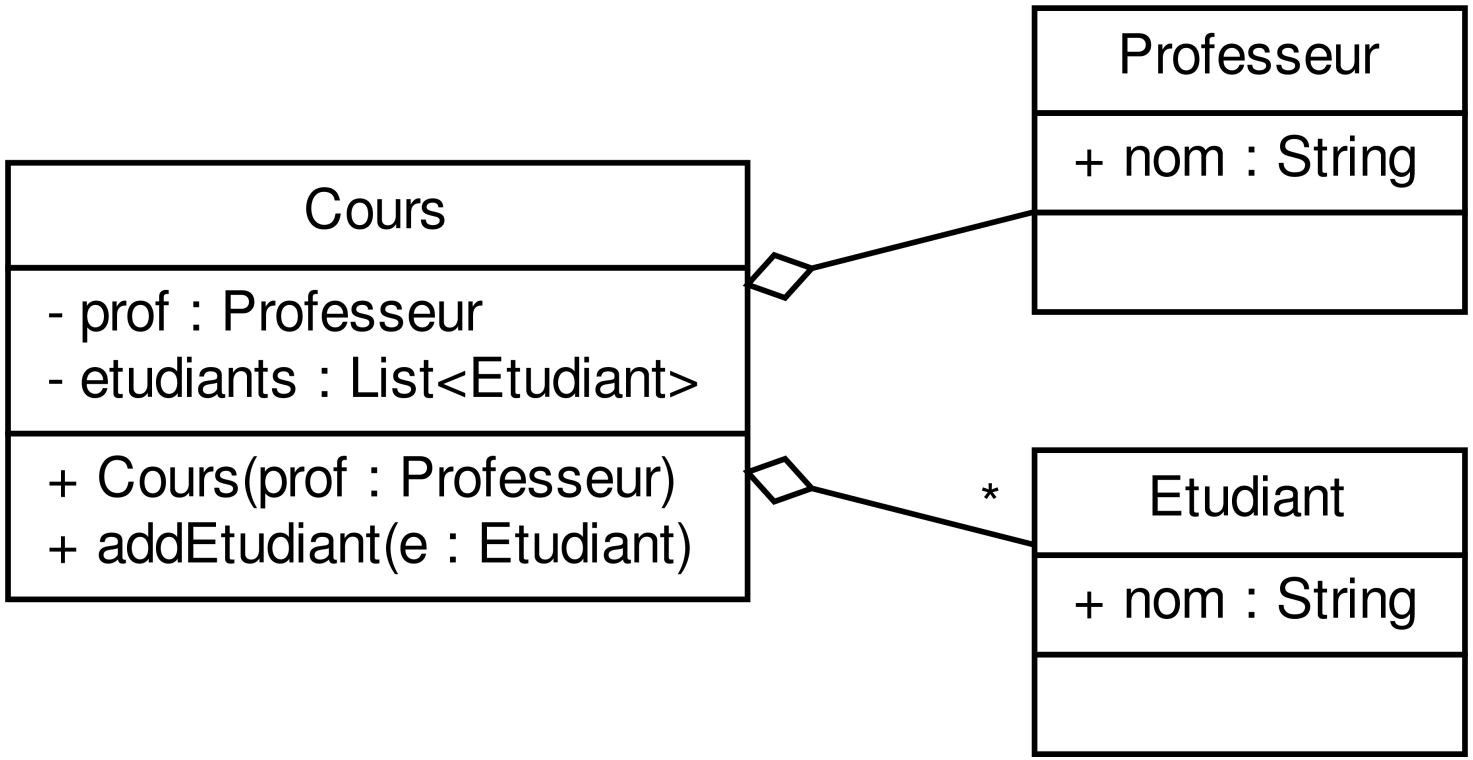
Multiplicité



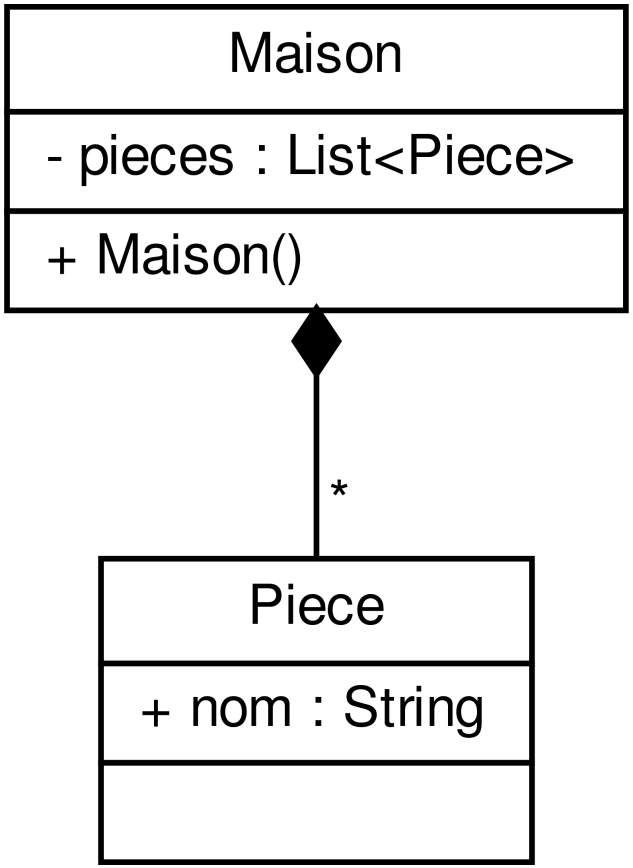
Multiplicité

Notation	**Signification**
0..1	Zéro ou un
1	un uniquement
0..* (ou *)	Zéro ou plus
1..*	Un ou plus
n	Seulement *n*
0..*n*	Zéro à *n*
1..n	Un à *n*

Agrégations

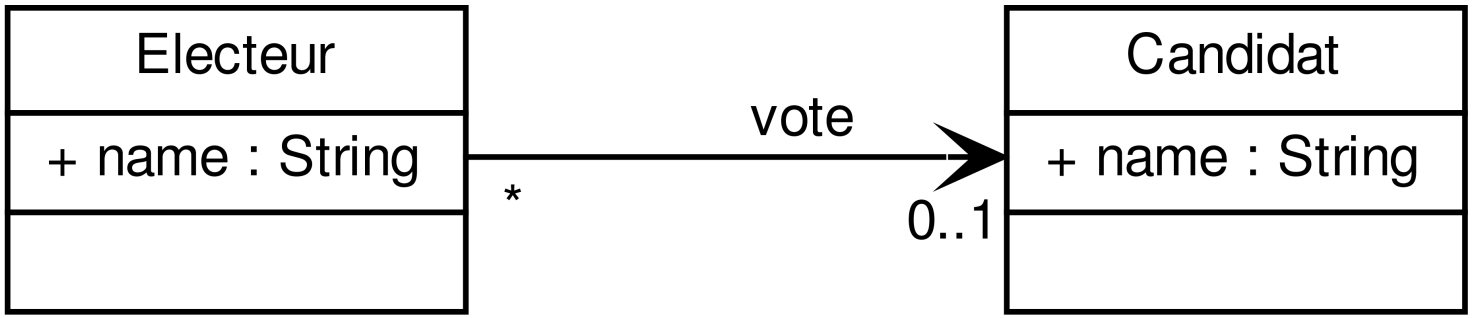


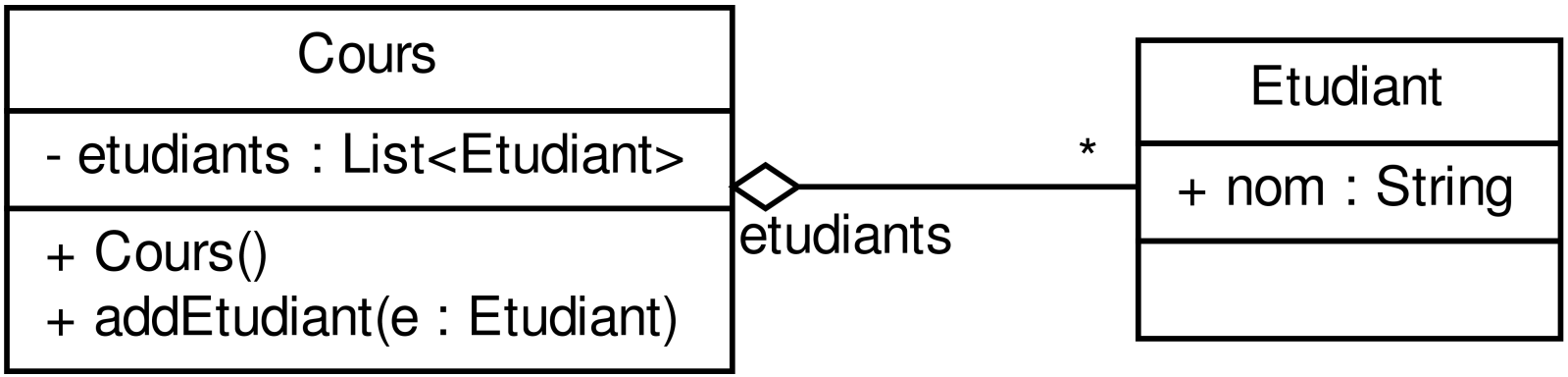
Compositions



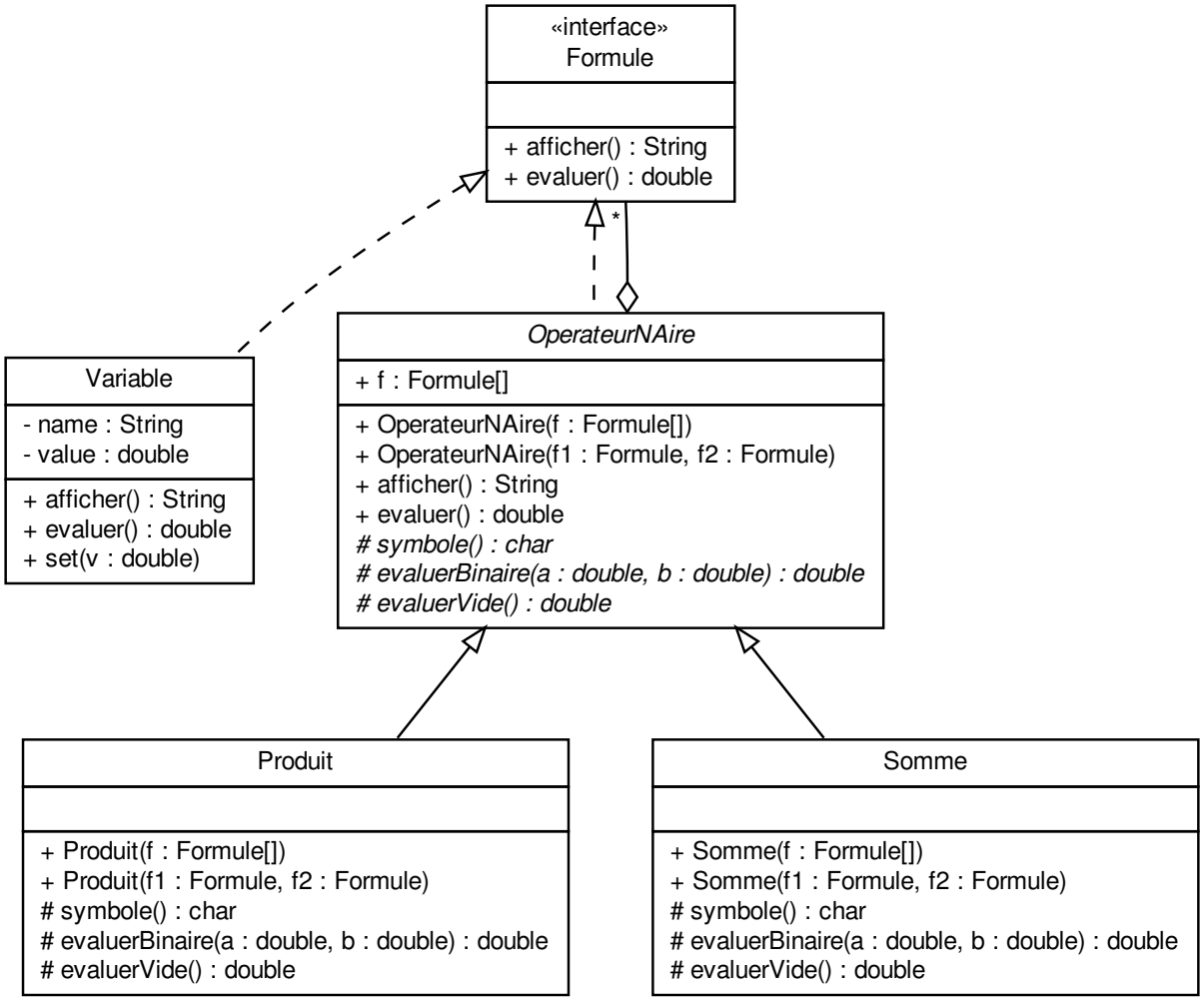
- Les pièces composent la maison.
- Les pièces n'existent que si la maison existe.

Nommage des relations

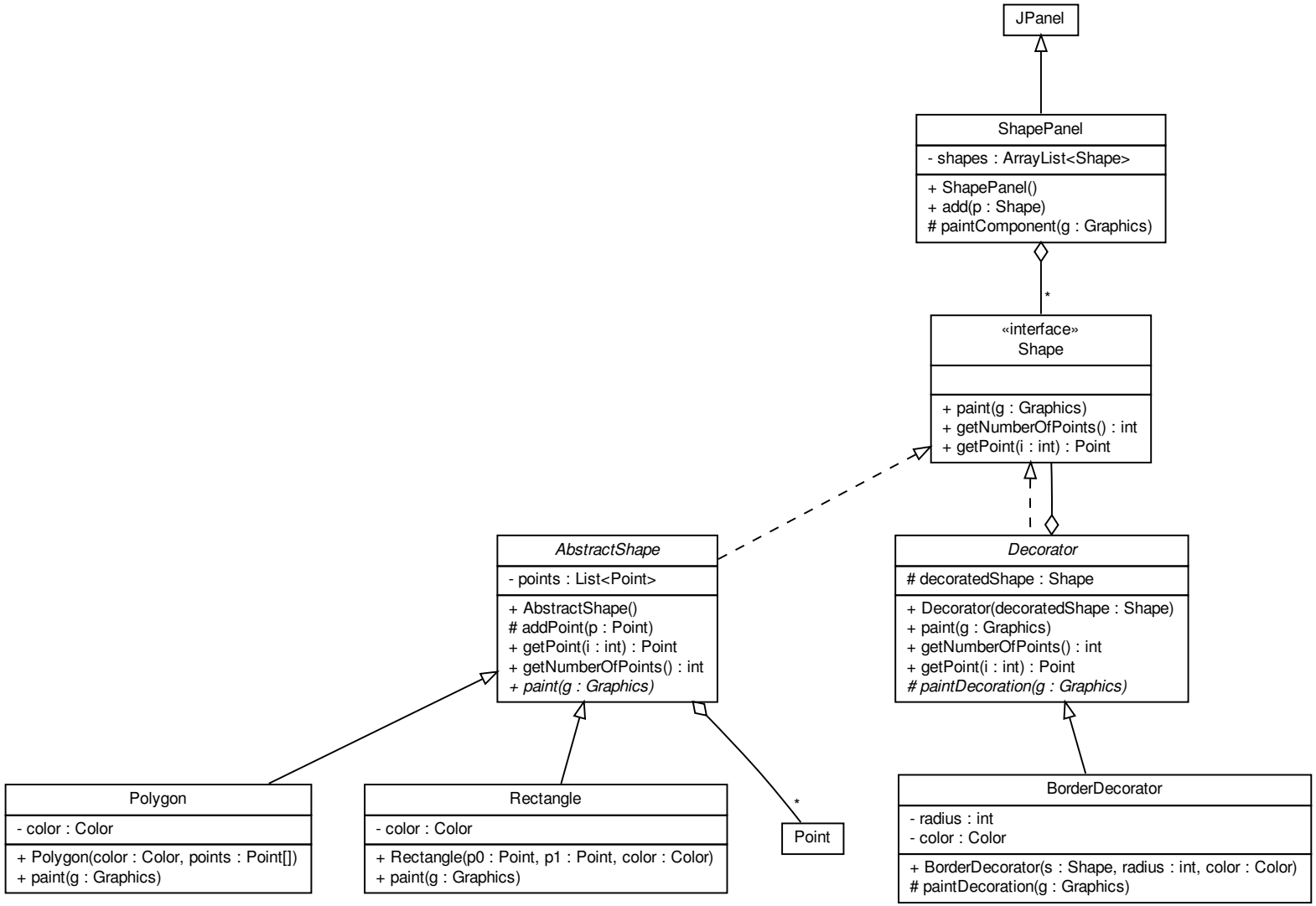




Exemple de diagramme de classes



Exemple de diagramme de classes



La programmation orientée objet (POO)

Les objectifs :

- Faciliter le développement et l'évolution des applications ;
- Permettre le travail en équipe ;
- Augmenter la qualité des logiciels (moins de bugs).

Solutions proposées :

- Découpler (séparer) les parties des projets ;
- Limiter (et localiser) les modifications lors des évolutions ;
- Réutiliser facilement du code.

Développement d'une application

Les différentes phases du développement d'une application :

- Définition du cahier des charges
- Conception de l'architecture générale
- Conception détaillée
- Implémentation
- Tests unitaires
- Intégration
- Qualification (vérification de la conformité aux spécifications)
- Mise en production
- Maintenance : correction des bugs + **évolutions**

Les évolutions

Prise en compte d'une évolution :

- Modification du cahier des charges
- Adaptation de l'architecture
- Implémentation des nouvelles fonctionnalités
- Tests unitaires
- Intégration
- Qualification (vérification de la conformité aux spécifications)
- Mise en production

Programme bien conçu

Un programme est **"bien conçu"** s'il permet de :

- Absorber les changements avec un minimum d'effort
- Implémenter les nouvelles fonctionnalités sans toucher aux anciennes
- Modifier les fonctionnalités existantes en modifiant localement le code

Objectifs :

- Limiter les modules impactés
  - $\Rightarrow$ Simplifier les tests unitaires
  - $\Rightarrow$ Rester conforme à la partie des spécifications qui n'ont pas changé
  - $\Rightarrow$ Facilité l'intégration
- Gagner du temps

Remarque : Le développement d'une application est une suite d'évolutions

Les cinq principes (pour créer du code) SOLID

- Single Responsibility Principle (SRP) :
  Une classe ne doit avoir qu'une seule responsabilité
- Open/Closed Principle (OCP) :
  Programme ouvert pour l'extension, fermé à la modification
- Liskov Substitution Principle (LSP) :
  Les sous-types doivent être substituables par leurs types de base
- Interface Segregation Principle (ISP) :
  Éviter les interfaces qui contiennent beaucoup de méthodes
- Dependency Inversion Principle (DIP) :
  Les modules d'un programme doivent être indépendants
  Les modules doivent dépendre d'abstractions

Single Responsibility Principle (SRP)

Principe :

Une classe ne doit avoir qu'une seule responsabilité (Robert C. Martin).

Signification et objectifs :

- Une responsabilité est une **"raison de changer"**
- Une classe ne doit avoir qu'une seule raison de changer

Pourquoi ?

- Si une classe a plusieurs responsabilités, elles sont couplées
- Dans ce cas, la modification d'une des responsabilités nécessite de :
  - tester à nouveau l'implémentation des autres responsabilités
  - modifier potentiellement les autres responsabilités
  - déployer à nouveau les autres responsabilités
- Donc, vous risquez de :
  - introduire des bugs
  - rendre moins locales les modifications

Single Responsibility Principle (SRP)

Principe :

Une classe ne doit avoir qu'une seule responsabilité (Robert C. Martin).

Signification et objectifs :

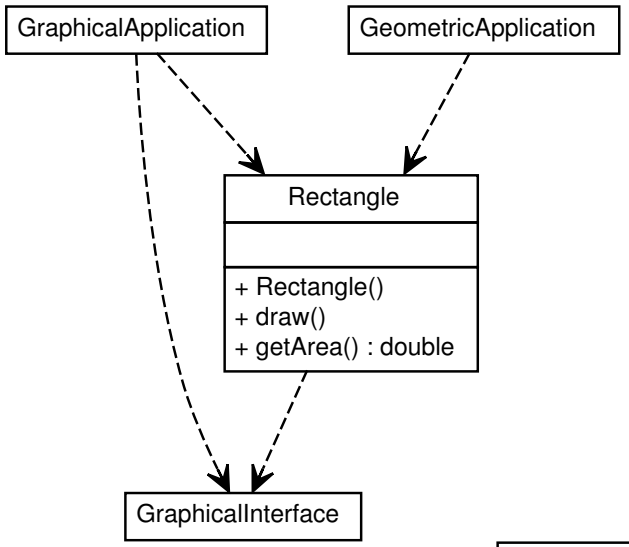
- Une responsabilité est une **"raison de changer"**
- Une classe ne doit avoir qu'une seule raison de changer

Avantages :

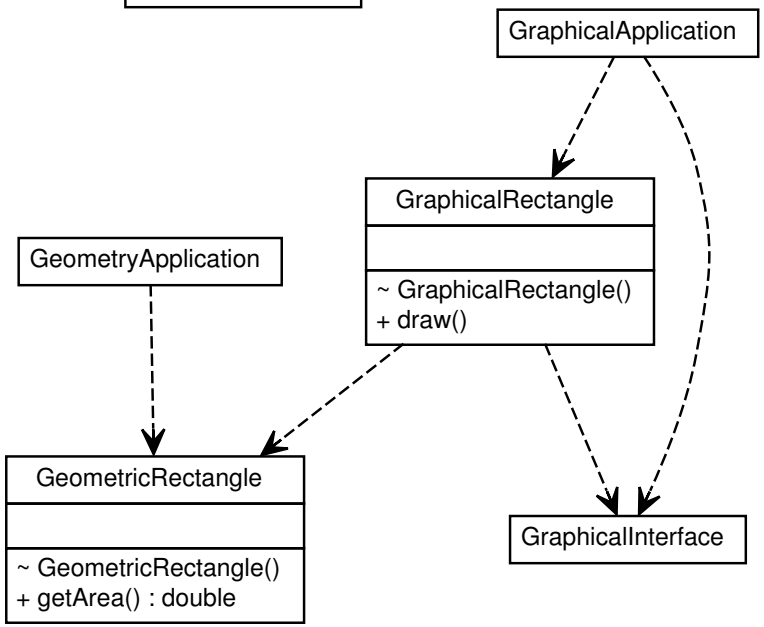
- Diminution de la complexité du code
- Amélioration de la lisibilité du code
- Meilleure organisation du code
- Modification locale lors des évolutions
- Augmentation de la fiabilité
- Les classes ont plus de chance d'être réutilisables

Single Responsibility Principle (SRP)

Violation de SRP :



Séparation des responsabilités :



Open/Closed Principle (OCP)

Principe :

Programme ouvert pour l'extension, fermé à la modification

Signification :

Vous devez pouvoir ajouter une nouvelle fonctionnalité :

- en ajoutant des classes (Ouvert pour l'extension)
- sans modifier le code existant (fermé à la modification)

Avantages :

- Le code existant n'est pas modifié $\Rightarrow$ augmentation de la fiabilité
- Les classes ont plus de chance d'être réutilisables
- Simplification de l'ajout de nouvelles fonctionnalités

Open/Closed Principle (OCP)

```
class Rectangle { public Point p1, p2; }
class Circle { public Point c; public int radius; }

class GraphicTools {
    static void drawRectangle(Rectangle r) { ... }
    static void drawCircle(Circle c) { ... }

    static void drawShapes(Object[] shapes) {
        for (Object o : shapes) {
            if (o instanceof Rectangle) {
                Rectangle r = (Rectangle)o;
                drawRectangle(r);
            }
            else if (o instanceof Circle) {
                Circle c = (Circle)o;
                drawCircle(c);
            }
        }
    }
}
```

Open/Closed Principle (OCP)

Solution : Abstraction et interfaces !

```
interface Shape { public void draw(); }

class Rectangle implements Shape {
        public Point p1, p2;
        void draw() { ... }
}

class Circle implements Shape {
        public Point c; public int radius;
        void draw() { ... }
}

class GraphicTools {
    static void drawShapes(Shape[] shapes) {
        for (Shape s : shapes)
            s.draw();
    }
}
```

Liskov Substitution Principle (LSP)

Principe :

Les sous-types doivent être substituables par leurs types de base

Signification :

Si une classe **A** étend une classe **B** (ou implémente une interface **B**) alors un programme **P** écrit pour manipuler des instances de type **B** doit avoir le même comportement s'il manipule des instances de la classe **A**.

Avantages :

- Diminution de la complexité du code
- Amélioration de la lisibilité du code
- Meilleure organisation du code
- Modification locale lors des évolutions
- Augmentation de la fiabilité
- Les classes ont plus de chance d'être réutilisables

Liskov Substitution Principle (LSP)

Une classe qui permet de représenter un rectangle géométrique :

```
public class Rectangle {
    private double w;
    private double h;

    public void setWidth(double w) { this.w = w; }
    public void setHeight(double h) { this.h = h; }
    public double getWidth() { return w; }
    public double getHeight() { return h; }
    public double getArea() { return w*h; }
}
```

Liskov Substitution Principle (LSP)

Un carré **est** un rectangle donc on devrait pouvoir écrire :

```
public class Square extends Rectangle {

    public void setWidth(double w) {
        super.setWidth(w);
        super.setHeight(w);
    }

    public void setHeight(double h) {
        super.setWidth(h);
        super.setHeight(h);
    }

}
```

Liskov Substitution Principle (LSP)

Violation de LSP :

```
public void test(Rectangle r) {
        r.setWidth(2);
        r.setHeight(3);
        if (r.getArea()!=3*2)
            System.out.println("bizarre␣!");
}
```

La mauvaise question :

Un carré **est**-il un rectangle ?

La bonne question :

Pour les utilisateurs,
votre carré **a**-t-il le même **comportement** que **votre** rectangle ?

La réponse :

Dans ce cas, **NON**

Liskov Substitution Principle (LSP)

Une solution :

```
public abstract class RectangularShape {
    public abstract double getWidth();
    public abstract double getHeight();
    public double getArea() { return getWidth()*getHeight(); }
}

public class Rectangle extends RectangularShape {
    private double w;
    private double h;

    public void setWidth(double w) { this.w = w; }
    public void setHeight(double h) { this.h = h; }
    public double getWidth() { return w; }
    public double getHeight() { return h; }
}
```

Liskov Substitution Principle (LSP)

Suite de la solution :

```
class Square extends RectangularShape {
    private double s;
    public void setSideLength(double s) { this.s = s; }
    public double getWidth() { return s; }
    public double getHeight() { return s; }

}
```

Utilisation :

```
public void testRectangle(Rectangle r) {
        r.setWidth(2); r.setHeight(3);
        if (r.getArea()!=3*2) System.out.println("jamais !");
}

public void testSquare(Square s) {
        s.setSideLength(2);
        if (s.getArea()!=2*2) System.out.println("jamais !");
}
```

Plan du cours

- Diagrammes de classes UML
- Principes SOLID :
  - Single Responsibility Principle (SRP)
  - Open/Closed Principle (OCP)
  - Liskov Substitution Principle (LSP)
  - Interface Segregation Principle (ISP)
  - Dependency Inversion Principle (DIP)
- Patrons de conception

Les cinq principes (pour créer du code) SOLID

- Single Responsibility Principle (SRP) :
  Une classe ne doit avoir qu'une seule responsabilité
- Open/Closed Principle (OCP) :
  Programme ouvert pour l'extension, fermé à la modification
- Liskov Substitution Principle (LSP) :
  Les sous-types doivent être substituables par leurs types de base
- Interface Segregation Principle (ISP) :
  Éviter les interfaces qui contiennent beaucoup de méthodes
- Dependency Inversion Principle (DIP) :
  Les modules d'un programme doivent être indépendants
  Les modules doivent dépendre d'abstractions

Interface Segregation Principle (ISP)

Principe :

Éviter les interfaces qui contiennent beaucoup de méthodes

Signification et objectifs :

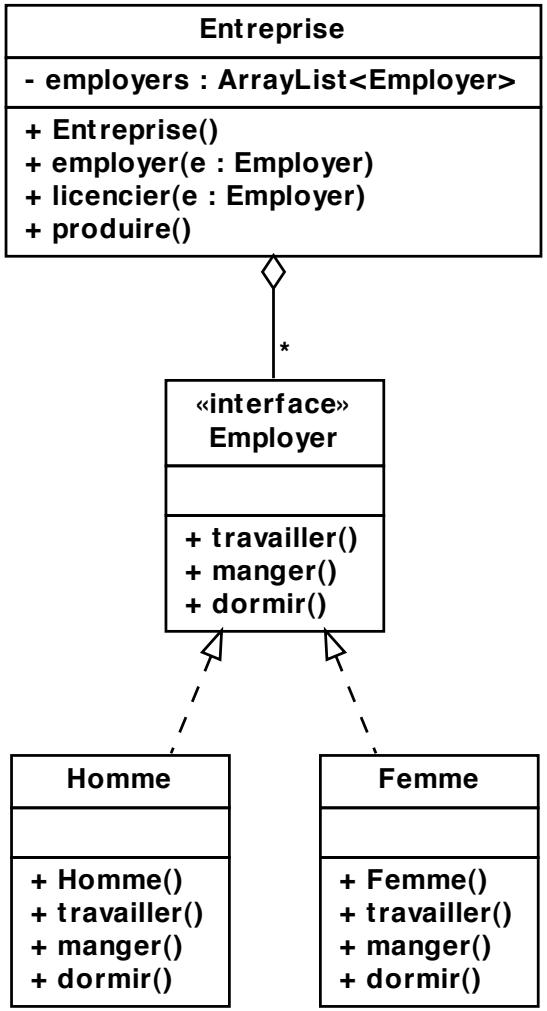
- Découper les interfaces en responsabilités distinctes (SRP)
- Quand une interface grossit, se poser la question du rôle de l'inferface
- Éviter de devoir implémenter des services qui n'ont pas à être proposés par la classe qui implémente l'interface
- Limiter les modifications lors de la modification de l'interface

Avantages :

- Le code existant est moins modifié $\Rightarrow$ augmentation de la fiabilité
- Les classes ont plus de chance d'être réutilisables
- Simplification de l'ajout de nouvelles fonctionnalités

Interface Segregation Principle (ISP)

Exemple de violation de ISP :



Interface Segregation Principle (ISP)

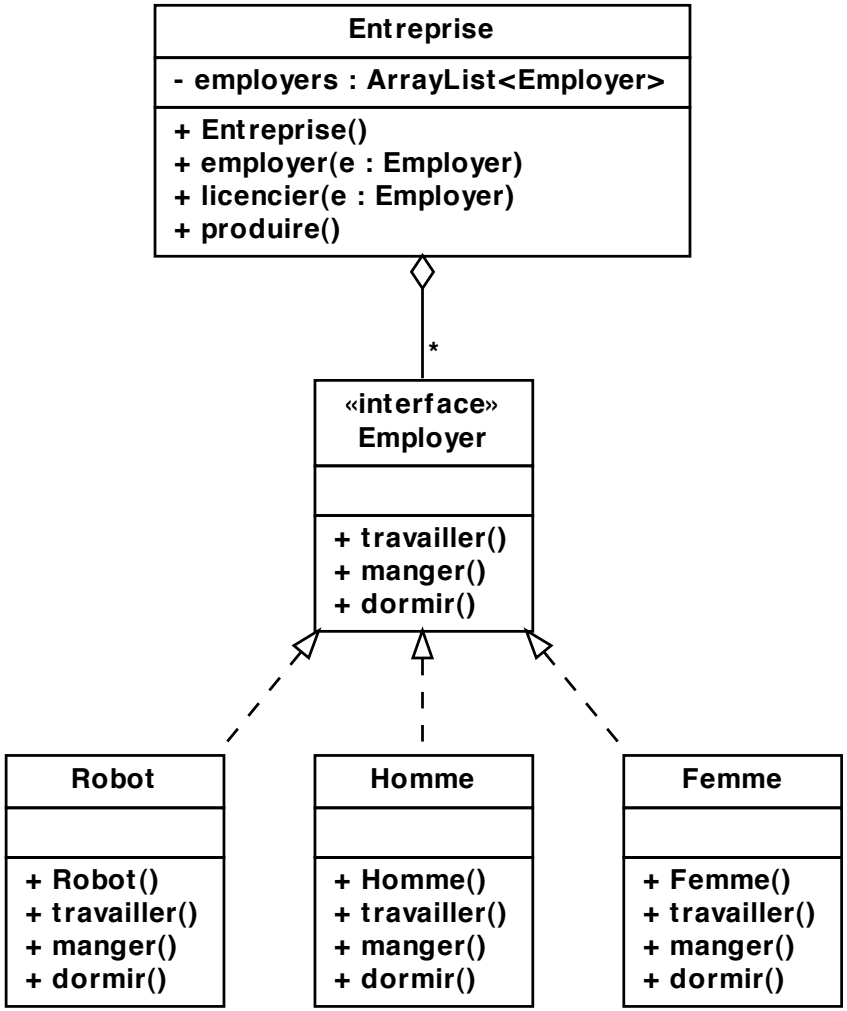
```
public interface Employer {
    void travailler();
    void manger();
    void dormir();
}

public class Femme implements Employer {
    public void travailler() {
        System.out.println("Je suis une femme et je travaille");
    }
    public void manger() {
        System.out.println("Je suis une femme et je mange");
    }
    public void dormir() {
        System.out.println("Je suis une femme et je dors");
    }
}
```

Interface Segregation Principle (ISP)

```
public class Entreprise {
    private final ArrayList<Employer> employers;

    public Entreprise() {
        employers = new ArrayList<Employer>();
    }

    public void employer(Employer e) {
        employers.add(e);
    }

    public void licencier(Employer e) {
        employers.remove(e);
    }

    public void produire() {
        for (Employer e : employers)
            e.travailler();
    }
}
```

Interface Segregation Principle (ISP)

Un robot peut travailler :



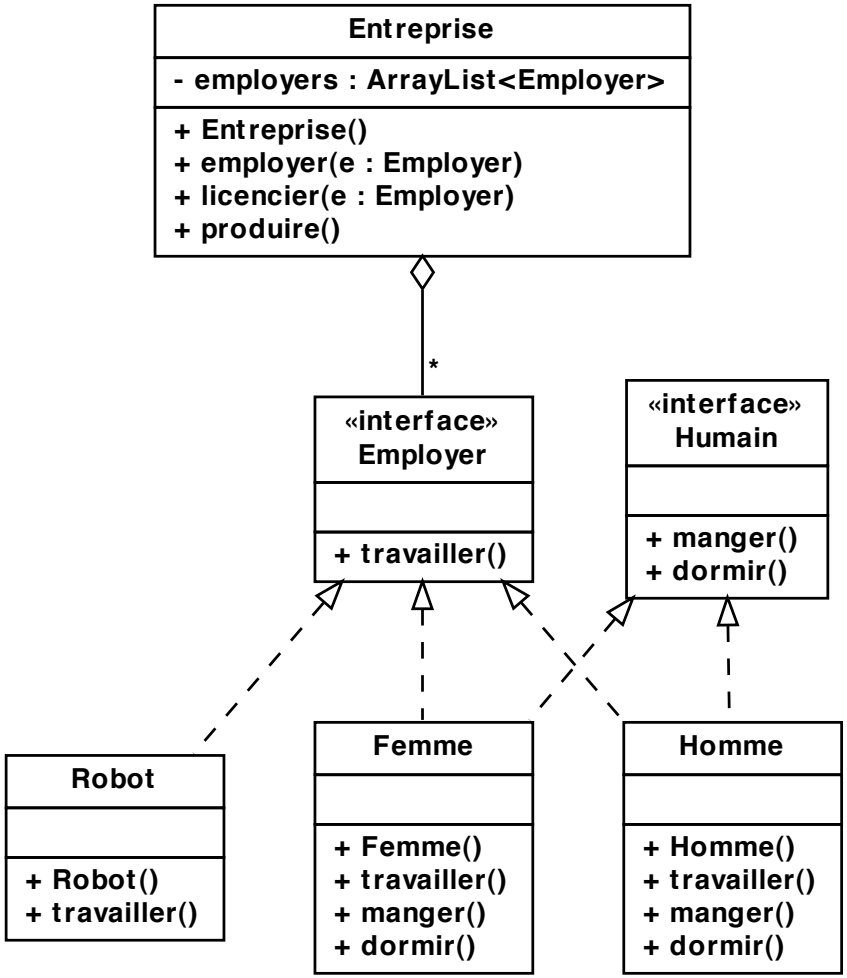
Interface Segregation Principle (ISP)

Problème : un robot est un travailleur qui ne sait pas dormir et manger

```
public class Robot implements Employer {
    public void travailler() {
        System.out.println("je travaille");
    }

    public void manger() {
        System.out.println("je ne sais pas le faire");
    }

    public void dormir() {
        System.out.println("je ne sais pas le faire");
    }
}
```

Interface Segregation Principle (ISP)

Solution : découper l'interface *Travailleur* en deux



Dependency Inversion Principle (DIP)

Principe :

Les modules d'un programme doivent être indépendants
Les modules doivent dépendre d'abstractions

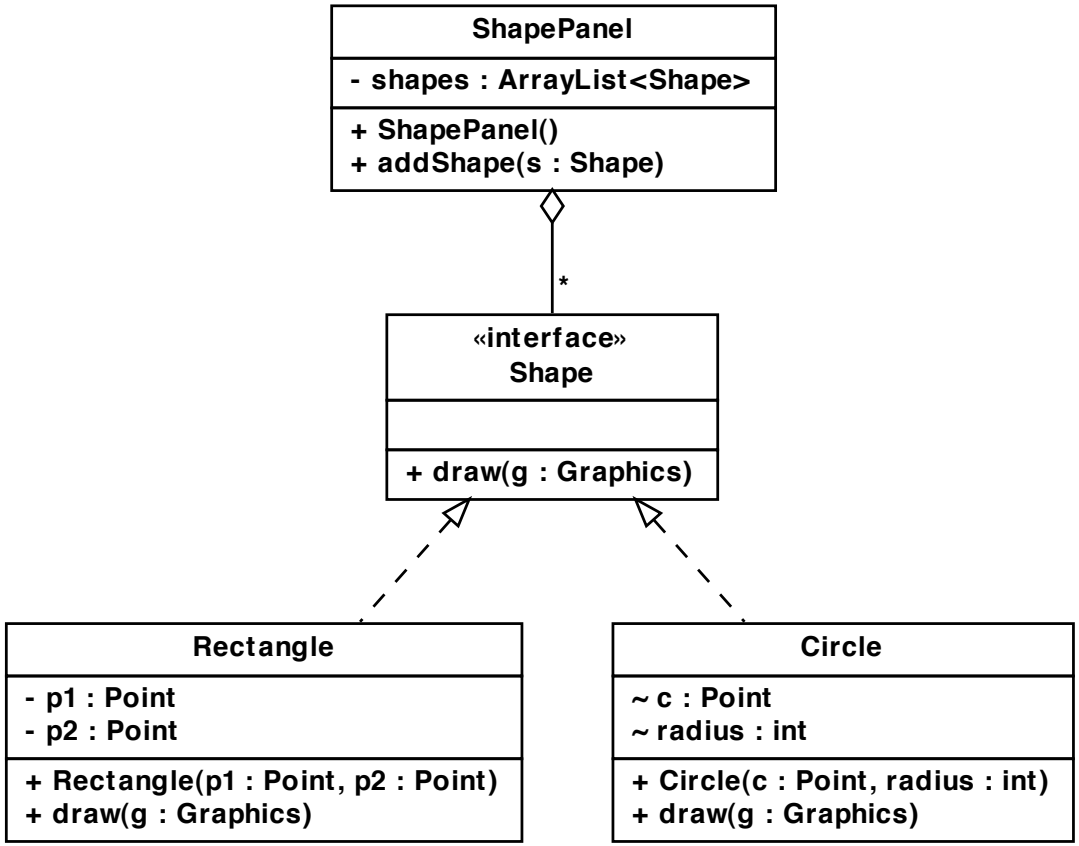
Signification et objectifs :

- Découpler les différents modules de votre programme
- Les lier en utilisant des interfaces
- Décrire correctement le comportement de chaque module
- Permet de remplacer un module par un autre module plus facilement

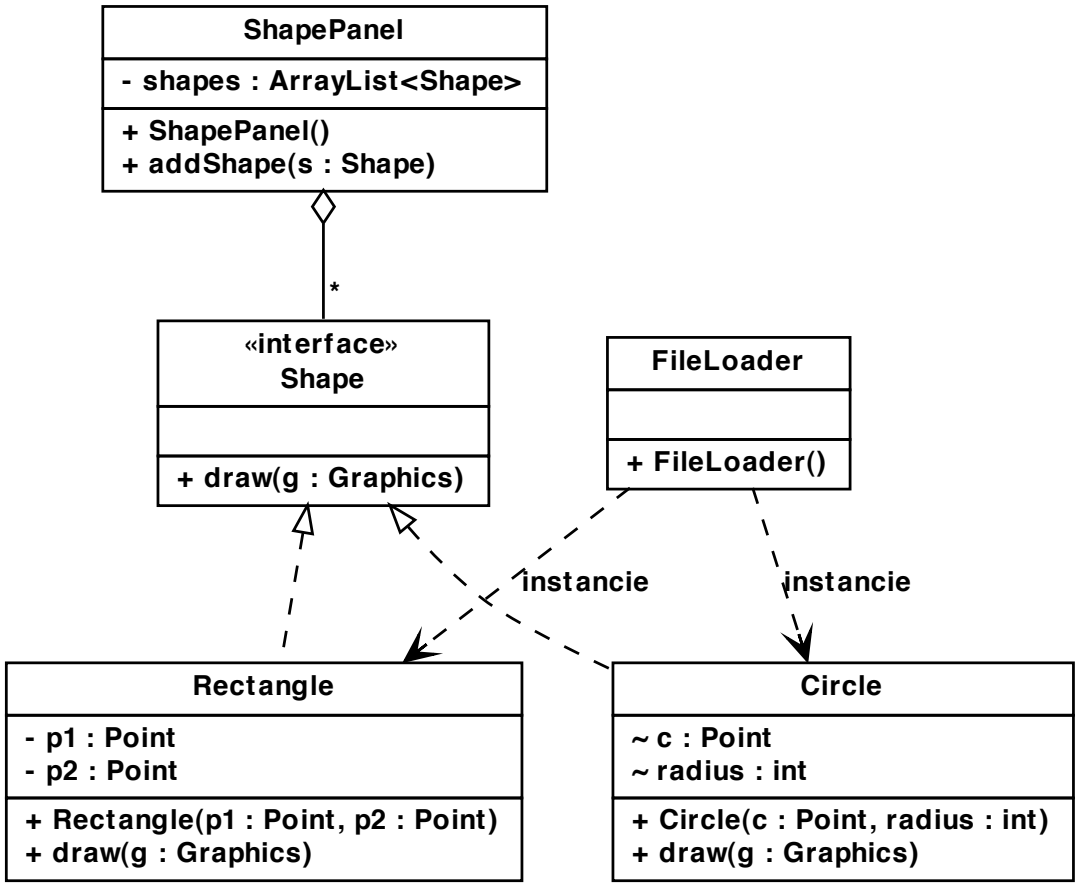
Avantages :

- Les modules sont plus facilement réutilisables
- Simplification de l'ajout de nouvelles fonctionnalités
- L'intégration est rendue plus facile

Dependency Inversion Principle (DIP)



Dependency Inversion Principle (DIP)



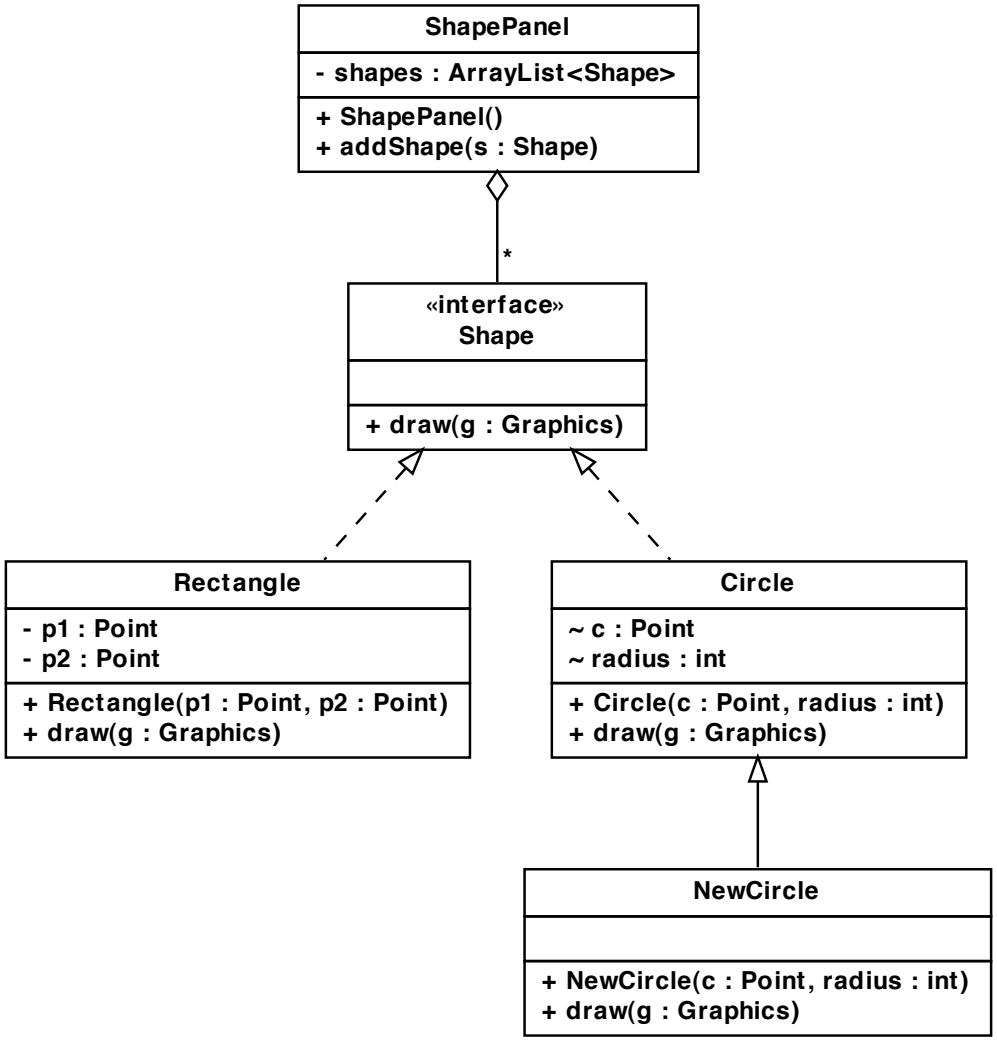
Dependency Inversion Principle (DIP)

```
public class FileLoader {

    public void loadCircle(ShapePanel p, Scanner s) {
        int x = s.nextInt();
        int y = s.nextInt();
        int r = s.nextInt();
        p.addShape(new Circle(new Point(x,y), r));
    }

    public void loadRectangle(ShapePanel p, Scanner s) {
        int x = s.nextInt();
        int y = s.nextInt();
        int x2 = s.nextInt();
        int y2 = s.nextInt();
        p.addShape(new Rectangle(new Point(x,y),
                                 new Point(x2, y2)));
    }

    ...
```

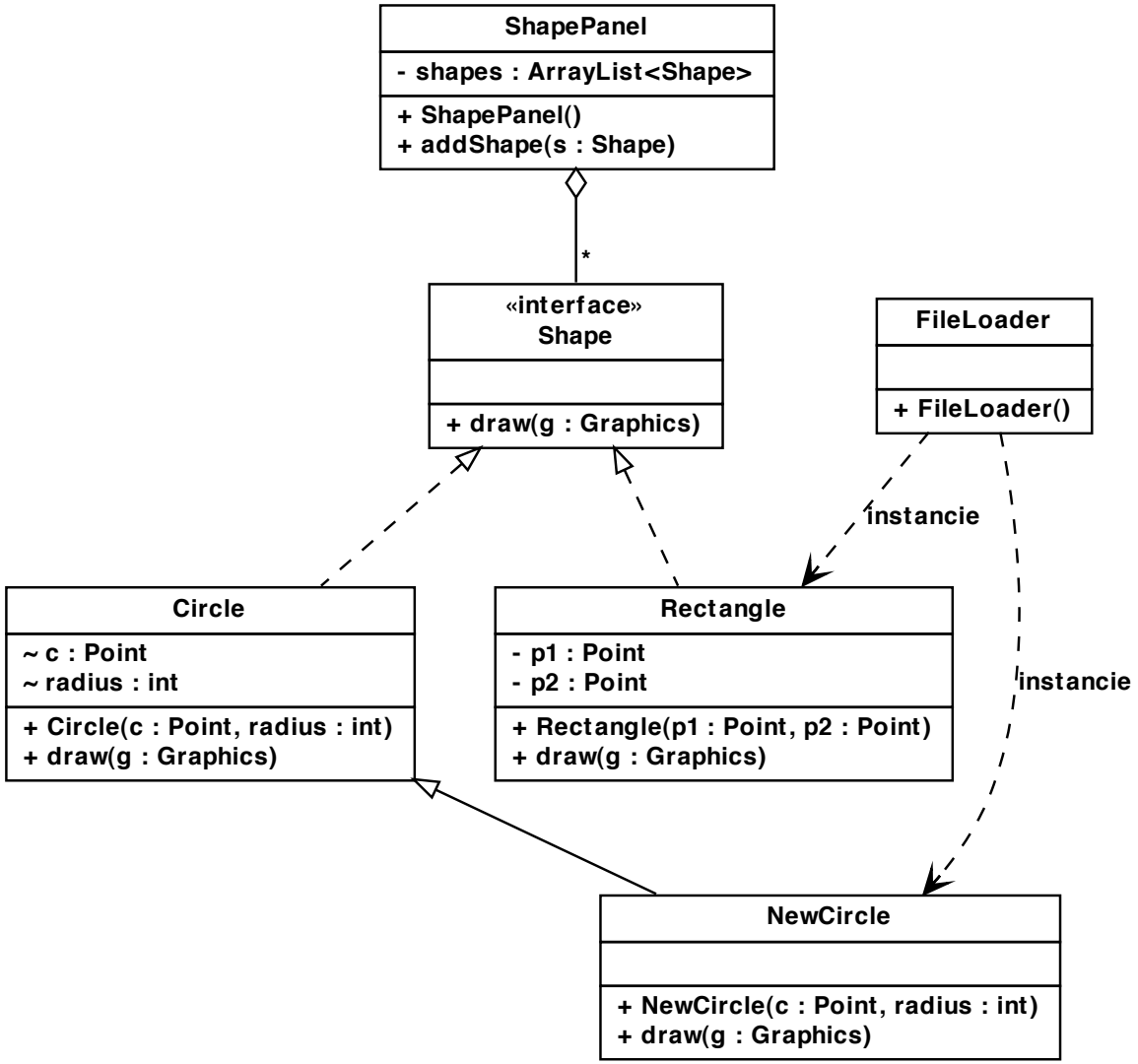
Dependency Inversion Principle (DIP)

```
    public void loadFile(ShapePanel p, String name) {
        Scanner s = new Scanner(name);
        while (s.hasNext()) {
            switch (s.nextInt()) {
                case 0:
                    loadCircle(p, s);
                case 1:
                    loadRectangle(p, s);
            }
        }
        s.nextInt();
    }

}
```

Dependency Inversion Principle (DIP)



Dependency Inversion Principle (DIP)

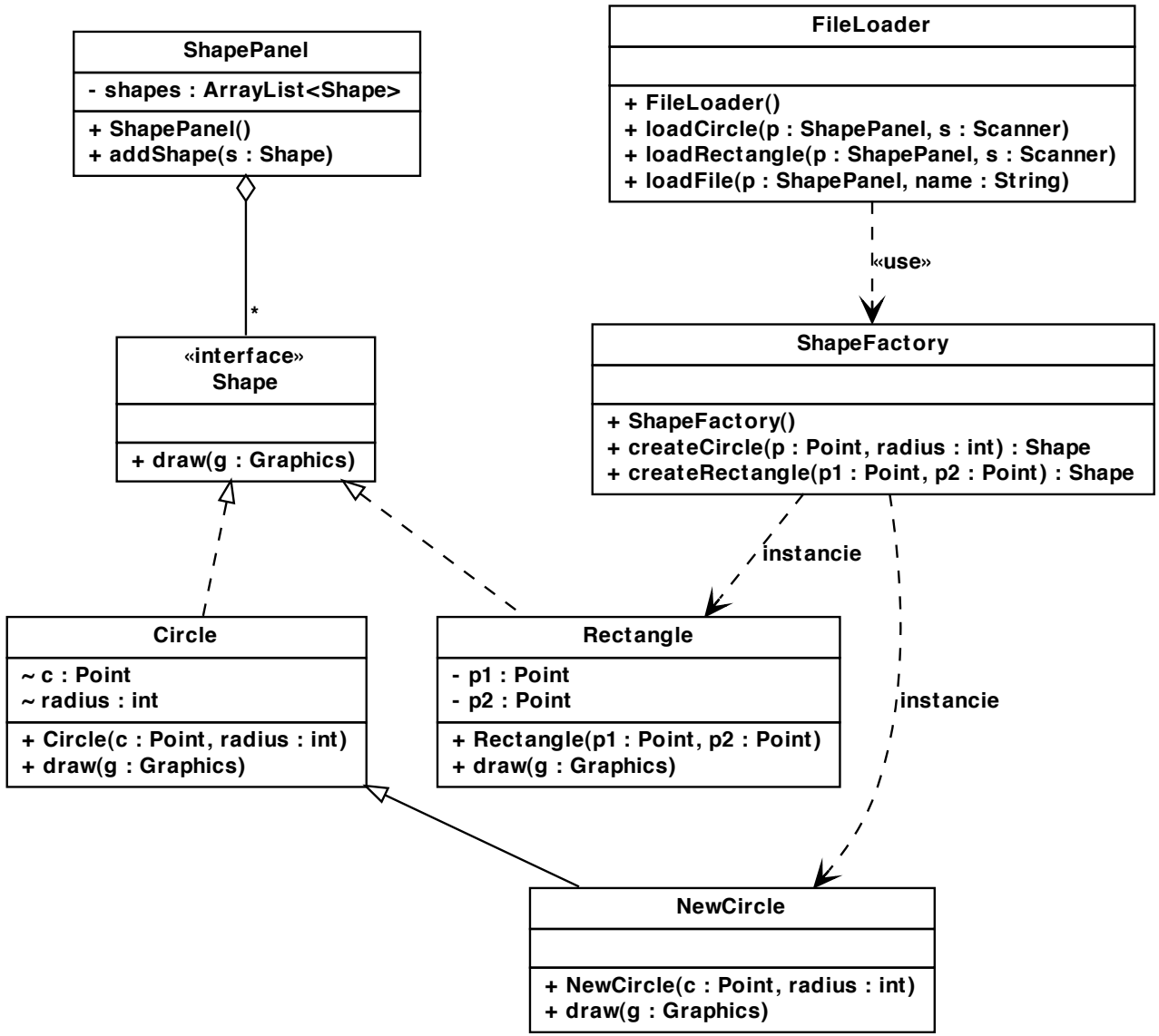


Dependency Inversion Principle (DIP)

Violation de OCP car on viole DIP :

```
public class FileLoader {

    public void loadCircle(ShapePanel p, Scanner s) {
        int x = s.nextInt();
        int y = s.nextInt();
        int r = s.nextInt();
        p.addShape(new NewCircle(new Point(x,y), r));
    }

    public void loadRectangle(ShapePanel p, Scanner s) {
        int x = s.nextInt();
        int y = s.nextInt();
        int x2 = s.nextInt();
        int y2 = s.nextInt();
        p.addShape(new Rectangle(new Point(x,y),
                                 new Point(x2, y2)));
    }
    ...
```

Dependency Inversion Principle (DIP)



Dependency Inversion Principle (DIP)

```
public class ShapeFactory {

    public Shape createCircle(Point p, int radius) {
        return new NewCircle(p, radius);
    }

    public Shape createRectangle(Point p1, Point p2) {
        return new Rectangle(p1, p2);
    }

}
```

Dependency Inversion Principle (DIP)

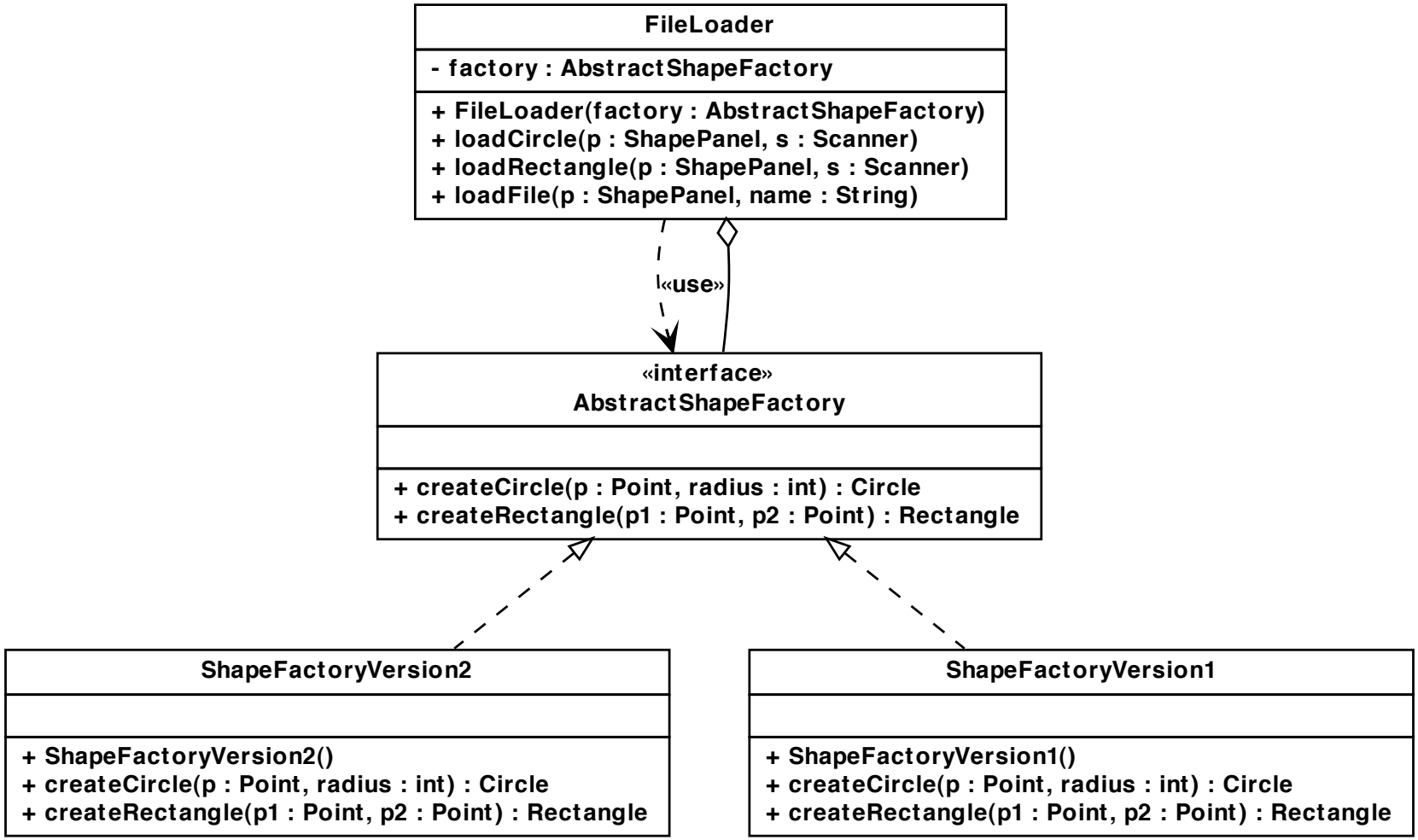
```
public class FileLoader {

    private ShapeFactory factory = new ShapeFactory();

    public void loadCircle(ShapePanel p, Scanner s) {
        int x = s.nextInt();
        int y = s.nextInt();
        int r = s.nextInt();
        p.addShape(factory.createCircle(new Point(x,y), r));
    }

    public void loadRectangle(ShapePanel p, Scanner s) {
        int x = s.nextInt();
        int y = s.nextInt();
        int x2 = s.nextInt();
        int y2 = s.nextInt();
        p.addShape(factory.createRectangle(new Point(x,y),
                                           new Point(x2, y2)));
    }
    ...
```

Dependency Inversion Principle (DIP)

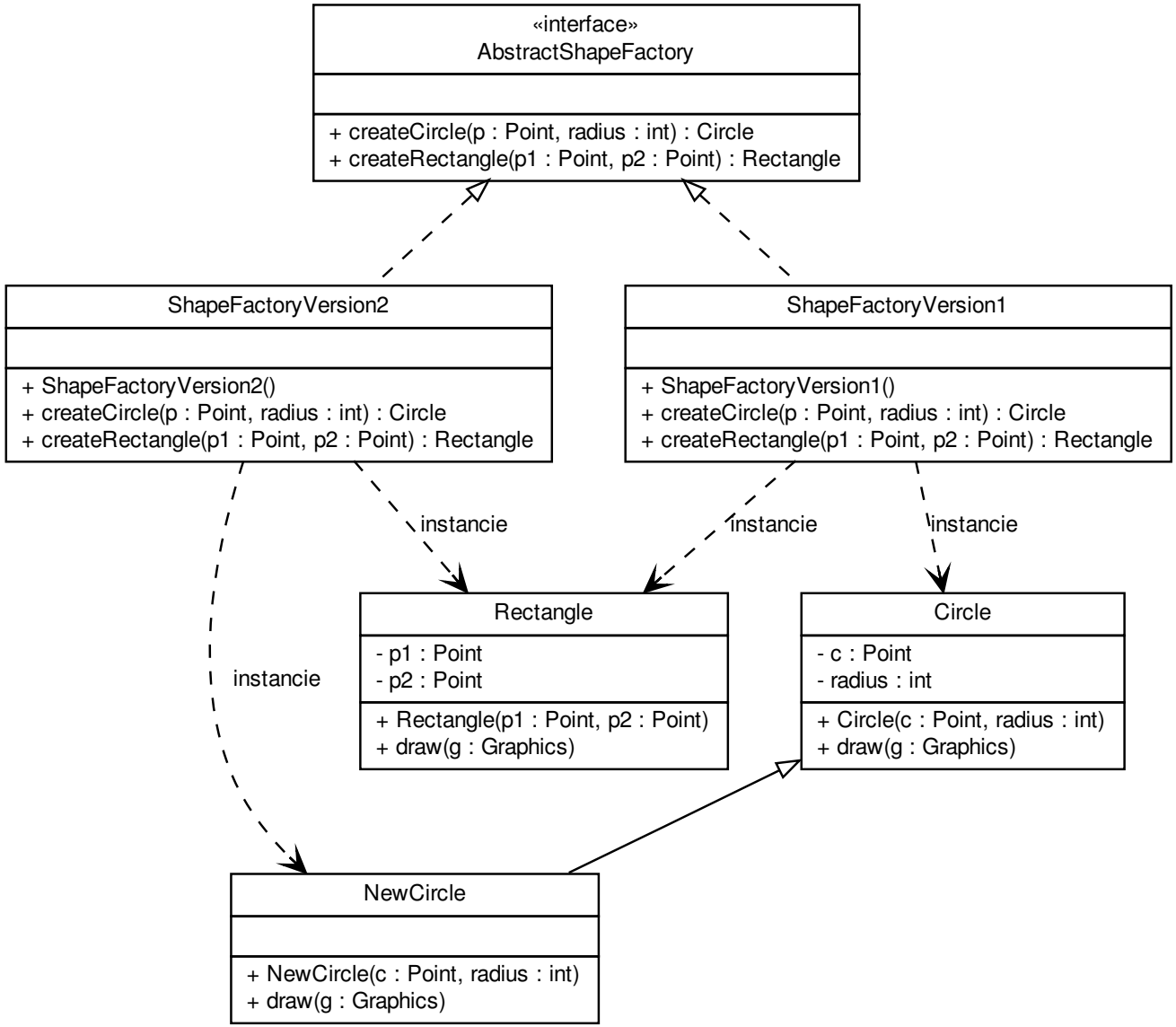
Question : Comment faire cohabiter les deux fabriques suivantes ?

```
public class ShapeFactoryVersion1 {
    public Shape createCircle(Point p, int radius) {
        return new Circle(p, radius);
    }

    public Shape createRectangle(Point p1, Point p2) {
        return new Rectangle(p1, p2);
    }
}

public class ShapeFactoryVersion2 {
    public Shape createCircle(Point p, int radius) {
        return new NewCircle(p, radius);
    }

    public Shape createRectangle(Point p1, Point p2) {
        return new Rectangle(p1, p2);
    }
}
```

Dependency Inversion Principle (DIP)

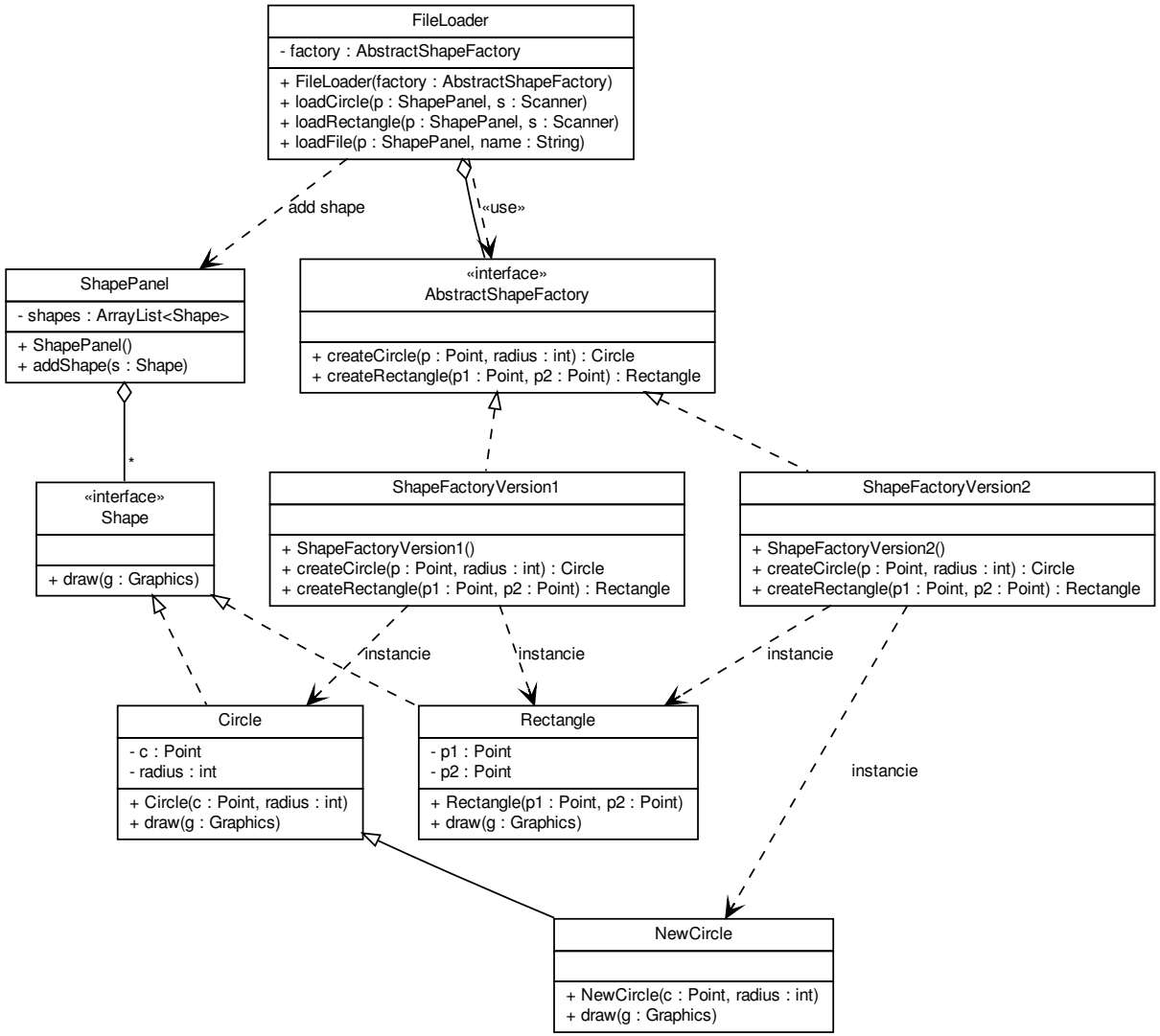


Dependency Inversion Principle (DIP)



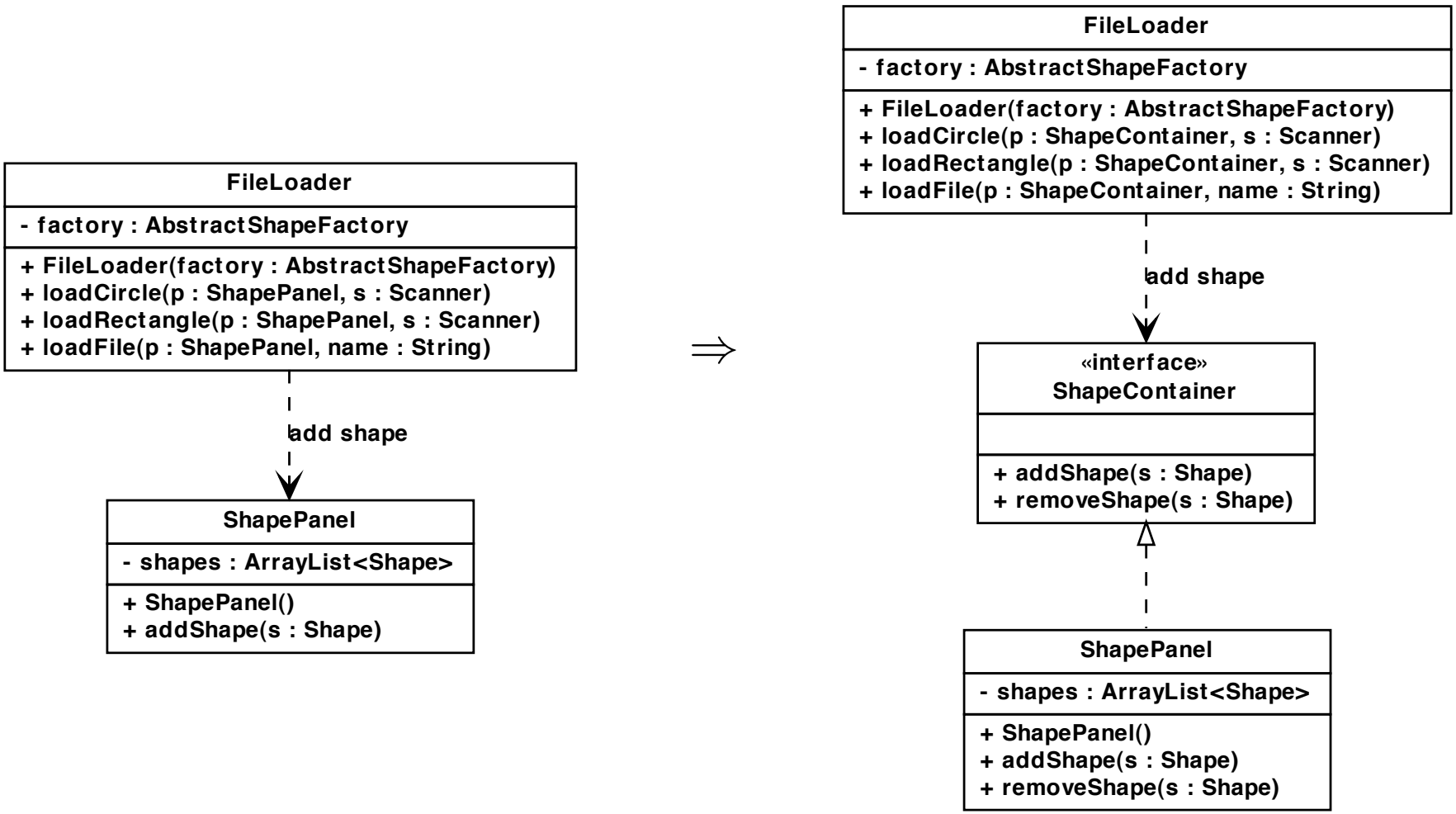
Dependency Inversion Principle (DIP)

Diagramme de classes gobal :



Dependency Inversion Principle (DIP)

Application du DIP entre *FileLoader* et *ShapePanel* :



Dependency Inversion Principle (DIP)

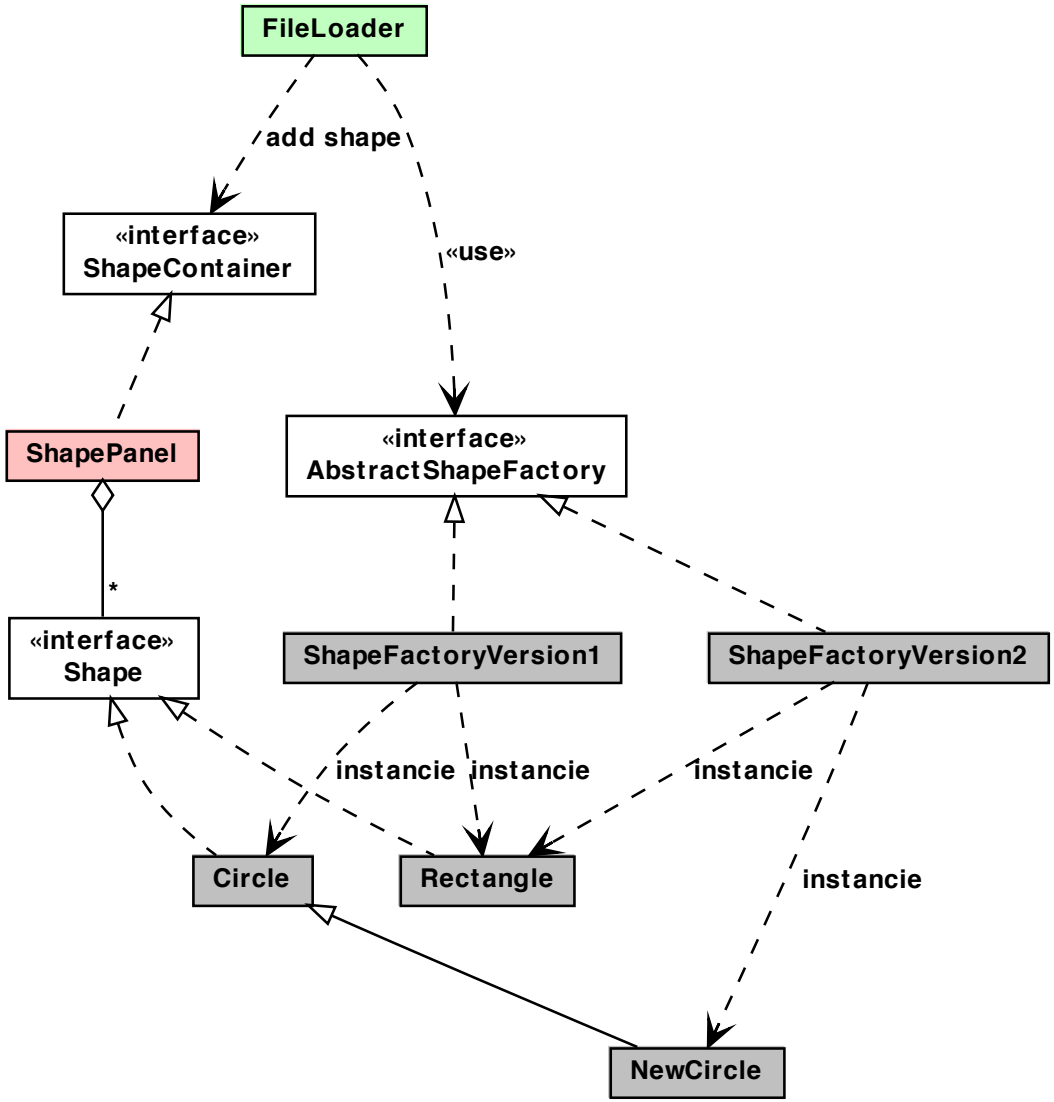
```
public interface ShapeContainer {
    public void addShape(Shape s);
    public void removeShape(Shape s);
}

public class ShapePanel extends JPanel
                        implements ShapeContainer {

    private final ArrayList<Shape> shapes;

    public ShapePanel() {
        shapes = new ArrayList<Shape>();
    }

    public void addShape(Shape s) { shapes.add(s); }

    public void removeShape(Shape s) { shapes.remove(s); }

}
```

Dependency Inversion Principle (DIP)

```
public class FileLoader {
    private final AbstractShapeFactory factory;

    public FileLoader(AbstractShapeFactory factory) {
        this.factory = factory;
    }

    public void loadCircle(ShapeContainer p, Scanner s) {
        int x = s.nextInt(), y = s.nextInt();
        int r = s.nextInt();
        p.addShape(factory.createCircle(new Point(x,y), r));
    }

    public void loadRectangle(ShapeContainer p, Scanner s) {
        int x = s.nextInt(), y = s.nextInt();
        int x2 = s.nextInt(), y2 = s.nextInt();
        p.addShape(factory.createRectangle(new Point(x,y),
                                           new Point(x2, y2)));
    }
    ...
```

Dependency Inversion Principle (DIP)

Diagramme de classes :



Patrons de conception (Design patterns)

- Les **patrons de conception** décrivent des solutions standards pour répondre aux problèmes rencontrés lors de la conception orientée objet
- Ils tendent à respecter les 5 principes SOLID
- Ils sont le plus souvent indépendants du langage de programmation
- Ils ont été formalisés dans le livre du **"Gang of Four"** (Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides – 1995)
- "Les patrons offrent la possibilité de capitaliser un savoir précieux né du savoir-faire d'experts" (Buschmann – 1996)
- Les **anti-patrons** (ou **antipatterns**) sont des erreurs courantes de conception.

Patrons de conception – GoF – Création

- Fabrique abstraite (Abstract Factory)
- Monteur (Builder)
- Fabrique (Factory Method)
- Prototype (Prototype)
- Singleton (Singleton)

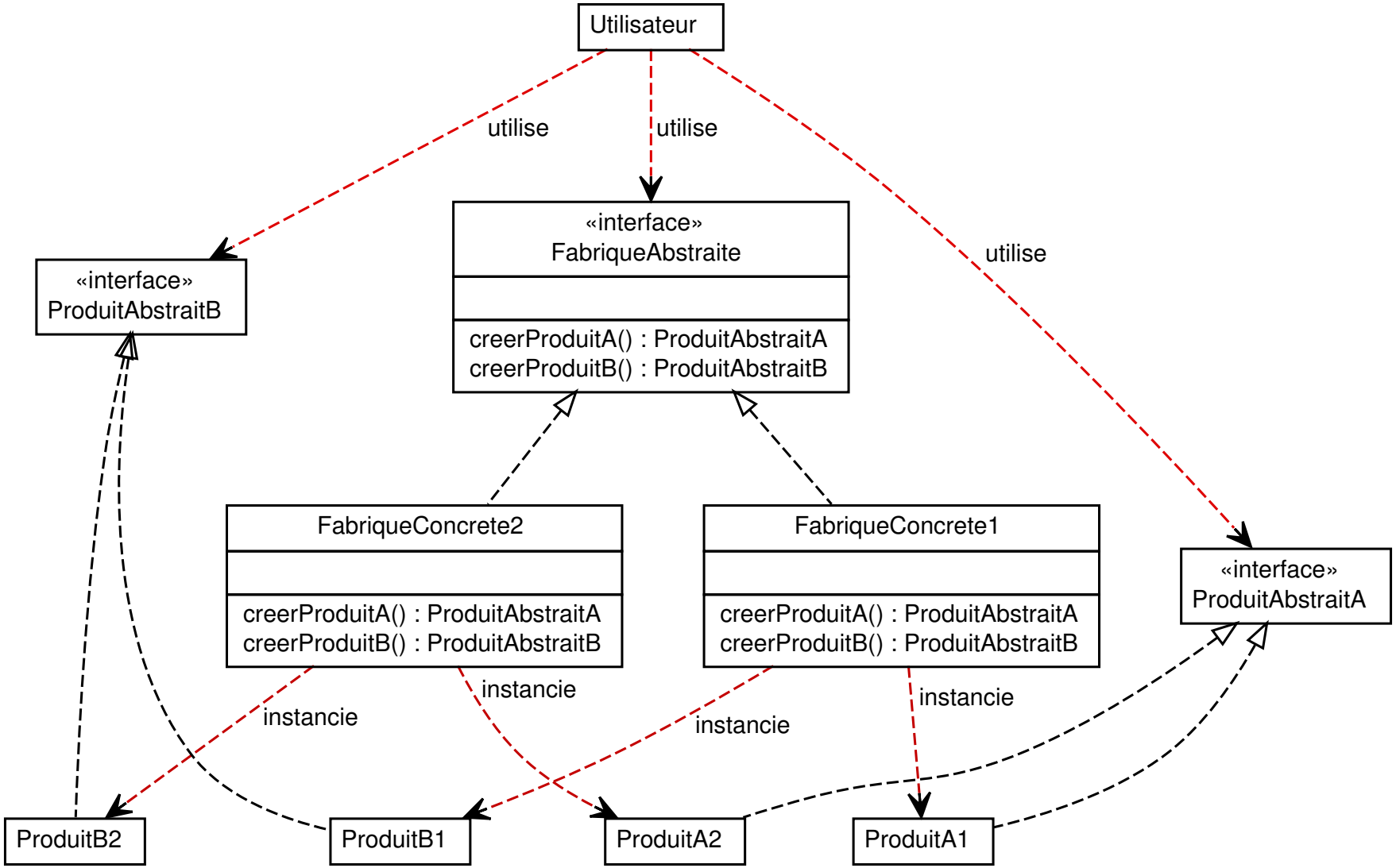
Patrons de conception – GoF – Structure

- Adaptateur (Adapter)
- Pont (Bridge)
- Objet composite (Composite)
- Décorateur (Decorator)
- Façade (Facade)
- Poids-mouche ou poids-plume (Flyweight)
- Proxy (Proxy)

Patrons de conception – GoF – Comportement

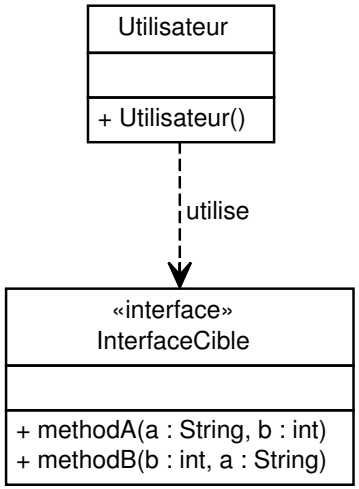
- Chaîne de responsabilité (Chain of responsibility)
- Commande (Command)
- Interpréteur (Interpreter)
- Itérateur (Iterator)
- Médiateur (Mediator)
- Mémento (Memento)
- Observateur (Observer)
- État (State)
- Stratégie (Strategy)
- Patron de méthode (Template Method)
- Visiteur (Visitor)
- Fonction de rappel (Callback)

Fabrique abstraite

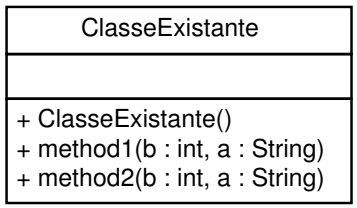


Objectif : Isoler la création des objets de leur utilisation

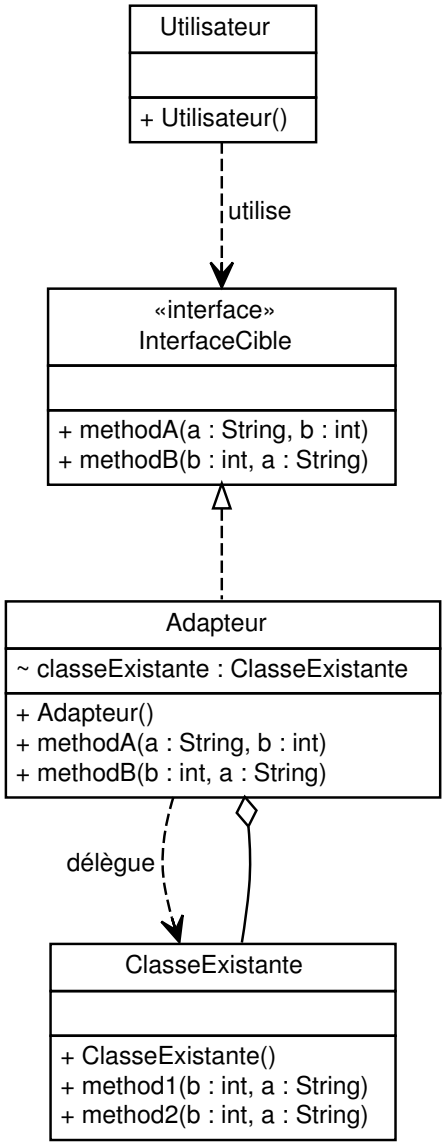
Adapteur



- methodA(a,b) $\rightarrow$ method1(b,a)
- methodB(b,a) $\rightarrow$ method2(b,a)

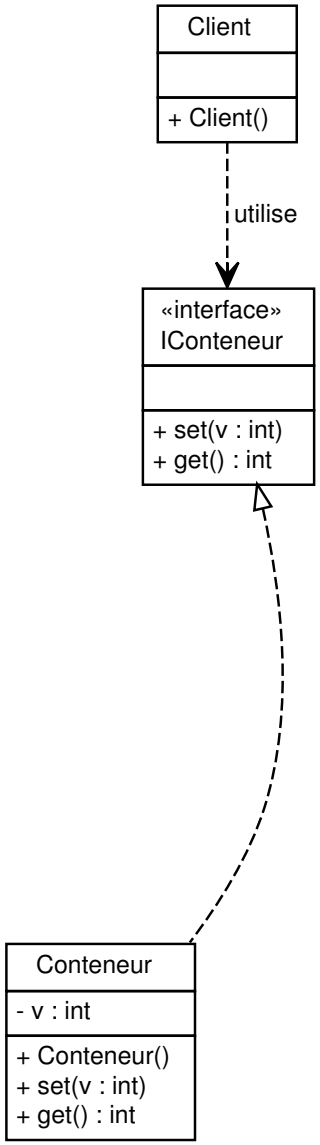


Adapteur



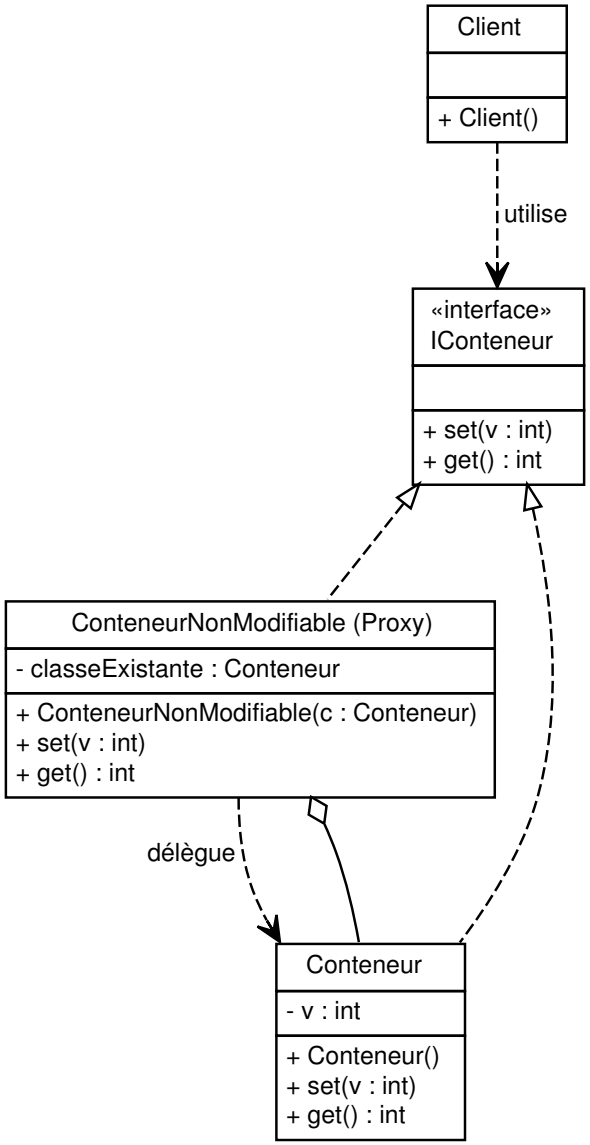

```
public class Adapteur implements InterfaceCible {

    ClasseExistante classeExistante;

    public Adapteur() {
        classeExistante = new ClasseExistante();
    }

    public void methodA(String a, int b) {
        classeExistante.method1(b, a);
    }

    public void methodB(int b, String a) {
        classeExistante.method2(b, a);
    }
}
```

Proxy




```
public class Conteneur implements IConteneur {

    private int v;

    public void set(int v) {
        this.v = v;
    }

    public int get() {
        return v;
    }
}
```

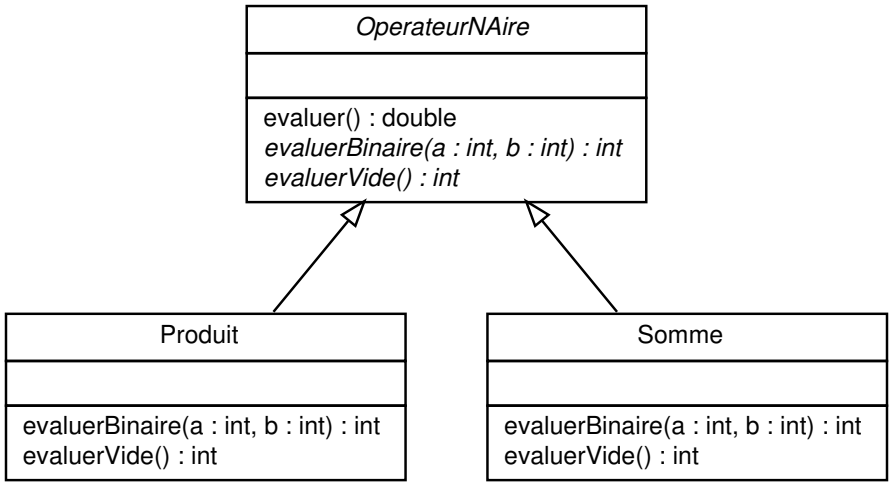
Proxy




```
public class ConteneurNonModifiable
                implements IConteneur {

    private Conteneur classeExistante;

    public ConteneurNonModifiable(Conteneur c) {
      classeExistante = c;
    }

    public void set(int v) {
      throw new UnsupportedOperationException();
    }

    public int get() {
      return classeExistante.get();
    }
}
```

Patron de méthode

Exemple d'utilisation du patron de méthode :



- La classe *OperateurNAire* est abstraite ;
- La méthode *evaluer* est une **méthode socle** qui utilise *evaluerBinaire* et *evaluerVide* pour définir son comportement ;
- Les méthodes *evaluerBinaire* et *evaluerVide* sont abstraites dans *OperateurNAire* et définies dans les classes qui étendent *OperateurNAire* ;

Fonction de rappel (Callback)

C/C++ :

```
typedef void (*Callback)(int); /* Pointeur sur fonction */

void callbackImpl1(int c) { printf("%d\n", c); }
void callbackImpl2(int c) { printf("* %d *\n", c); }

void function(int d, Callback callback) {
  /* ... */ callback(d); /* ... */
}

int main(void) {
  function(2, callbackImpl1);
  function(4, callbackImpl2);
}
```

Fonction de rappel (Callback)

Java :

```
public interface Callback { void method(int c); }

public class CallbackImpl1 implements Callback {
    public void method(int c) { System.out.println(c);  }
}

public class CallbackImpl2 implements Callback {
    public void method(int c) { System.out.println("* "+c+" *");  }
}

public class MaClasse {
    public static void function(int d, Callback callback) {
        /* ... */ callback.method(d); /* ... */
    }
    public static void main(String[] args) {
        Callback c1 = new CallbackImpl1(); function(2, c1);
        Callback c2 = new CallbackImpl2(); function(4, c2);
    }
}
```

Fonction de rappel (Callback)

C# :

```
using System;

public delegate void Callback(int c);

class MaClasse {

  public static void callbackImpl1(int c) {Console.WriteLine("{0}",c);}
  public static void callbackImpl2(int c) {Console.WriteLine("* {0} *",c);}

  public static void function(int d, Callback callback) {
    /* ... */ callback(d); /* ... */
  }

  static void Main(string[] args) {
    Callback c1 = new Callback(callbackImpl1); function(2,c1);
    Callback c2 = new Callback(callbackImpl2); function(4,c2);
  }
}
```

Les flux d'entrée-sortie

- Les flux (stream) permettent d'envoyer et de recevoir des données
- Les noms des classes qui gèrent les flux se terminent par :

	Flux d'octets	Flux de caractères
Flux d'entrée	InputStream	Reader
Flux de sortie	OutputStream	Writer

- Les classes qui gèrent les flux sont dans le package *java.io*
- Certaines classes permettent de lire ou d'écrire dans ou vers différents supports (fichier, chaîne de caractères, tubes, socket, etc.)
- D'autres permettent de faire des traitements sur les données (filtres)
- InputStream, OutputStream, Reader, Writer = classes abstraites

Les flux d'entrée (octets)

Flux d'octets d'entrée (étendant InputStream) :

- **ByteArrayInputStream** : Lecture dans un tableau d'octets
- **FileInputStream** : Lecture dans un fichier
- **ObjectInputStream** : Lecture d'objets dans un InputStream
- **PipedInputStream** : Lecture dans un tube
- **SequenceInputStream** : Union de plusieurs flux d'octets
- **AudioInputStream** : Lecture dans un flux audio

Les flux d'entrée (octets)

Flux d'octets d'entrée (étendant InputStream) :

- **FilterInputStream** : Ajoute un traitement sur le flux d'entrée
  - **BufferedInputStream** : Ajoute un tapon
  - **CheckedInputStream** : Calcule un checksum sur l'entrée
  - **CipherInputStream** : Chiffre ou déchiffre le flux
  - **DataInputStream** : Permet de lire des données primitives Java
  - **LineNumberInputStream** : Permet de connaître la ligne courante
  - **InflaterInputStream** : Permet de décompresser un flux
    - **GZIPInputStream** : Format GZIP
    - **ZipInputStream** : Format ZIP
  - ...

Les flux d'entrée (octets)

Exemple :

```
byte[] bytes = {10, 11, 12};

InputStream is = new ByteArrayInputStream(bytes);

while (is.available() > 0)
    System.out.println(is.read());
```

Sortie :

```
10
11
12
```

Les flux d'entrée (octets)

Les méthodes de la classe **InputStream** :

- **int available()** : retourne nombre d'octets restants
- **void close()** : ferme le flux
- **void mark(int readlimit)** : marque la position courante
- **boolean markSupported()** : est-ce mark et reset sont supportés ?
- **abstract int read()** : lit un octet
- **int read(byte[] b)** : lit une sequence d'octets
- **int read(byte[] b, int off, int len)** : lit une sequence d'octets
- **void reset()** : retourne à la marque précédente
- **long skip(long n)** : saute des octets

Les flux d'entrée (octets)

Création d'un fichier :

```
$ echo "abc" > abc.txt
```

Exemple de lecture dans un fichier :

```
InputStream is = new FileInputStream("abc.txt");
while (is.available()>0) {
    System.out.println(is.read());
}
is.close();
```

Sortie :

```
97
98
99
10
```

Les flux d'entrée (octets)

Construction d'un fichier Zip :

```
$ echo "abc" > abc.txt
$ echo "truc" > truc.txt
$ zip archive.zip abc.txt truc.txt
```

Exemple de lecture dans un fichier Zip :

```
InputStream is = new FileInputStream("archive.zip");
is = new BufferedInputStream(is);
ZipInputStream zip = new ZipInputStream(is);
ZipEntry entry = zip.getNextEntry();
while (entry != null) {
    System.out.println("Contenu de "+ entry.getName()+" :");
    for (int c = zip.read(); c != -1; c = zip.read())
        System.out.print(c+" ");
    zip.closeEntry();
    entry = zip.getNextEntry();
}
zip.close();
```

Les flux d'entrée (octets)

Construction d'un fichier Zip :

```
$ echo "abc" > abc.txt
$ echo "truc" > truc.txt
$ zip archive.zip abc.txt truc.txt
```

Sortie :

```
Contenu de abc.txt :
97 98 99 10
Contenu de truc.txt :
116 114 117 99 10
```

Les flux d'entrée (caractères)

Flux de caractères d'entrée (étendant Reader) :

- **CharArrayReader** : Lecture dans un tableau de caractères
- **BufferedReader** : Lecture avec un tapon
  - **LineNumberReader** : Permet de connaître le numéro de ligne
- **InputStreamReader** : Lecture dans un flux d'octets
  - **FileReader** : Lecture dans un fichier
- **PipedReader** : Lecture dans un tube
- **StringReader** : Lecture dans une chaîne de caractères
- **FilterReader** : Ajoute un traitement sur le flux d'entrée
  - **PushbackReader** : Permet de remettre des caractères dans le flux

Les flux d'entrée (caractères)

Exemple :

```
Reader r =  new StringReader("toto");

int c;
while ((c = r.read())!=-1)
    System.out.println((char)c);
```

Sortie :

```
t
o
t
o
```

Les flux d'entrée (octets)

Les méthodes de la classe **Reader** :

- **void close()** : ferme le flux
- **void mark(int readlimit)** : marque la position courante
- **boolean markSupported()** : est-ce mark et reset sont supportés ?
- **abstract int read()** : lit un caractère
- **int read(char[] b)** : lit une séquence de caractères
- **int read(char[] b, int off, int len)** : lit une séquence de caractères
- **int ready()** : est-ce que le flux est prêt à être lu ?
- **void reset()** : retourne à la marque précédente
- **long skip(long n)** : saute des caractères

Les flux d'entrée (caractères)

Pour lire un fichier ligne à ligne, il faut utiliser un BufferedReader :

```
Reader reader = new StringReader("L1\nL2\nL3");
BufferedReader bufferedReader = new BufferedReader(reader);

String ligne;
while ((ligne = bufferedReader.readLine())!=null)
    System.out.println(ligne);
```

Sortie :

```
L1
L2
L3
```

Les flux d'entrée (caractères)

Autre exemple :

```
Reader reader =  new StringReader("L1\nL2\nL3");
LineNumberReader lineNumberReader =
                    new LineNumberReader(reader);

String ligne;
while ((ligne = lineNumberReader.readLine())!=null) {
   System.out.println(lineNumberReader.getLineNumber()+" : ");
   System.out.println(ligne);
}
```

Sortie :

```
1 : L1
2 : L2
3 : L3
```

Les flux de sortie (octets)

Flux d'octets de sortie (étendant OutputStream) :

- **ByteArrayOutputStream** : Écriture dans un tableau d'octets
- **FileOutputStream** : Écriture dans un fichier
- **ObjectOutputStream** : Écriture d'objets dans un OutputStream
- **PipedOutputStream** : Écriture dans un tube

Les flux de sortie (octets)

Flux d'octets de sortie (étendant OutputStream) :

- **FilterOutputStream** : Ajoute un traitement sur le flux de sortie
  - **BufferedOutputStream** : Ajoute un tapon
  - **CheckedOutputStream** : Calcule un checksum sur la sortie
  - **CipherOutputStream** : Chiffre ou déchiffre le flux
  - **DataOutputStream** : Permet d'écrire des données primitives Java
  - **PrintStream** : Ajoute des fonctions de formatage
  - **DeflaterOutputStream** : Permet de compresser un flux
    - **GZIPOutputStream** : Format GZIP
    - **ZipOutputStream** : Format ZIP
  - ...

Les flux de sortie (octets)

Exemple :

```
ByteArrayOutputStream output = new ByteArrayOutputStream();

output.write(97);
output.write(98);
output.write(99);

byte[] bytes = output.toByteArray();
for (byte b : bytes)
    System.out.println(b);
System.out.println(output);
```

Sortie :

```
97
98
99
abc
```

Les flux de sortie (octets)

Les méthodes de la classe **OutputStream** :

- **void close()** : ferme le flux
- **int write(byte[] b)** : écrit une sequence d'octets
- **int write(byte[] b, int off, int len)** : écrit une sequence d'octets
- **abstract void write(int b)** : écrit un octet sur le flux
- **void flush()** : force les buffers à être écrit

Les flux de sortie (octets)

Copie de deux fichiers :

```
InputStream input = new BufferedInputStream(
                        new FileInputStream("original.pdf")
                    );
OutputStream output = new BufferedOutputStream(
                        new FileOutputStream("copie.pdf")
                    );

byte[] b = new byte[200];

int n = input.read(b);
while (n!=-1) {
    output.write(b, 0, n);
    n = input.read(b);
}

input.close();
output.close();
```

Les flux de sortie (caractères)

Flux de caractères de sortie (étendant Writer) :

- **CharArrayWriter** : Écriture dans un tableau de caractères
- **BufferedWriter** : Écriture avec un tapon
- **OutputStreamWriter** : Écriture dans un flux d'octets
  - **FileWriter** : Écriture dans un fichier
- **PipedWriter** : Écriture dans un tube
- **StringWriter** : Écriture dans une chaîne de caractères
- **PrintWriter** : Écriture avec des méthodes de formatage
- **FilterWriter** : Ajoute un traitement sur le flux de sortie

Les flux de sortie (caractères)

Numérotation des lignes d'un fichier texte :

```
LineNumberReader input = new LineNumberReader(
                            new FileReader("t.txt")
                         );
Writer output = new BufferedWriter(
                            new FileWriter("t2.txt")
                        );

String s = input.readLine();
while (s!=null) {
    output.write(input.getLineNumber()+" : ");
    output.write(s+"\n");
    s = input.readLine();
}

input.close();
output.close();
```

Les flux de sortie (caractères)

Écriture formatée dans un fichier :

```
PrintWriter output = new PrintWriter("texte.txt");
output.println("bonjour");
output.printf("%d -- %d -- %d\n", 12, 14, 15);
output.printf("%s %.2f %04d\n", "aaaa", 12.44557, 45);
output.close();
```

Contenu du fichier après l'exécution :

```
bonjour
12 -- 14 -- 15
aaaa 12,45 0045
```

System.{out,in,err}

Dans la classe **System** :

- public static final InputStream in : entrée standard
- public static final PrintStream out : sortie standard
- public static final PrintStream err : sortie d'erreur
- public static Console console() : retourne la console

Exemple :

```
System.out.println("ecriture sur la sortie standard");
System.err.println("ecriture sur la sortie d'erreur");

BufferedReader input = new BufferedReader(
                           new InputStreamReader(System.in));
String s = input.readLine();
while (s!=null) {
    System.out.println(s);
    s = input.readLine();
}
```

System.console()

Dans la classe **Console** :

- void flush() : vide le buffer
- Console format(String fmt, Object... args) : écriture formatée
- Console printf(String format, Object... args) : écriture formatée
- Reader reader() : retourne le Reader de la console
- String readLine() : lit une ligne
- String readLine(String fmt, Object... args) : idem avec format
- char[] readPassword() : lit un mot de passe
- char[] readPassword(String fmt, Object... args) : idem avec format
- PrintWriter writer() : retourne le Writer de la console

Attention : *System.console()* retourne une référence *null* si le système ne supporte pas cette fonctionnalité

Les exceptions et les erreurs

Quelques exceptions liées aux entrées-sorties :

- IOError
- IOException
  - EOFException
  - FileNotFoundException
  - ZipException
  - CharacterCodingException
    - MalformedInputException
    - UnmappableCharacterException
  - ...
- IllegalFormatException

Exemple :

```
try {
    FileReader reader = new FileReader("truc.txt");
} catch (FileNotFoundException ex) {
    System.out.println("truc.txt n'existe pas !");
}
```

Astuce : la classe *Scanner*

Exemples d'utilisation de la classe Scanner :

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();

String input = "1 abc 2 abc truc abc machin abc";
Scanner s = new Scanner(input).useDelimiter("\\s*abc\\s*");
System.out.println(s.nextInt());
System.out.println(s.nextInt());
System.out.println(s.next());
System.out.println(s.next());
s.close();

String input = "1 abc 2 abc truc abc machin abc";
Scanner s = new Scanner(input);
s.findInLine("(\\d+) abc (\\d+) abc (\\w+) abc (\\w+)");
MatchResult result = s.match();
for (int i=1; i<=result.groupCount(); i++)
        System.out.println(result.group(i));
s.close();
```

Java et le multitâche

- Un programme peut lancer plusieurs **threads**
- Les threads lancés par un programme s'éxecutent **en parallèle**
- Si l'ordinateur n'a pas assez de processeur pour exécuter tous les threads en parallèle, le système d'exploitation va simuler le multitâche en changeant régulièrement le thread en cours d'exécution
- Tous les threads partagent **le même espace mémoire**
- En Java, l'interface **Runnable** et la classe **Thread** permettent de créer des threads

L'interface *Runnable*

- L'interface *Runnable* contient la méthode **void run()**

```
public class Compteur implements Runnable {

    public void run() {
        int cpt = 0;
        while (cpt < 100) {
            System.out.print(cpt+" ");
            cpt++;
        }
    }
}
```

La classe *Thread*

- Chaque instance de cette classe correspond à un thread
- Elle implémente l'interface *Runnable*

Quelques méthodes non-statiques de la classe Thread :

- **Thread()** : constructeur par défaut
- **Thread(Runnable target)** : constructeur avec un Runnable
- **void start()** : démarre le thread
- **void join()** : attend que le thread meure
- **void interrupt()** : interrompt le thread
- **boolean isAlive()** : est-ce que le thread est en vie ?
- **boolean isInterrupted()** : est-ce que le thread est interrompu ?
- **boolean isDaemon()** : est-ce que le thread est un démon ?

Lancement d'un thread

Première méthode :

```
public class Compteur implements Runnable {
    public void run() {
        int cpt = 0;
        while (cpt < 10) { System.out.print(cpt+" "); cpt++; }
    }
}

public class Main  {
    public static void main(String[] args) {
        Thread th1 = new Thread(new Compteur());
        th1.start();
        Thread th2 = new Thread(new Compteur());
        th2.start();
    }
}
```

Sortie :

0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9

Lancement d'un thread

Deuxième méthode :

```
public class Compteur extends Thread {
    public void run() {
        int cpt = 0;
        while (cpt < 10) { System.out.print(cpt+" "); cpt++; }
    }
}

public class Main  {
    public static void main(String[] args) {
        Thread th1 = new Compteur();
        th1.start();
        Thread th2 = new Compteur();
        th2.start();
    }
}
```

Sortie :

0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9

La classe *Thread*

Les méthodes statiques agissent sur le thread **en cours d'exécution**

Quelques méthodes statiques de la classe Thread :

- **Thread currentThread()** : retourne le thread courant
- **boolean interrupted()** : est-ce que le thread courant est interrompu ?
- **void sleep(long millis)** : endort le thread courant
- **void yield()** : interrompt le thread courant
- **int activeCount()** : nombre de threads actifs

L'état d'interruption

- Interrompre un thread signifie vouloir qu'il arrête son exécution
- Un thread peut être interrompu avec la méthode **interrupt()**
- Si le thread dort, il faut le réveiller pour qu'il puisse s'interrompre
- **Solution** : L'état d'interruption et l'exception InterruptedException
- La méthode **interrupted()** permet de consulter l'état d'interruption

Compteur qui s'arrête quand le thread est interrompu :

```
public class Compteur extends Thread {
    public void run() {
        int cpt = 0;
        while (!interrupted()) cpt++;
        System.out.println(cpt);
    }
}
```

L'état d'interruption

Utilisation du compteur :

```
public class Main {
    public static void main(String[] args) {
        Thread th1 = new Compteur();
        th1.start(); ...; th1.interrupt();
    }
}
```

Compteur qui compte moins vite :

```
public class Compteur extends Thread {
    public void run() {
        int cpt = 0;
        while (!interrupted()) {
          cpt++;
          Thread.sleep(100);
          // Problème : interruption pendant que je dors!
        }
        System.out.println(cpt);
    }
}
```

L'état d'interruption

Utilisation du compteur :

```
public class Main {
    public static void main(String[] args) {
        Thread th1 = new Compteur();
        th1.start(); ...; th1.interrupt();
    }
}
```

Compteur qui compte moins vite :

```
public class Compteur extends Thread {
    public void run() {
        int cpt = 0;
        while (!interrupted()) {
          cpt++;
          try { Thread.sleep(100); }
          catch(InterruptedException e) { break; }
        }
        System.out.println(cpt);
    }
}
```

L'état d'interruption

Thread qui interrompt un autre thread après un certain temps :

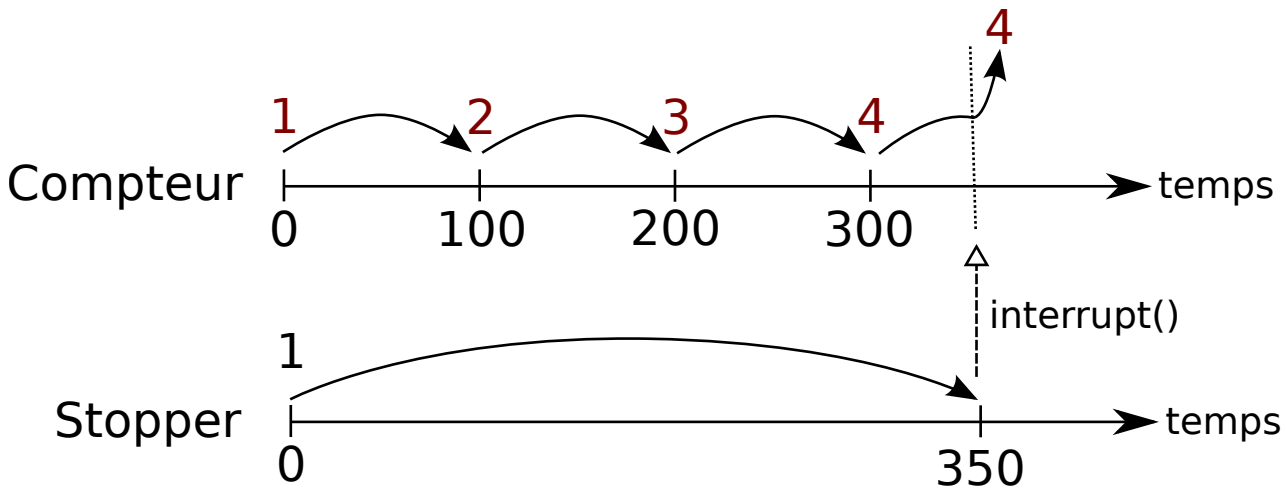
```
public class Stopper extends Thread {

    Thread thread;
    int timeLimit;

    public Stopper(Thread thread, int timeLimit) {
        this.thread = thread;
        this.timeLimit = timeLimit;
    }

    void run() {
        try {
            sleep(timeLimit);
        } catch (InterruptedException ex) {
            return;
        }
        thread.interrupt();
    }
}
```

L'état d'interruption

Exemple d'utilisation des classes précédentes :

```
public static void main(String[] args) {
        Thread th1 = new Compteur();
        th1.start();
        Thread th2 = new Stopper(th1, 350);
        th2.start();
}
```

Exécution :



Problème de synchronisation

Une pile d'entiers avec une temporisation lors des empilements :

```
public class PileLente {
    private int[] tab; private int size;

    public PileLente() {
        tab = new int[1000]; size = 0;
    }

    void push(int p) throws InterruptedException {
        int s = size; tab[s] = p;
        Thread.sleep(100);
        size = s + 1;
    }

    int pop() { size--; return tab[size]; }

    int size() { return size; }
}
```

Synchronisation

Un thread qui empile 10 entiers :

```
public class Empileur extends Thread {

    private PileLente pile;

    public Empileur(PileLente pile) {
        this.pile = pile;
    }

    public void run() {
        for (int i = 0; i < 10; i++)
            try {
            pile.push(i);
        } catch (InterruptedException ex) {
            return;
        }
    }
}
```

Synchronisation

Exécution en parallèle de deux empileurs sur une même pile :

```
public static void main(String[] args)
                              throws InterruptedException {
        PileLente pile = new PileLente();
        Thread th1 = new Empileur(pile);
        Thread th2 = new Empileur(pile);
        th1.start();
        th2.start();
        th1.join(); // On attend la fin du thread 1
        th2.join(); // On attend la fin du thread 2
        while (pile.size()>0) {
            System.out.print(pile.pop()+" ");
}
```

Sortie :

```
9 8 7 6 5 4 3 2 1 0
```

Question : Où sont passés les 10 éléments manquants ?

Synchronisation

Exécution (en parallèle) de la méthode **push** dans deux threads :

Thread 1	Thread 2
int s = size ; (s = x)	
tab[s] = p ;	
sleep(100) ;	int s = size ; (s = x)
– –	tab[s] = p ;
– –	sleep(100) ;
– –	– –
size = s + 1 ; (size = $x + 1$)	– –
	size = s + 1 ; (size = $x + 1$)

Problème : Deux entiers ont été empilés et la pile et la taille de la pile est passée de x à $x + 1$

Le mot-clé *Synchronized*

Interdire que deux invocations sur une même instance s'entremêlent :

```
public class PileLente {
    ...
    synchronized void push(int p) throws InterruptedException {
        int s = size; tab[s] = p;
        Thread.sleep(100);
        size = s + 1;
    }
    synchronized int pop() { size--; return tab[size]; }
    ...
}
```

Signification : Si un thread **T** invoque la méthode **push** alors qu'une autre méthode synchronisée est en cours d'exécution dans un autre thread alors le thread **T** est suspendu et doit attendre que la méthode **push** soit disponible (c'est-à-dire, qu'aucun thread ne soit en train d'exécuter une méthode synchronisée sur l'instance)

Obtenir des structures synchronisées en Java

Transformation d'une structure de données en sa version synchronisée :

```
Collection<String> collection = new ArrayList<String>();
collection = Collections.synchronizedCollection(collection);

List<String> list = new ArrayList<String>();
list = Collections.synchronizedList(list);

Set<String> set = new HashSet<String>();
set = Collections.synchronizedSet(set);

Map<String, Integer> map = new HashMap<String, Integer>();
map = Collections.synchronizedMap(map);
```

Question : Comment obtenir la même fonctionnalité sur une structure de données que vous avez écrite ?

Swing et les threads

- Chaque évènement est géré dans un thread séparé
- Si une exception est jetée lors de son traitement, seul le thread est tué
- Cela permet de rendre les applications graphiques plus robustes

La "bonne façon" de débuter une application Swing :

```
public class Main {

  public void creerEtMontrer() { .... }

  public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        creerEtMontrer();
      }
    });
  }
}
```

JavaDoc

- Outil pour générer automatiquement **une documentation** du code
- Il génère une documentation au format HTML
- Cette documentation est lisible dans Eclipse ou NetBeans
- Le document contient :
  - Une description des membres (publics et protégés) des classes
  - Des liens permettant de naviguer entre les classes
  - Des informations sur l'héritage
- La documentation est générée à partir des commentaires
- Les commentaires doivent respecter un certain format

Exemple de JavaDoc

Overview Package **Class** Use Tree Deprecated Index Help

Java™ Platform Standard Ed. 6

PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

java.awt.event

Interface MouseListener

All Superinterfaces:
EventListener

All Known Subinterfaces:
MouseInputListener

All Known Implementing Classes:
AWTEventMulticaster, BasicButtonListener, BasicComboPopup.InvocationMouseHandler, BasicComboPopup.ListMouseHandler, BasicDesktopIconUI.MouseInputHandler, BasicFileChooserUI.DoubleClickListener, BasicInternalFrameUI.BorderListener, BasicInternalFrameUI.GlassPaneDispatcher, BasicListUI.MouseInputHandler, BasicMenuItemUI.MouseInputHandler, BasicMenuUI.MouseInputHandler, BasicScrollBarUI.ArrowButtonListener, BasicScrollBarUI.TrackListener, BasicSliderUI.TrackListener, BasicSplitPaneDivider.MouseHandler, BasicTabbedPaneUI.MouseHandler, BasicTableHeaderUI.MouseInputHandler, BasicTableUI.MouseInputHandler, BasicTextUI.BasicCaret, BasicToolBarUI.DockingListener, BasicTreeUI.MouseHandler, BasicTreeUI.MouseInputHandler, DefaultCaret, FormView.MouseEventListener, HTMLEditorKit.LinkController, MetalFileChooserUI.SingleClickListener, MetalToolBarUI.MetalDockingListener, MouseAdapter, MouseDragGestureRecognizer, MouseInputAdapter, ToolTipManager

---

```
public interface MouseListener
extends EventListener
```

The listener interface for receiving "interesting" mouse events (press, release, click, enter, and exit) on a component. (To track mouse moves and mouse drags, use the `MouseMotionListener`.)

The class that is interested in processing a mouse event either implements this interface (and all the methods it contains) or extends the abstract `MouseAdapter` class (overriding only the methods of interest).

The listener object created from that class is then registered with a component using the component's `addMouseListener` method. A mouse event is generated when the mouse is pressed, released clicked (pressed and released). A mouse event is also generated when the mouse cursor enters or leaves a component. When a mouse event occurs, the relevant method in the listener object is invoked, and the `MouseEvent` is passed to it.

Since:
1.1

See Also:
`MouseAdapter`, `MouseEvent`, Tutorial: Writing a Mouse Listener

Exemple de JavaDoc

Method Summary	
void	**mouseClicked**(MouseEvent e) Invoked when the mouse button has been clicked (pressed and released) on a component.
void	**mouseEntered**(MouseEvent e) Invoked when the mouse enters a component.
void	**mouseExited**(MouseEvent e) Invoked when the mouse exits a component.
void	**mousePressed**(MouseEvent e) Invoked when a mouse button has been pressed on a component.
void	**mouseReleased**(MouseEvent e) Invoked when a mouse button has been released on a component.

Method Detail

mouseClicked

void **mouseClicked**(MouseEvent e)

Invoked when the mouse button has been clicked (pressed and released) on a component.

mousePressed

void **mousePressed**(MouseEvent e)

Invoked when a mouse button has been pressed on a component.

mouseReleased

void **mouseReleased**(MouseEvent e)

Invoked when a mouse button has been released on a component.

mouseEntered

void **mouseEntered**(MouseEvent e)

Invoked when the mouse enters a component.

Exemple de JavaDoc

`void` **`mouseClicked`**`(`MouseEvent `e)`

Invoked when the mouse button has been clicked (pressed and released) on a component.

---

mousePressed

`void` **`mousePressed`**`(`MouseEvent `e)`

Invoked when a mouse button has been pressed on a component.

---

mouseReleased

`void` **`mouseReleased`**`(`MouseEvent `e)`

Invoked when a mouse button has been released on a component.

---

mouseEntered

`void` **`mouseEntered`**`(`MouseEvent `e)`

Invoked when the mouse enters a component.

---

mouseExited

`void` **`mouseExited`**`(`MouseEvent `e)`

Invoked when the mouse exits a component.

Overview Package **Class** Use Tree Deprecated Index Help

Java™ Platform Standard Ed. 6

PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Submit a bug or feature

For further API reference and developer documentation, see Java SE Developer Documentation. That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

Exemple de commentaire JavaDoc

```
/**
 * Cette classe contient une méthode additionner.
 */
public class Truc {
 /**
  * Retourne la somme des deux valeurs
  * spécifiées en parametre.
  * @param a la première valeur.
  * @param b la deuxième valeur.
  * @return la somme des deux valeurs.
  */
   public int additionner(int a, int b) {
     return a+b;
   }
}
```

Résultat du commentaire précédent

Package Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS
SUMMARY: NESTED | FIELD | CONSTR | METHOD

FRAMES NO FRAMES All Classes
DETAIL: FIELD | CONSTR | METHOD

Class Truc

java.lang.Object
  └ **Truc**

public class **Truc**
extends java.lang.Object

Cette classe contient une méthode additionner.

Constructor Summary

Truc()

Method Summary

int	additionner(int a, int b) Retourne la somme des deux valeurs spécifiées en parametre.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Truc

Résultat du commentaire précédent

Method Summary

int	**additionner**(int a, int b) Retourne la somme des deux valeurs spécifiées en parametre.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Truc

```
public Truc()
```

Method Detail

additionner

```
public int additionner(int a,
                       int b)
```

Retourne la somme des deux valeurs spécifiées en parametre.

Parameters:

`a` - la première valeur.
`b` - la deuxième valeur.

Returns:

la somme des deux valeurs.

Package Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS
SUMMARY: NESTED | FIELD | CONSTR | METHOD

FRAMES NO FRAMES All Classes
DETAIL: FIELD | CONSTR | METHOD

Les tags disponibles

@author	1.0	pour préciser l'auteur de la fonctionnalité
@docRoot	1.3	chemin relatif qui mène à la racine de la doc.
@deprecated	1.0	indique que le membre ou la classe est dépréciée
@link	1.2	permet de créer un lien
@param	1.0	pour décrire un paramètre
@return	1.0	pour décrire la valeur de retour
@see	1.0	pour ajouter une section "Voir aussi"
@since	1.1	depuis quelle version la fonctionnalité est disponible
@throws	1.2	pour indiquer les exceptions jetées par les méthodes
@version	1.0	pour indiquer la version de la classe
@exception	1.0	synonyme de @throws
@serial	1.2	(pour la sérialisation)
@serialData	1.2	(pour la sérialisation)
@serialField	1.2	(pour la sérialisation)

Commentaire d'une classe ou d'une interface

```
/**
 * Une classe truc qui permet d'additionner
 * Par exemple :
 * <pre>
 *    Truc truc = new Truc();
 *    int b = truc.additionner(3, 6);
 * </pre>
 *
 * @author  Bertrand Estellon
 * @version 1.0
 * @see     java.lang.String
 */
public class Truc {
  ....
}
```

Commentaire d'une classe ou d'une interface

Class Truc

```
java.lang.Object
  └ Truc
```

```
public class Truc
extends java.lang.Object
```

Une classe truc qui permet d'additionner Par exemple :

```
Truc truc = new Truc();
int b = truc.additionner(3, 6);
```

See Also:

String

Commentaire des champs

```
public class Position {
    /**
     * La coordonnée X.
     *
     * @see #getPoint()
     */
    public int x = 12;

    /**
     * La coordonnée Y.
     *
     * @see #getPoint()
     */
    public int y = 40;
    ....
}
```

Commentaire des champs

x

```
public int x
```

La coordonnée X.

See Also:

#getPoint()

---

y

```
public int y
```

La coordonnée Y.

See Also:

#getPoint()

Commentaire des méthodes

```
/**
 * Returns the character at the
 * specified index. An index ranges from
 * <code>0</code> to <code>length() - 1</code>.
 *
 * @param     index  the index of the desired character.
 * @return    the desired character.
 * @throws    StringIndexOutOfRangeException
 *              if the index is not
 *              in the range <code>0</code>
 *              to <code>length()-1</code>.
 * @see       java.lang.Character#charValue()
 */
public char charAt(int index) {
    ....
}
```

Commentaire des méthodes

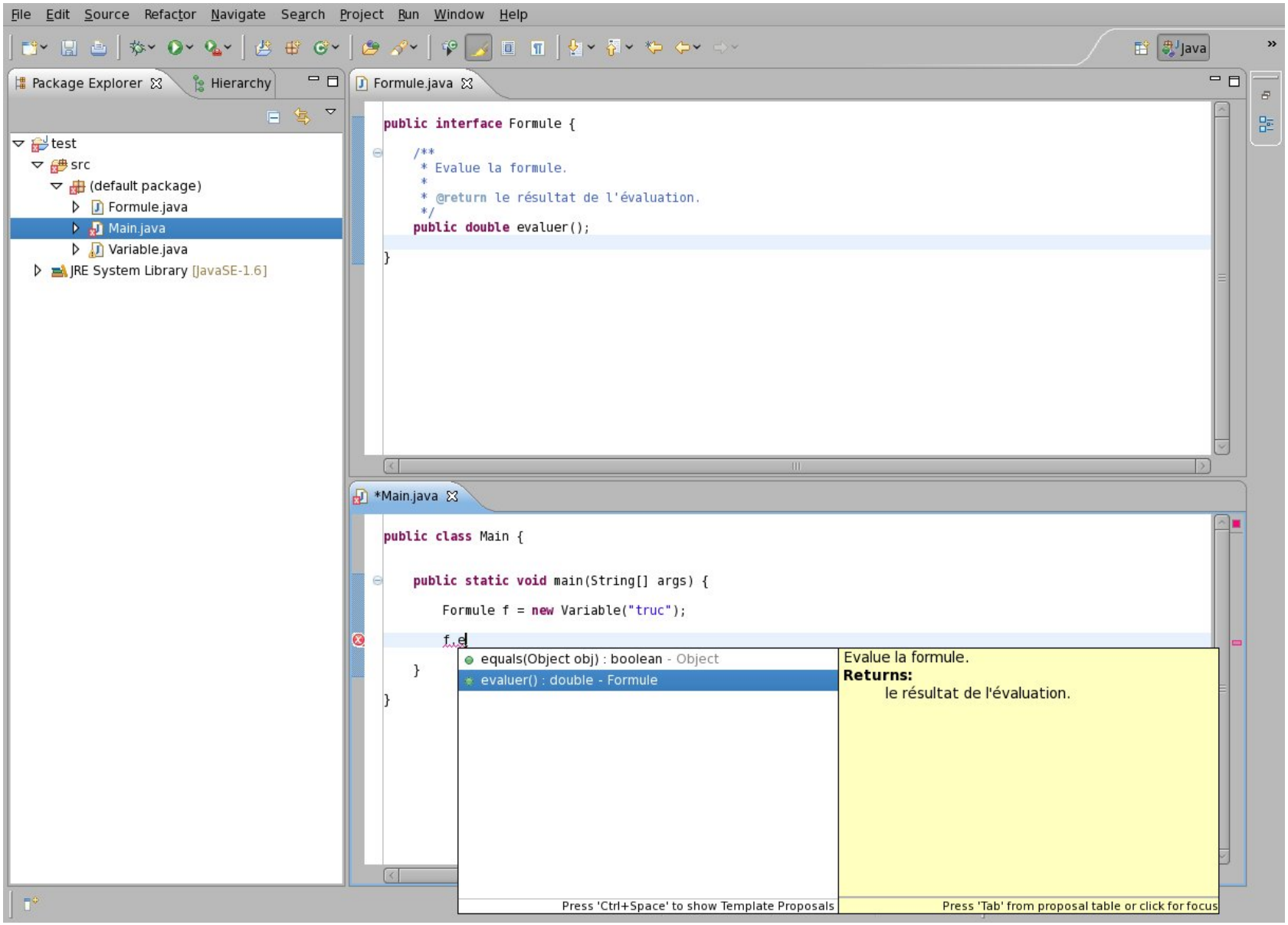
charAt

```
public char charAt(int index)
```

Returns the character at the specified index. An index ranges from `0` to `length() - 1`.

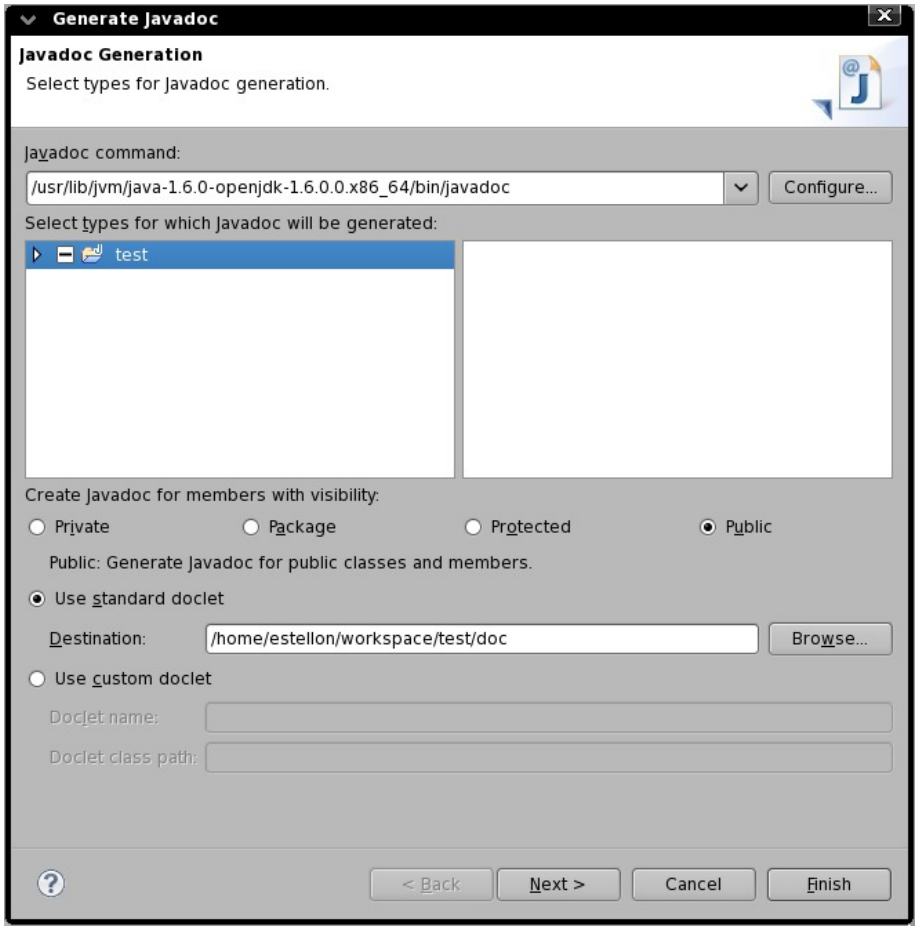
Parameters:
`index` - the index of the desired character.
Returns:
the desired character.
Throws:
`StringIndexOutOfRangeException` - if the index is not in the range `0` to `length()-1`.
See Also:
`Character.charValue()`

Intégration dans les IDE



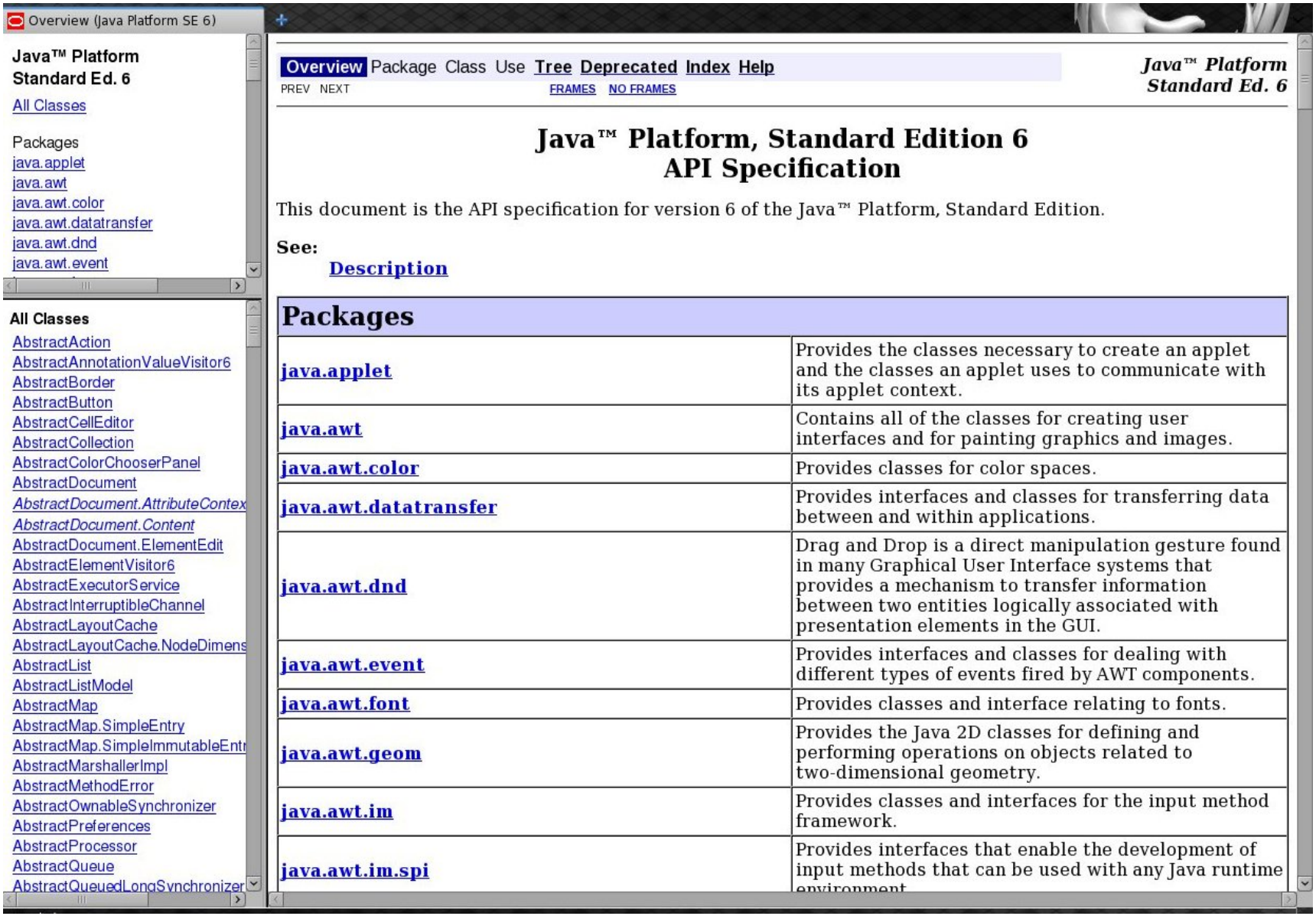
Génération de la documentation

- Avec Eclipse ou NetBeans :



- En ligne de commande avec la commande **javadoc**

Documentation d'un projet...



Révision – Les mots réservés...

Gestions des paquets :

`package`
`import`

Définitions des classes, interfaces et énumérations :

`class`
`interface`
`enum`

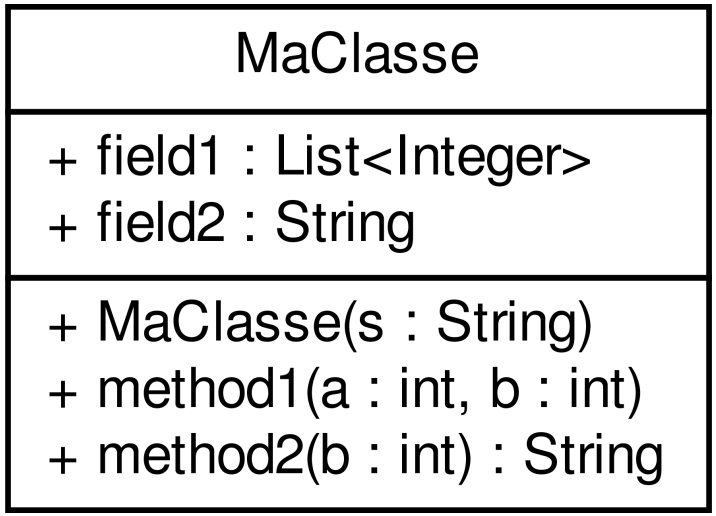
Héritage :

`extends`
`implements`

Révision – Classes

```
class UneClasse {

  int field1;
  String field2;

  UneClasse(int p1) { /* Constructeur de Truc */
    field1 = p1;
  }

  int method(int p1, int p2) {
    return p1 + p2 + field1;
  }

}
```

Révision – UML – Schéma d'une classe



- **Champ** → nom ' :' type
- **Méthode** → nom '(' arguments ')' ' :' type
- **Arguments** → argument (',' argument) *
- **Argument** → nom ' :' type

Révision – Interfaces

```
interface UneInterface {

  /** Description du comportement de la méthode.
   *
   * @param p1 ...
   */
  void method1(int p1);

  /** Description du comportement de la méthode.
   *
   * @param p1 ...
   * @param p2 ...
   * @return ....
   */
  int method2(int p2, int p3);
}
```

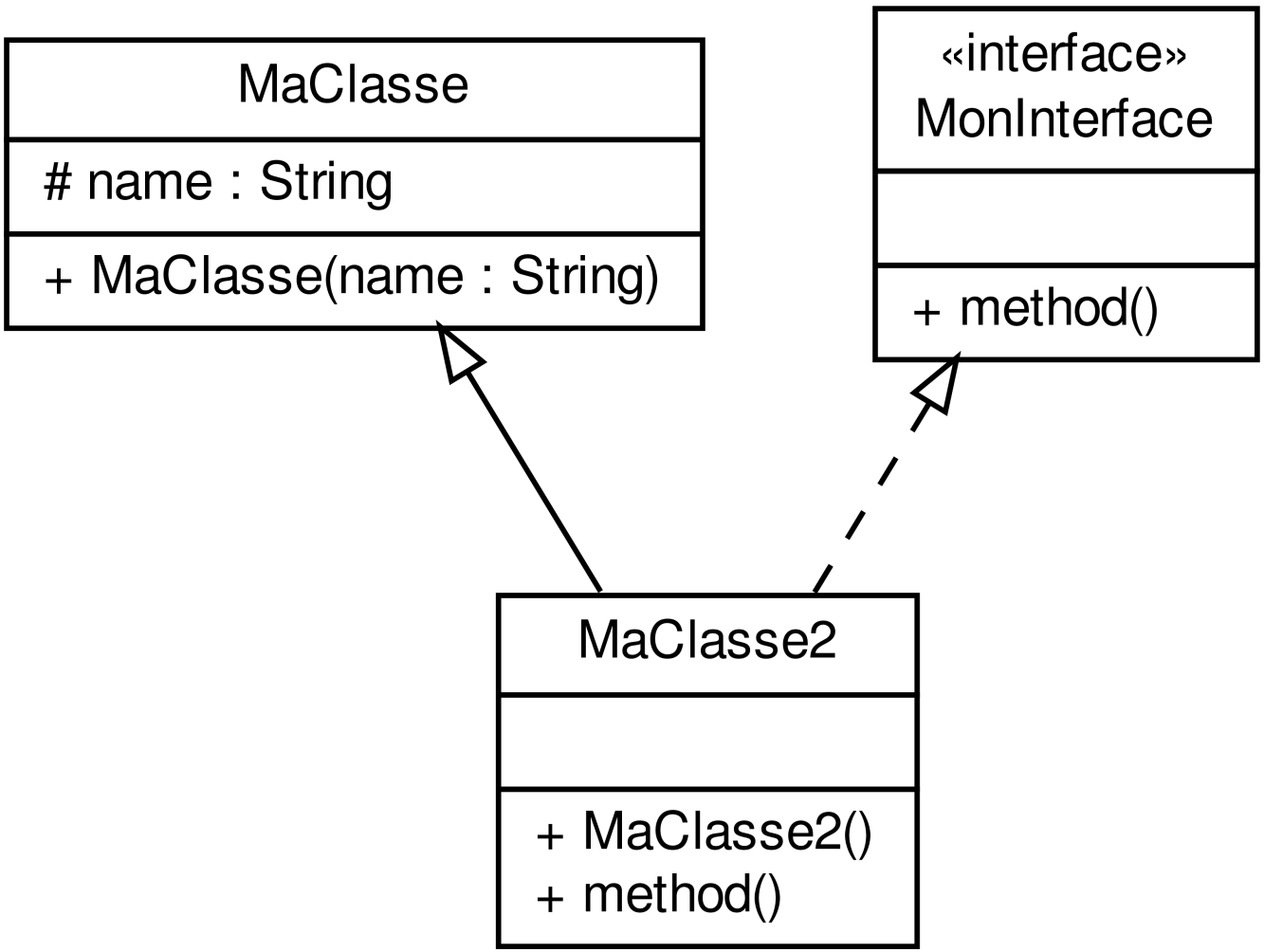
Révision – Implémentation d'une interface

```
class UneClasse implements UneInterface {

 String nom;

 UneClasse(String nom) {
  this.nom = nom;
 }

 void method1(int p1) {
  System.out.println(p1);
 }

 int method2(int p1, int p2) {
  return p1+p2;
 }
```

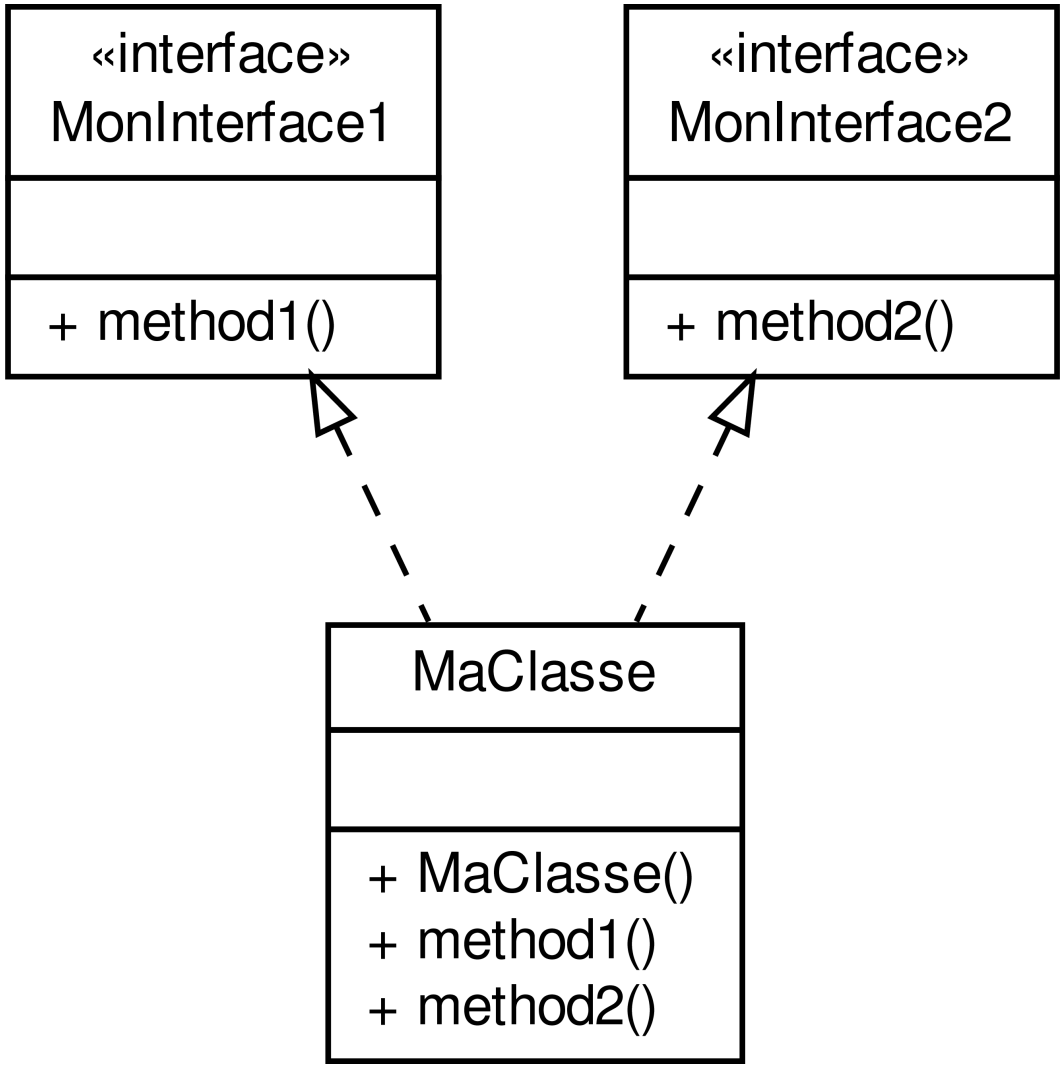
Révision – Extension d'une classe

```
class UneAutreClasse extends UneClasse {

 String nom;
 int prix;

 UneAutreClasse(String nom, int prix) {
  super(nom);
  this.prix = prix;
 }

 void method1(int p1) { System.out.println(nom+" : "+p1); }

 int method3(int p) { return p+prix; }

}
```

Révision – UML – Extensions et implémentations



Révision – UML – Implémentation multiples



Révision – Polymorphisme

Une interface et deux classes qui l'implémentent :

```
interface I { void method(); }

class A implements I {
 void method() { System.out.println("A"); }
}

class B implements I {
 void method() { System.out.println("B"); }
}
```

Laquelle des deux méthodes est appelée ?

```
int test(boolean choix)
 I i; if (choix) i = new A(); else i = new B();
 i.method();
}
```

Révision – Polymorphisme

Une classe qui étend une classe :

```
class A {
 void method() { System.out.println("A"); }
}

class B extends A {
 void method() { System.out.println("B"); }
}
```

Laquelle des deux méthodes est appelée ?

```
int test(boolean choix)
 A a; if (choix) a = new A(); else i = new B();
 a.method();
}
```

Révision – Énumération

```
enum Bidule {
    Truc,
    Machin,
    Chose
}
```

Énumération (la suite)

Il est possible d'associer de l'information aux valeurs :

```
enum Bidule {
    Truc ("Truc", 5),
    Machin ("Machin", 4),
    Chose ("Chose", 8)

    private final double nom;
    private final double prix;

    Bidule(String nom, int prix) {
        this.nom = nom;
        this.prix = prix;
    }
    private double nom()   { return nom; }
    private double prix() { return prix; }
}
```

Énumération (la suite)

Utilisation :

```
public static void main(String[] args) {
    for (Bidule b : Bidule.values())
       System.out.printf("Le prix de %s est %d",
             b.nom(),
             b.prix());
}
```

Révision – Les mots réservés...

Modificateurs de membres ou de variables :

```
public/protected/private
static
final
abstract
synchronized
throws
```

Révision – modificateur **public** et packages

- Par défaut, une classe ou une méthode est **non-publique** : elle n'est accessible que depuis les classes du **même** paquet.
- Une classe ou un membre publics est accessible de n'importe où.
- Pour rendre une classe ou un membre **public** :

```
public class ClassePublique {
  public int proprietePublique;
  public void methodePublique() { }
}
```

- Si fichier contient une classe publique, le nom du fichier doit être formé du nom de la classe suivi de ".java".
- Un fichier ne peut contenir qu'une seule classe publique.

Révision – modificateur **private** et encapsulation

- Un membre **privé** n'est accessible que par les méthodes de la classe qui le contient.
- Pour rendre un membre privé, on utilise le modificateur **private** :

```
public class ClassePublique {
  private int proprietePrivee;
  private void methodePrivee() { }
}
```

- Encapsulation : Tout ce qui participe à l'implémentation des services doit être privé (afin de permettre la modification de l'implémentation des services sans risquer d'impacter les autres classes)

Révision – modificateur **protected** et héritage

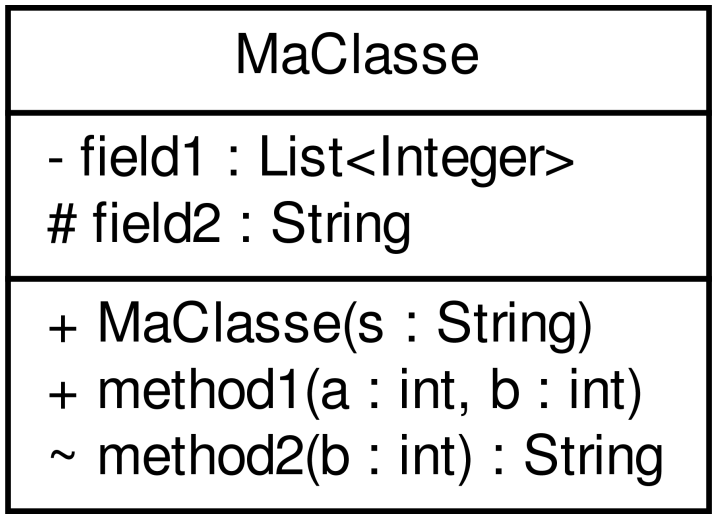
- Un membre **protégé** est accessible depuis :
  - les méthodes de la classe qui le contient ;
  - des méthodes des classes qui étendent la classe qui le contient.

- Pour rendre un membre protégé, on utilise le modificateur **protected** :

```
public class ClassePublique {
 protected int proprieteProtegee;
 protected void methodeProtegee() { }
}
```

- Utilisation possible : Commencer l'implémentation d'un service dans une classe et la terminer dans les classes qui l'étendent .
- Interprétation : Les membres protégés forment une interface entre une classe et les classes qui l'étendent.

Révision – UML – Visibilité



-	:	privé (*private*)
#	:	protégé (*protected*)
+	:	publique (*public*)
~	:	non-publique (*default*)

Révision – Données et méthodes statiques

Les méthodes et des données **statiques** sont directement associées à la classe (et non aux instances de la classe) :

```
class Compteur {
 private static int cpt = 0;
 static void compte() { cpt++; }
 static int valeur() { return cpt; }
}
```

Utilisation :

```
Compteur.compte();
Compteur.compte();
Compteur.compte();
System.out.println(Compteur.valeur());
```

Révision – Classes abstraites

```
abstract class A {
 abstract int prix();
 void afficherPrix() { System.out.println(prix()); };
}

class B extends A {
 int prix() { return 5; }
}

class C extends A {
 int prix() { return 10; }
}
```

Révision – Classes abstraites

Les deux classes B et C héritent de la méthode **afficherPrix()** :

```
B b = new B();
C c = new C();
b.afficherPrix();
c.afficherPrix();
```

Classes abstraites et polymorphisme :

```
A a;
if (choix) a = new B(); else a = new C();
a.afficherPrix();
```

Révision – Les mots réservés...

Dans le code :

```
if/else
for
do/while
switch/case/default
continue/break
try/catch/finally
throw
return
new
null
false
true
this
super
```

Révision – this

```
class Machin {
  private String nom;
  private int prix;

  public Machin(String nom, int prix) {
    this.nom = nom;
    this.prix = prix;
  }

  int ajouterPrix(int prix) {
    return prix + this.prix;
  }
}
```

Révision – super

```
class Truc {
  private String nom;
  public Truc(String nom) { this.nom = nom; }
  public String toString() { return nom; }
}

class Machin extends Truc {
  private int prix;

  public Machin(String nom, int prix) {
    super(nom); this.prix = prix;
  }
  public String toString() {
    return super.toString() + " "+prix;
  }
}
```

Révision – Exceptions

Elles étendent *Exception* et on les jette avec le mot-clé **throw** :

```
throw new MyException("truc");
```

On les captures avec **try/catch** :

```
try {
  ...
} catch(MyException e) {
...
}
```

On les fait remonter avec **throws** :

```
void method() throws MyException {
  ...
}
```

Révision – Les mots réservés...

Les types simples :

```
boolean
byte
char
short
int
long
double
float
void
```

Révision – Les types primitifs

byte	entier	8 bits	-128 à 127	0
short	entier	16 bits	-32768 à 32767	0
int	entier	32 bits	$-2^{31}2^{31}-1$	0
long	entier	64 bits	$-2^{63}2^{63}-1$	0
float	flotant	32 bits		0.0
double	flotant	64 bits		0.0
char	caractère	16 bits	caractères Unicode	\u0000
boolean	boolean	1 bit	false ou true	false

```
int a = 12;
double b = 13.5;
```

Révision – Structures de données Java

Des interfaces :

- **Collection<V>** : Groupe d'éléments
  - **List<V>** : Liste d'éléments ordonnés et accessibles via leur indice
  - **Set<V>** : Ensemble d'éléments uniques
  - **Queue<V>** : Une file d'éléments (FIFO)
  - **Deque<V>** : Une file à deux bouts (FIFO-LIFO)
- **Map<K,V>** : Ensemble de couples clé-valeur.

Il est préférable d'utiliser les interfaces pour typer les variables :

```
List<Integer> l = new ArrayList<Integer>();
```
(code qui utilise l)

car on peut changer la structure de données facilement :

```
List<Integer> l = new LinkedList<Integer>();
```
(code qui utilise l et qui n'a pas à être modifié)

Révision – Structures de données Java

Implémentations de **List<V>** :

- ArrayList<V> : Tableau dont la taille varie dynamiquement.
- LinkedList<V> : Liste chaînée.
- Stack<V> : Pile (mais une implémentation de Deque est préférable).

Implémentations de **Map<K,V>** :

- HashMap : table de hachage
- LinkedHashMap : table de hachage + listes chainées
- TreeMap : arbre rouge-noir (Les éléments doivent être **comparables**)

Implémentations de **Set<V>** :

- HashSet<V> : avec une HashMap.
- LinkedHashSet<V> : avec une LinkedHashMap.
- TreeSet<V> : avec une TreeMap.

Implémentations de **Deque<V>** :

- ArrayDeque<V> : avec un tableau dynamique.
- LinkedList<V> : avec une liste chaînée.

Révision – Boxing et unboxing

Liste d'entiers :

```
List<Integer> liste = new ArrayList<Integer>();
```

Il faut utiliser la classe d'emballage **Integer** associé au type simple **int** :

```
int i = 2;
Integer j = new Integer(i);
p.empiler(j);
Integer k = p.depiler();
int l = j.intValue();
```

Depuis Java 5, autoboxing et auto-unboxing :

```
int i = 2;
p.empiler(i);
int l = p.depiler();
```

(Attention : des allocations sont effectuées sans new dans le code)

Révision – Classes d'emballage

Les classes d'emballage associés aux types simples :

- **Number** : classe abstraite pour les nombres
  - **Byte** : public Byte(byte b)
  - **Short** : public Short(short s)
  - **Integer** : public Integer(int i)
  - **Long** : public Long(long l)
  - **Float** : public Float(float f)
  - **Double** : public Double(double d)
- **Boolean** : public Boolean(boolean b)
- **Character** : public Character(char c)

Création et utilisation d'une liste de booléens :

```
List<Boolean> liste = new ArrayList<Boolean>();
liste.add(true);
```

Révision – Classes génériques

Définition d'une classe générique :

```
class MaClasseGenerique<T> {

  void method1(T e) { ... }
  T method2() { ... }

}
```

Condition sur les paramètres :

```
class MaClasseGenerique<T extends Comparable<T>> { ... }
```

Type " ? super T" et " ? extends T" :

```
class MaClasseGenerique<T> {
  void method1(? super T p1, ? extends T p2) { ... }
}
```

Révision – Les mots réservés...

Les autres mots réservés de Java :

```
assert
goto
native
assert
strictfp
volatile
instanceof
transient
```

Révision – Les cinq principes SOLID

- Single Responsibility Principle (SRP) :
  Une classe ne doit avoir qu'une seule responsabilité
- Open/Closed Principle (OCP) :
  Programme ouvert pour l'extension, fermé à la modification
- Liskov Substitution Principle (LSP) :
  Les sous-types doivent être substituables par leurs types de base
- Interface Segregation Principle (ISP) :
  Éviter les interfaces qui contiennent beaucoup de méthodes
- Dependency Inversion Principle (DIP) :
  Les modules d'un programme doivent être indépendants
  Les modules doivent dépendre d'abstractions

Révision – Patrons de conception

- Les **patrons de conception** décrivent des solutions standards pour répondre aux problèmes rencontrés lors de la conception orientée objet
- Il tendent à respecter les 5 principes SOLID
- Ils sont le plus souvent indépendants du langage de programmation
- Ils ont été formalisés dans le livre du **"Gang of Four"** (Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides – 1995)
- "Les patrons offrent la possibilité de capitaliser un savoir précieux né du savoir-faire d'experts" (Buschmann – 1996)
- Les **anti-patrons** (ou **antipatterns**) sont des erreurs courantes de conception.

Les utilisations de Java...

- Interfaces graphiques (Swing, AWT, SWT...)
- Applications réseaux (Intéraction avec le réseau, RMI, sécurité, ...)
- Site Web (JFace, JSP, DOM, Servlets, GWT...)
- Gestion des formats de données (XML, Excels...)
- Bases de données (JDBC, JDO, Hibernate...)
- Gestion des tests unitaires (JUnit)
- Application sur téléphones mobiles (J2ME, **Android**...)
- ...

Pour utiliser ces technologies, les concepts de POO sont indispensables

Les autres langages à objets...

- **C++** : très utilisé
- **Objective C** : langage utilisé par Apple (IPhone, IPad...)
- **ActionScript** : langage d'Adobe (Flash)
- **C#** : langage de Microsoft (.NET, SilverLight)
- **PHP** : langage très utilisé sur le Web
- **Python**
- **Ruby**

Pour utiliser ces langages, les concepts de POO sont indispensables

La syntaxe change mais les concepts de POO sont les mêmes