Programmation et Conception Orientées Objet

Bertrand Estellon

Département Informatique et Interactions
Aix-Marseille Université

12 septembre 2016

Premier objectif du cours : approfondir la POO

Nous allons revoir et approfondir les notions de POO vues en L2 :

- Classes et instances
- Visibilité, composition et délégation
- Interface et polymorphisme
- Extension et redéfinition de méthode
- Types paramétrés
- Gestion des exceptions

Deuxième objectif du cours : programmer “proprement”

Un programme “propre” :

- respecte les attentes des utilisateurs ;
- est fiable ;
- peut évoluer facilement/rapidement ;
- est compréhensible par tous.

Nous allons voir comment atteindre ces objectifs en utilisant la POO en :

- Séparant et découplant les parties des projets en paquets/classes ;
- Limitant et localisant les modifications lors des évolutions ;
- Faisant en sorte d’écrire du code facilement réutilisable.

Deuxième objectif du cours : programmer “proprement”

Pour programmer “proprement” dans un langage objet, vous devez :

- nommer correctement les éléments que vous manipulez ;
- écrire du code lisible par un autre humain ;
- relire et améliorer votre code une fois qu'il est écrit.

Par conséquent :

- Il est donc impératif de bien maîtriser la programmation ;
- À partir de maintenant, vous devez écrire du code qui fonctionne et qui puisse être lu sans aucun effort ;
- À l'examen, **un code ne vérifiant pas ces conditions sera “faux”**.

Deuxième objectif du cours : programmer “proprement”

Un programme en C que je considère comme mal écrit :

```
struct Rect {
  int w,h;
};

typedef struct Rect Rect;

int compter(Rect** l, int k) {
  int r = 0;
  for (int i = 0; i < k; i++)
    if (l[i]->w==l[i]->h) r++;
  return r;
}
```

Deuxième objectif du cours : programmer “proprement”

Après un petit effort de “refactoring” :

```
struct Rectangle {
  int width, height;
};

typedef struct Rectangle Rectangle;

int isSquare(Rectangle *rectangle) {
  return rectangle->width == rectangle->height;
}

int countSquares(Rectangle* rectangles[], int length) {
  int nbSquares = 0;
  for (int index = 0; index < length; index++)
    if (isSquare(rectangles[index])) nbSquares++;
  return nbSquares;
}
```

Plan du cours

- Classes et instances
- Bonnes pratiques de programmation
- Composition, délégation, interface et polymorphisme
- Extension et redéfinition de méthode
- Surcharge de méthode, types paramétrés
- Exceptions et énumérations
- Diagrammes de classes
- Principes SOLID
- Patrons de conception

Évaluation

- Un partiel (`CC`)
  - Normalement simple
  - Notation très exigeante sur la qualité du code
  - Il est nécessaire de venir en cours pour réussir
- Un examen final (`ET`) :
  - Résoudre des problèmes de conception
  - Notation encore plus exigeante sur la qualité du code
- `MCC = 0.3*CC + 0.7*ET` (avec documents autorisés)

Le langage Java

Le langage Java :

- est un langage de programmation orienté objet
- 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
- performant
- multitâches

Conseil : Apprendre d'autres langages orientés objet (C#, C++, ...)

Mon premier programme Java

Le programme `HelloWorld.java` :

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

Compilation et exécution :

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

Instructions, variables et expressions

Les données sont manipulées à l'aide de variables et d'expressions :

```
class Exemple {
  public static void main(String arg[]) {
    int variable1 = 5;
    int variable2 = 9;
    variable2 = variable2 + variable1;
    variable1 = 2*variable2 - 3;
    System.out.println(variable1); // "25"
    System.out.println(variable2); // "14"
  }
}
```

Comme en C, les instructions sont séparées par des points-virgules.

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

Les constantes :

```
int varInteger = 12;
double varDouble = 13.5;
char varChar = 't';
boolean varBoolean = true;
```

Les tableaux

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

```
int[] arrayOfInt;
double[] arrayOfDouble;
```

Allocation d'un tableau :

```
arrayOfInt = new int[10];
arrayOfDouble = new double[3];
```

Utilisation d'un tableau :

```
arrayOfInt[0] = 5;
arrayOfInt[9] = 10;
arrayOfInt[2] = arrayOfInt[0] + 3*arrayOfInt[9];
arrayOfDouble[2] = arrayOfInt[2] + 3.5;
```

Les tableaux

L'expression `new int[10]` retourne une "référence vers un tableau" :

```
int[] array = new int[10];     // Allocation d'un tableau
int[] array2 = array;          // Copie de la référence
```

Seule la référence est copiée, il n'y a qu'un seul tableau :

```
array[4] = 2; // Affectation d'une valeur à la 5ème case
System.out.println(array2[4]); // "5"
```

Taille du tableau :

```
int[] array = new int[10]; // Allocation d'un tableau
int length = array.length; // Récupération de la taille
System.out.println(length);
/* ou directement */ System.out.println(array.length);
```

Références et Garbage Collector

Une variable de type “référence” peut valoir `null` :

```
int[] array = null;
// La variable "array" contient la référence "null".
// Elle pointe vers aucune instance.
```

Important : Une référence contient soit `null` soit la référence d'un tableau ou d'un objet compatible avec le type de la variable.

Le **Garbage collector** peut supprimer les éléments devenus innaccessibles :

```
int[] array = new int[10];
array[3] = 6;
...
array = null;
// La mémoire utilisée pour le tableau peut être libérée
```

Les tableaux à plusieurs indices

Déclaration :

```
int[][] matrix;
```

Création :

```
matrix = new int[10][];
for (int row = 0; row < matrix.length; row++)
  matrix[row] = new int[5];
/* ou directement */
matrix = new int[10][5];
```

Tableaux multidimensionnels “non rectangulaires” :

```
array = new int[10][];
for (int row = 0; row < array.length; row++)
  array[row] = new int[row+1];
```

Des tableaux aux objets

Un tableau :

- peut être **alloué** (ou construit) : `array = new int[5]`
- est **structuré** : il est constitué d'un ensemble de cases
- possède un **état** : la valeur de ses cases
- possède une **interface**

L'**interface** d'un tableau :

- affecter une valeur à une case : array[index] = value ;
- consulter la valeur d'une case : array[index] ;
- récupérer sa longueur : array.length ;

Les types "référence vers des tableaux" sont définis dans le langage

Les objets

Un **objet** :

- peut être **construit**
- est **structuré** : il est constitué d'un ensemble de propriétés
- possède un **état** : la valeur de ses propriétés
- possède une **interface** : l'ensemble des méthodes et propriétés accessibles par les utilisateurs de l'objet

Dans les langages orientés objet, une **classe** (d'objets) définit :

- une façon de construire des objets (**constructeur**)
- la structure des objets de la classe (**propriétés**)
- le comportement des objets de la classe (**méthodes**)
- l'interface des objets de la classe (**méth. et prop. non-privées**)
- un type "référence vers des objets de cette classe"

Classes et instances

Le mot clé `class` permet de définir une classe :

```
class Counter {
  /* Définition des propriétés et des méthodes */
}
```

On peut ensuite définir une variable de type *"référence vers un Counter"* :

```
Counter counter;
```

Une classe définit également un "moule" pour fabriquer des objets.

La création d'une instance s'effectue avec le mot clé `new` :

```
Counter counter = new Counter();
```

La variable `counter` contient alors une référence vers une instance de la classe `Counter`.

Les propriétés

Les propriétés décrivent la structure de données de la classe :

```
class Counter { int position; int step; }
```

On accède aux propriétés avec l'opérateur "." (point) :

```
Counter counter = new Counter();
counter.position = 12; counter.step = 2;
counter.position += counter.step;
```

Chaque instance possède son propre état :

```
Counter counter1 = new Counter();
Counter counter2 = new Counter();
counter1.position = 10; counter2.position = 20;
if (counter1.position!=counter2.position)
  System.out.println("différent");
```

Références et Garbage Collector

Comme pour les tableaux, on peut copier les références :

```
Counter counter1 = new Counter();
Counter counter2 = counter1;
counter1.position = 10;
if (counter1.position==counter2.position)
  System.out.println("égalité");
```

Important : Une référence contient soit `null` soit la référence d'un objet compatible avec le type de la variable.

Le **Garbage collector** peut supprimer les objets devenus innaccessibles :

```
Counter c = new Counter(); ...; c = null;
// La mémoire utilisée pour l'objet peut être libérée
```

Les méthodes

Méthodes qui modifient l'objet :

```
class Counter {
  int position, step;

  void init(int initPos, int s) {
    position = initPos; step = s;
  }

  void count() { position += step; }
}
```

On invoque les méthodes avec l'opérateur "." (point) :

```
Counter counter = new Counter();
counter.init(4, 8+12); counter.count(); counter.count();
System.out.println(counter.position);
```

Les méthodes

Méthode permettant de consulter l'état de l'objet :

```
class Counter {

   /* Code du slide précédent. */

   int getPosition() { return position; }
}
```

Exemple d'utilisation :

```
Counter counter = new Counter();
counter.init(4, 8+12); counter.count(); counter.count();
System.out.println(counter.getPosition());
```

Les méthodes

Méthode qui modifie et retourne une valeur :

```
class Counter {

   /* Code de la méthode init. */

   int count() { position += step; return position; }
}
```

Exemple d'utilisation :

```
Counter counter = new Counter();
counter.init(4, 8+12);
System.out.println(counter.count());
System.out.println(counter.count());
```

Comparaison de références

Il est possible de comparer deux références :

```
public class Test {
  public static void main(String arg[]) {
    Counter counter1 = new Counter();
    Counter counter2 = counter1;

    if (counter1==counter2) System.out.println("==");

    Counter counter3 = new Counter();

    if (counter1!=counter3) System.out.println("!=");
    if (counter2!=counter3) System.out.println("!=");
  }
}
```

Déclaration d'un constructeur

Les constructeurs permettent d'agréger l'allocation et l'initialisation :

```
class Counter {
 int position, step;

 Counter(int p, int s) { position = p; step = s; }

 Counter(int p) { position = p; step = 1; }

 /* Méthodes count() et getPosition() */
}
```

Exemple d'utilisation :

```
Counter c1 = new Counter(2,10);
Counter c2 = new Counter(22);
```

Constructeur par défaut

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

```
public class Counter  {
  int position = 10, step = 1;
}
```

Ce code est équivalent à celui-ci :

```
public class Counter {
  int position, step;

  public Counter() {
    position = 10; step = 1;
  }
}
```

Le mot-clé this

Le mot-clé `this` fait référence à l'instance en construction ou à l'instance qui a permis d'invoquer la méthode en cours d'exécution :

```
class Counter {
 int position, step;

 Counter(int position, int step) {
   this.position = position; this.step = step;
 }

 void count() {
   int position = this.position;
   position += step; this.position = position;
 }
}
```

Le mot-clé this

Le mot-clé `this` permet également d'appeler un constructeur dans un constructeur de façon à ne dupliquer du code :

```
class Counter {
 int position, step;

 Counter(int position, int step) {
   this.position = position; this.step = step;
 }

 Counter(int position) {
   this(position, 1);
 }

 /* Autres méthodes. */
}
```

Surcharge de méthode

Dans une classe, deux méthodes peuvent avoir le même nom. Java doit pouvoir décider de la méthode à exécuter en fonction des paramètres :

```
class Value {
  int value = 0;

  void set()           { value = 0; }
  void set(int value)     { this.value = value; }
  void set(double vallue) { this.value = (int)value; }
}
```

Exemple d'utilisation :

```
Value value = new Value();
value.set();     // Première méthode
value.set(2);    // Deuxième méthode
value.set(2.5); // Troisième méthode
```

Propriétés et méthodes statiques

Les propriétés et des méthodes statiques sont directement associées à la classe et non aux instances de la classe :

```
class Counter {
  static int step;
  int position;

  Counter(int position) { this.position = position; }

  static void setStep(int step) {
    Counter.step = step;
  }

  void count() { position += step; }
}
```

Données et méthodes statiques

Comme les propriétés et méthodes statiques sont associées à la classe, il n'est pas nécessaire de posséder une instance pour les utiliser :

```
public class Test {
  public static void main(String[] arg) {
    Counter.setStep(3);
    Counter counter1 = new Counter(2);
    Counter counter2 = new Counter(3);
    counter1.count(); counter2.count();
    System.out.println(counter1.position); // → 5
    System.out.println(counter2.position); // → 6
    Counter.setStep(4); counter1.count(); counter2.count();
    System.out.println(counter1.position); // → 9
    System.out.println(counter2.position); // → 10
  }
}
```

Données et méthodes statiques

Une méthode statique ne peut utiliser que :

- des propriétés 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 Counter {
  /* ... */
  static void setStep(int step) {
    Counter.step = step;
  }
  /* ... */
}
```

Chaînes de caractères

Trois classes permettent de gérer les chaînes de caractères :

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

Déclaration et création :

```
String hello = "Hello";
String world = "World";
```

Concaténation :

```
String helloWorld = hello + " "+ world + " ! ";
int integer = 13;
String helloWorld1213 = hello + " " + world
                        + " " + 12 + " " + integer;
```

Chaînes de caractères

Affichage :

```
System.out.print(helloWorld);   // affiche "Hello World !"
System.out.println(helloWorld); // affiche "Hello World !"
                                // 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"
```

Point d'entrée et arguments

Le programme *Echo.java* :

```
public class Echo {
  public static void main(String[] args) {
    for (int i = 0; i < args.length; i++)
     System.out.println(args[i]);
  }
}
```

```
$ javac Echo.java
$ ls
Echo.java Echo.class
$ java Echo toto aaa
toto
aaa
```

Programmer “proprement” : nommage

Un nommage correct des méthodes et variables améliore grandement la lisibilité et la maintenabilité d'un code.

Êtes-vous capable de dire rapidement ce que fait cette classe ?

```
class Structure {
  private int s[] = new int[100];
  private int t = 0;

  public void end(int i) { s[t] = i; t++; }

  public int top() { t--; return s[t]; }

  public bool test() { return t == 0; }

}
```

Programmer “proprement” : nommage

Le nommage des variables, attributs, classes et méthodes est important :

```
class Stack {
  private int stack[] = new int[100];
  private int size = 0;

  public void push(int i) {
    stack[size] = i; size++;
  }

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

  public bool isEmpty() {
    return size == 0;
  }
}
```

Conventions de nommage en Java

Conventions officielles de nommage en Java :

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

```
class Stack {
  private int stack[] = new int[100];
  private int size = 0;

  public void pushInteger(int i) { stack[size] = i; size++; }
  public int popInteger() { size--; return stack[size]; }
}
```

Un programme “propre”

Clean Code (Robert C. Martin)

Un programme “propre” :

- respecte les attentes des utilisateurs ;
- est fiable ;
- peut évoluer facilement/rapidement ;
- est compréhensible.

En résumé :

- Un programme informatique est de qualité si l’effort nécessaire à l’ajout d’une nouvelle fonctionnalité par un développeur extérieur au projet est faible.

Pourquoi ?

- Pour programmer les fonctionnalités les unes après les autres
- Pour ajouter des fonctionnalités à moindre coût
- Pour que les programmes soient utilisables plus longtemps
- Pour valoriser les codes écrits par les développeurs (car réutilisables)
- Pour effectuer des tests à toutes les étapes du développement
- Pour que les développeurs soient heureux de travailler

Nommage

Êtes-vous capable de dire rapidement ce que fait cette classe ?

```
class Cream {
  private int donut = 0;
  public void pt() { donut++; }
  public int t() { return donut; }
}
```

Le même programme avec un meilleur nommage des variables :

```
class Counter {
  private int count = 0;
  public void increment() { count++; }
  public int value() { return count; }
}
```

Nommage

Pourquoi ce programme est mal écrit ?

```
List<Rectangle> get(List<Rectangle> list) {
  List<Rectangle> list2 = new ArrayList<Rectangle>();
  for (Rectangle x : list)
    if (x.w == x.h) list2.add(x);
  return list2;
}
```

```
class Rectangle {
  public int w, h;
}
```

Nommage

Après un renommage :

```
List<Rectangle> findSquares(List<Rectangle> rectangles) {
  List<Rectangle> squares = new ArrayList<Rectangle>();
  for (Rectangle rectangle : rectangles)
    if (rectangle.isSquare()) squares.add(rectangle);
  return squares;
}
```

```
class Rectangle {
  private int width, height;
  /* ... */
  boolean isSquare() { return width == height; }
}
```

Nommage des méthodes

```
class Rectangle {
  private Point point1, point2;

  public Rectangle(Point point1, Point point2) {
    this.point1 = point1; this.point2 = point2;
  }

  public translate(int dx, int dy) {
    point1.translate(dx, dy); point2.translate(dx, dy);
  }

  int width() { return Math.abs(point1.x - point2.x); }
  int height() { return Math.abs(point1.y - point2.y); }
  boolean isSquare() { return width() == height(); }
}
```

Nommage des méthodes

- Méthodes procédurales → groupe verbal à l'infinitif :

```
public translate(int dx, int dy) {
  point1.translate(dx, dy);
  point2.translate(dx, dy);
}
```

- Expressions non booléennes → groupe nominal :

```
int width() { return Math.abs(point1.x - point2.x); }
int height() { return Math.abs(point1.y - point2.y); }
```

- Expressions booléennes → groupe verbal au présent :

```
boolean isSquare() { return width() == height(); }
boolean contains(Point point) { ... }
/* if (rectangle.isSquare()) ... */
```

Pourquoi découper et nommer...

Quelle est la sémantique du texte suivant ?

L'animal vertébré vivipare caractérisé par la présence de mamelles, aux organes d'audition et d'équilibration à l'étendue supérieure à la moyenne dans le sens de la longueur, qui se nourrit en grignotant avec ses incisives, et qui se multiplie avec une rapidité d'intensité forte, fait descendre par le gosier, postérieurement à une réduction en petites parcelles avec les dents, un être vivant appartenant au règne végétal cultivé pour l'usage culinaire, muni de pédicelles partant en faisceau depuis son pédoncule, et dont la partie souterraine permettant la fixation au sol est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer...

Quelle est la sémantique du texte suivant ?

L'animal vertébré vivipare caractérisé par la présence de mamelles,
qui se nourrit en grignotant avec ses incisives,
et qui se multiplie avec une rapidité d'intensité forte,
aux organes d'audition et d'équilibration
à l'étendue supérieure à la moyenne dans le sens de la longueur,
fait descendre par le gosier,
postérieurement à
une réduction en petites parcelles avec les dents,
un être vivant appartenant au règne végétal
cultivé pour l'usage culinaire,
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer...

Quelle est la sémantique du texte suivant ?

Le mammifère
qui se nourrit en grignotant avec ses incisives,
et qui se multiplie avec une rapidité d'intensité forte,
aux organes d'audition et d'équilibration
à l'étendue supérieure à la moyenne dans le sens de la longueur,
fait descendre par le gosier,
postérieurement à
une réduction en petites parcelles avec les dents,
un être vivant appartenant au règne végétal
cultivé pour l'usage culinaire,
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer...

Quelle est la sémantique du texte suivant ?

Le mammifère
rongeur
et qui se multiplie avec une rapidité d'intensité forte,
aux organes d'audition et d'équilibration
à l'étendue supérieure à la moyenne dans le sens de la longueur,
fait descendre par le gosier,
postérieurement à
une réduction en petites parcelles avec les dents,
un être vivant appartenant au règne végétal
cultivé pour l'usage culinaire,
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer...

Quelle est la sémantique du texte suivant ?

Le mammifère
rongeur
très prolifique
aux organes d'audition et d'équilibration
à l'étendue supérieure à la moyenne dans le sens de la longueur,
fait descendre par le gosier,
postérieurement à
une réduction en petites parcelles avec les dents,
un être vivant appartenant au règne végétal
cultivé pour l'usage culinaire,
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer...

Quelle est la sémantique du texte suivant ?

Le mammifère
rongeur
très prolifique
aux oreilles
à l'étendue supérieure à la moyenne dans le sens de la longueur,
fait descendre par le gosier,
postérieurement à
une réduction en petites parcelles avec les dents,
un être vivant appartenant au règne végétal
cultivé pour l'usage culinaire,
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer...

Quelle est la sémantique du texte suivant ?

Le mammifère
rongeur
très prolifique
aux oreilles
longues
fait descendre par le gosier,
postérieurement à
une réduction en petites parcelles avec les dents,
un être vivant appartenant au règne végétal
cultivé pour l'usage culinaire,
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer...

Quelle est la sémantique du texte suivant ?

Le mammifère
rongeur
très prolifique
aux oreilles
longues
avale
postérieurement à
une réduction en petites parcelles avec les dents,
un être vivant appartenant au règne végétal
cultivé pour l'usage culinaire,
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer...

Quelle est la sémantique du texte suivant ?

Le mammifère
rongeur
très prolifique
aux oreilles
longues
avale
après
une réduction en petites parcelles avec les dents,
un être vivant appartenant au règne végétal
cultivé pour l'usage culinaire,
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer...

Quelle est la sémantique du texte suivant ?

Le mammifère
rongeur
très prolifique
aux oreilles
longues
avale
après
mâchage
un être vivant appartenant au règne végétal
cultivé pour l'usage culinaire,
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer...

Quelle est la sémantique du texte suivant ?

Le mammifère
rongeur
très prolifique
aux oreilles
longues
avale
après
mâchage
une plante
cultivé pour l'usage culinaire,
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer...

Quelle est la sémantique du texte suivant ?

Le mammifère
rongeur
très prolifique
aux oreilles
longues
avale
après
mâchage
une plante
potagère
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer...

Quelle est la sémantique du texte suivant ?

Le mammifère
rongeur
très prolifique
aux oreilles
longues
avale
après
mâchage
une plante
potagère
ombellifère
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer...

Quelle est la sémantique du texte suivant ?

Le mammifère
rongeur
très prolifique
aux oreilles
longues
avale
après
mâchage
une plante
potagère
ombellifère
à racine

est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer...

Quelle est la sémantique du texte suivant ?

Le mammifère
rongeur
très prolifique
aux oreilles
longues
avale
après
mâchage
une plante
potagère
ombellifère
à racine
rouge.

Pourquoi découper et nommer...

Quelle est la sémantique du texte suivant ?

Le mammifère rongeur très prolifique aux longues oreilles
avale après mâchage
une plante potagère ombellifère à racine rouge.

Pourquoi découper et nommer...

Quelle est la sémantique du texte suivant ?

Le lapin
avale après mâchage
une plante potagère ombellifère à racine rouge.

Pourquoi découper et nommer...

Quelle est la sémantique du texte suivant ?

Le lapin

mange

une plante potagère ombellifère à racine rouge.

Pourquoi découper et nommer...

Quelle est la sémantique du texte suivant ?

Le lapin
mange
une carotte.

Pourquoi découper et nommer...

Quelle est la sémantique du texte suivant ?

Le lapin mange une carotte.

Un programme facile à comprendre ?

```
public static Formula analyze(String[] elements) {
  Stack<Formula> stack = new Stack<Formula>();
  Formula a, b, c;
  for (String e : elements) {
    switch (e) {
     case "+":
      a = stack.pop(); b = stack.pop();
      c = new Sum(a, b); stack.push(c);
      break;
     case "*":
      a = stack.pop(); b = stack.pop();
      c = new Product(a, b); stack.push(c);
      break;
      /* ... */
```

Un programme facile à comprendre ?

```
    default:
        int v = Integer.parseInt(e);
        a = new Constant(v));
        stack.push(a);
        break;
    }
  }
  return stack.pop();
}
```

Ce code est illisible car :

- Programme mal découpé : la méthode a trop de responsabilités ;
- Problème dans le nommage des identifiants ;
- La variable “a” n’a pas toujours le même rôle.

Amélioration de la lisibilité du code

Analyser ⇒ traiter chacun des éléments :

```
Formula analyze(String[] elements) {
  Stack<Formula> stack = new Stack<Formula>();
  for (String element : elements)
    processElement(stack, element);
  return stack.pop();
}
```

Amélioration de la lisibilité du code

Traiter un élément ⇒ exécuter un traitement en fonction de l'élément :

```
void processElement(Stack<Formula> stack, String element) {
  switch (element) {
    case "+": processSum(stack);                  break;
    case "*": processProduct(stack);              break;
    default : processInteger(stack, element); break;
  }
}
```

Amélioration de la lisibilité du code

Traiter une somme ⇒ dépiler deux formules et empiler la somme :

```
void processSum(Stack<Formula> stack) {
  Formula right = stack.pop();
  Formula left = stack.pop();
  Formula sum = new Sum(left, right);
  stack.push(sum);
}
```

Traiter un produit ⇒ dépiler deux formules et empiler le produit :

```
void processProduct(Stack<Formula> stack) {
  Formula right = stack.pop();
  Formula left = stack.pop();
  Formula product = new Product(left, right);
  stack.push(product);
}
```

Amélioration de la lisibilité du code

Traiter un entier ⇒ empiler une constante contenant l'entier :

```
void processInteger(Stack<Formula> stack, String element) {
  int value = Integer.parseInt(element);
  Formula constant = new Constant(value);
  stack.push(constant);
}
```

Un programme facile à comprendre ?

```
public class BadLinkedList<T> {
  private class Node { T element;  Node next; }

  private Node head;

  public BadLinkedList() { head = null; }

  public void addFirst(T element) {
    Node head = this.head;
    this.head = new Node();
    this.head.element = element;
    this.head.next = head;
  }
  /* ... */
}
```

Un programme facile à comprendre ?

```
public class BadLinkedList<T> {
  /* ... */

  public void addLast(T element) {
    Node node = head;
    if (node == null) {
      head = new Node(); head.element = element;
    } else {
      while (node.next!=null) node = node.next;
      node.next = new Node();
      node.next.element = element;
    }
  }
}
```

Amélioration de la lisibilité du code

```
public class LinkedList<T> {
  private class Node {
    /* ... */
    Node(T element, Node next) {
      this.element = element; this.next = next;
    }
  }

  private Node head;

  public void addFirst(T element) {
    head = new Node(element, head);
  }
  /* ... */
```

Amélioration de la lisibilité du code

Méthode pour insérer un élément à la fin de la liste :

- Si la liste est vide, cela revient à ajouter l'élément en tête ;
- Sinon placer l'élément dans un nœud ;
- Insérer ce nœud après le dernier nœud de la liste.

```
public class LinkedList<T> {
  /* ... */
  public void addLast(T element) {
    if (isEmpty()) { addFirst(element); return; }
    Node tail = tail();
    Node node = new Node(element);
    link(tail, node);
  }
  /* ... */
}
```

Amélioration de la lisibilité du code

Méthode pour calculer le dernier nœud de la liste :

- Si la liste est vide, il n'y a pas de dernier nœud ;
- Partir de la tête de la liste ;
- Tant qu'un nœud suivant existe, avancer sur le suivant ;
- Retourner le nœud.

```
public class LinkedList<T> {
  private Node tail() {
    if (isEmpty()) return null;
    Node node = head;
    while (node.hasNext()) node = node.next;
    return node;
  }
}
```

Amélioration de la lisibilité du code

Quelques méthodes pour rendre le code plus lisible :

```
public class LinkedList<T> {

  private class Node {
    /* ... */
    boolean hasNext() { return next!=null; }
  }

  public boolean isEmpty() { return head==null; }

  private void link(Node first, Node second) {
    first.next = second;
  }
}
```

Les méthodes

Une méthode ne doit faire qu’une chose :

```
class User {
  private boolean authenticated;
  private String password;

  public boolean authenticate(String password) {
    if (password.equals(this.password)) {
      authenticated = true;
      return true;
    }
    return false;
  }
}
```

Les méthodes

Ne pas mentir :

```
class User {
  private boolean authenticated;
  private String password;

  public boolean checkPassword(String password) {
    if (password.equals(this.password)) {
      authenticated = true;
      return true;
    }
    return false;
  }
}
```

Les arguments des méthodes

```
void drawCircle(int x, int y, int radius) {
  /* ... */
}
```

Il est toujours préférable d'essayer de limiter le nombre d'arguments :

```
void drawCircle(Point center, int radius) {
  /* ... */
}
```

Les arguments des méthodes

Un mauvais code :

```
List<Integer> copy(List<Integer> list, boolean onlyEven) {
  List<Integer> result = new List<Integer>();
  for (int i : list)
    if (!onlyEven || i%2==0) result.add(i);
  return result;
}
```

Drapeaux pour préciser le comportement → mauvaise idée

Les arguments des méthodes

Il est préférable d'écrire deux méthodes :

```
List<Integer> copy(List<Integer> list) {
  List<Integer> result = new List<Integer>();
  for (int i : list) result.add(i);
  return result;
}

List<Integer> copyOnlyEven(List<Integer> list) {
  List<Integer> result = new List<Integer>();
  for (int i : list)
    if (i%2==0) result.add(i);
  return result;
}
```

Les boucles

Un code mal écrit :

```
boolean addToList(String[] source, int index,
                  List<String> destination) {
  if (index < 0 || index >= source.length) return false;
  destination.add(source[index]);
  return true;
}

public void copy(String[] source,
                 List<String> destination) {
  int index = 0;
  while (addToList(source, index, destination)) index++;
}
```

- Il faut essayer de ne pas mélanger condition de boucle et action.

Les boucles

L'action de la boucle :

```
void addToList(String[] source, int index,
                    List<String> destination) {
  destination.add(source[index]);
}
```

Le test de la boucle :

```
boolean indexIsValid(String[] source, int index) {
  return index >= 0 && index < source.length;
}
```

Les boucles

La boucle réécrite en positionnant correctement le test et l'action :

```
public void copy(String[] source,
                 List<String> destination) {
  for (int index = 0;
       indexIsValid(source, index);
       index++)
    addToList(source, index, destination);
}
```

Évidemment dans ce cas, on écrit directement :

```
public void copy(String[] source,
                 List<String> destination) {
  for (int index = 0; index < source.length; index++)
    destination.add(source[index]);
```

Les boucles

Un code qui mélange la condition d'arrêt et l'action à réaliser :

```
boolean contains(int[] array, int element) {
  boolean ok = false;
  for (int i = 0; !ok && i < array.length; i++)
    if (array[i] == element) ok = true;
  return ok;
}
```

La variable ok ne fait pas partie de la condition de boucle donc :

```
boolean contains(int[] array, int element) {
  for (int i = 0; i < array.length; i++)
    if (array[i] == element) return true;
  return false;
}
```

Les boucles

Un code qui mélange la condition d'arrêt et l'action à réaliser :

```
void copyToStop(Node head, List<String> destination) {
  Node node = head;
  boolean ok = false;
  while (node != null && !ok) {
    String element = node.element;
    if (!element.equals("stop")) destination.add(element);
    else ok = true;
    node = node.next;
  }
}
```

Les boucles

Utilisation du `break` pour simplifier le code :

```
void copyToStop(Node head, List<String> destination) {
  for (Node node = head; node != null; node = node.next) {
    String element = node.element;
    if (element.equals("stop")) break; // or return
    destination.add(element);
  }
}
```

Remarque : Si elle est bien écrite, une méthode courte avec plusieurs points de sortie est plus facile à lire qu'une méthode qui utilise des variables booléennes “artificielles” pour sortir des boucles.

Refactoring : une démarche littéraire

Fonctions de refactoring :

- renommage “intelligent” des variables, méthodes, classes, etc.
- déplacement de code, de méthode, de classe, etc.
- Introduction d'un nouveau paramètre à une méthode
- Transformation d'une variable en propriété...

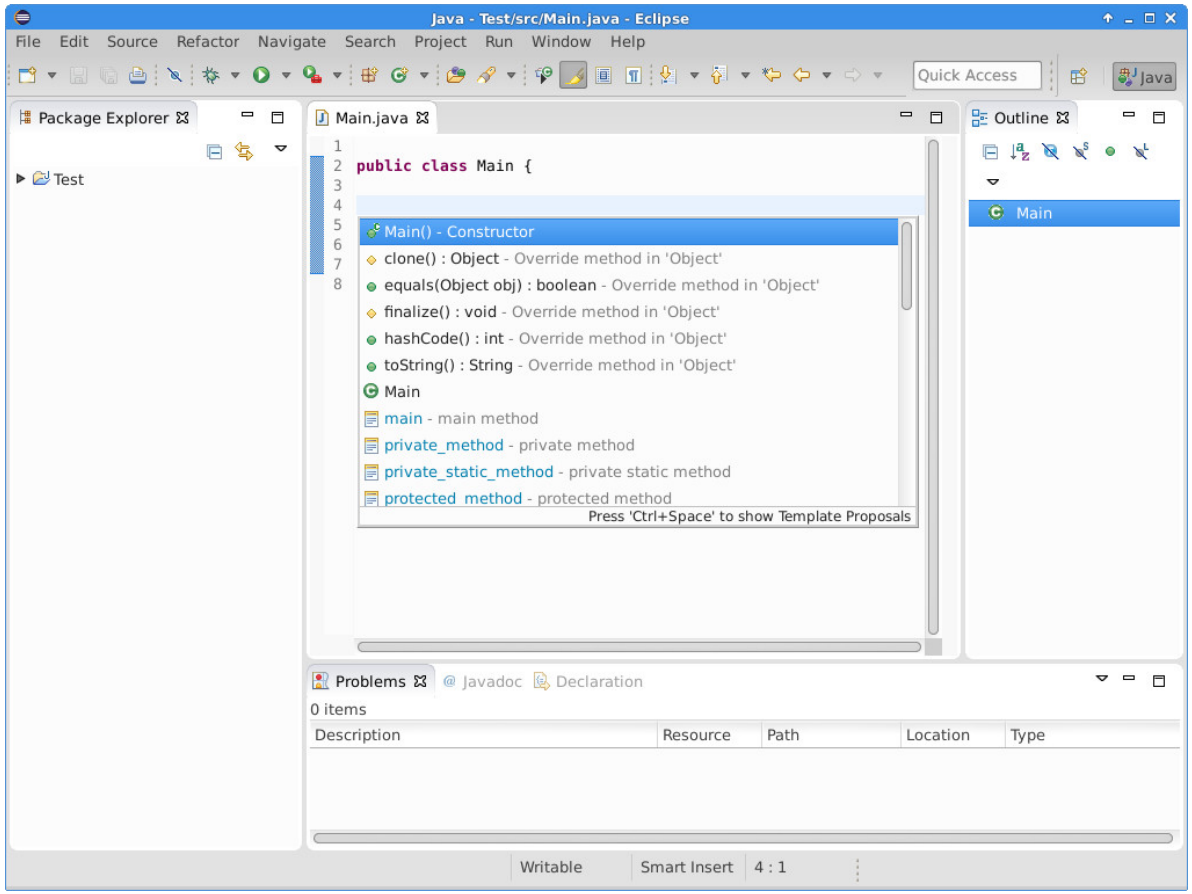
Une fois que votre code fonctionne, vous devez :

- revoir le nom de chacune des variables, méthodes, classes, etc.
- rendre le code le plus agréable à lire en le réorganisant

Remarque : Un programme est un document (au même titre qu'une lettre ou qu'un courriel) $\Rightarrow$ relire et soigner le fond et la forme.

Utilisation d’Eclipse

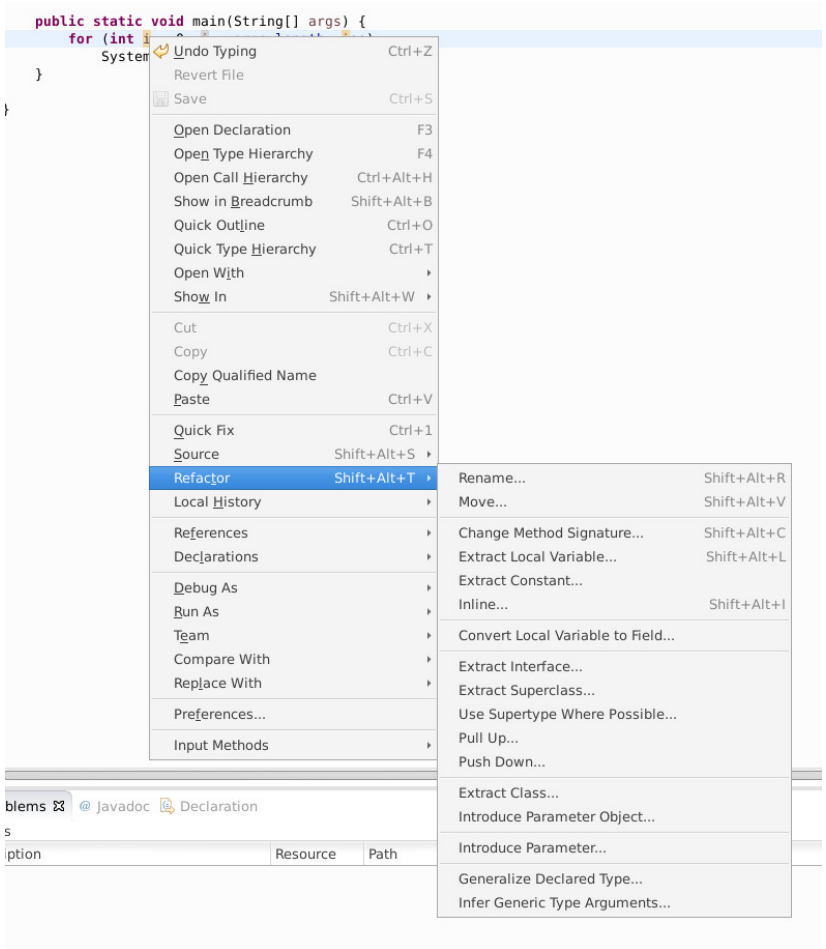
Auto-complétion (Ctrl+Espace) :

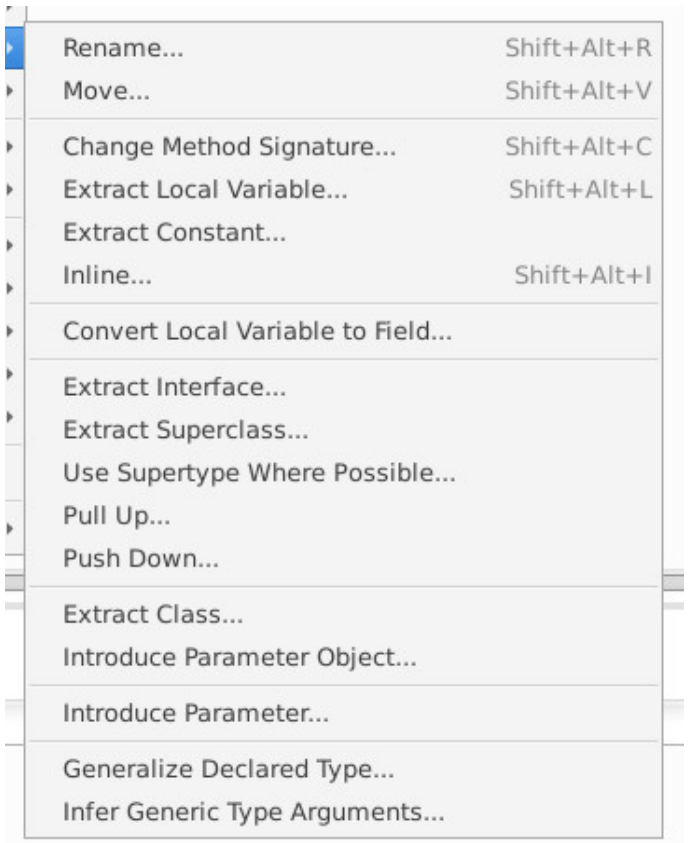


Ré-indentation du code : Shift+Ctrl+F (pour Format)

Utilisation d'Eclipse

Fonction de refactoring :





Les commentaires **inutiles**

```
String get(String[] source, int index) {
  // Teste si l'index est dans les limites du tableau.
  if (index < 0 || index >= source.length) return null;
  return source[index];
}
```

Si un commentaire semble nécessaire, le remplacer par une méthode :

```
boolean indexIsInBounds(String[] source, int index) {
  return index >= 0 && index < source.length;
}

String get(String[] source, int index) {
  if (!indexIsInBounds(source, index)) return null;
  return source[index];
}
```

Les commentaires **inutiles**

Les commentaires se désynchronisent souvent du code :

```
/* doit toujours retourner true. */
boolean isAvailable() {
  return true;
}
```

risque de devenir un jour :

```
/* doit toujours retourner true. */
boolean isAvailable() {
  return false;
}
```

Les commentaires **inutiles**

Un autre exemple de commentaires inutiles :

```
boolean printStrings(String[] elements) {
  // On affiche tous les éléments du tableau
  for (String element : elements) {
    // On affiche un élément du tableau
    System.out.println(element);
  }
}
```

Les commentaires **inutiles**

Des commentaires qui peuvent sembler utiles :

```
/* une méthode qui retourne que les carrés : */
List<Rectangle> get(List<Rectangle> list) {
  /* le résultat sera stocké dans cette liste : */
  List<Rectangle> list2 = new ArrayList<Rectangle>();
  for (Rectangle x : list)
    if (x.w == x.h /* un carré ? */) list2.add(x);
  return list2;
}

class Rectangle {
  public int w; /* largeur */
  public int h; /* hauteur */
}
```

Les commentaires **inutiles**

En traduisant les commentaires en code Java :

```
List<Rectangle> findSquares(List<Rectangle> rectangles) {
  List<Rectangle> squares = new ArrayList<Rectangle>();
  for (Rectangle rectangle : rectangles)
    if (rectangle.isSquare()) squares.add(rectangle);
  return squares;
}
```

```
class Rectangle {
  private int width, height;

  boolean isSquare() { return width == height; }
}
```

Les commentaires **utiles**

Des commentaires pour décrire les tâches à réaliser peuvent être utiles :

```
void processElement(Stack<Formula> stack, String element) {
  // TODO : prendre en compte les signes '-' et '/'
  switch (element) {
    case "+": processSum(stack);                  break;
    case "*": processProduct(stack);              break;
    default : processInteger(stack, element); break;
  }
}
```

1 items

✓	!	Description	Resource	Path	Location	Type
		TODO : prendre en compte les signes '-' et '/'	Analyser.java	/Test/src	line 4	Java Task

Les commentaires **utiles**

Documentation ou spécification du comportement d'une méthode :

```
/**
 * Returns <tt>true</tt> if this list contains the specified element.
 * More formally, returns <tt>true</tt> if and only if this list
 * contains at least one element <tt>e</tt> such that
 * <tt>(o==null ? e==null : o.equals(e))</tt>.
 *
 * @param o element whose presence in this list is to be tested
 * @return <tt>true</tt> if this list contains the specified element
 */
public boolean contains(Object o) {
  return indexOf(o) >= 0;
}
```

(Exemple provenant de l'implémentation de la classe ArrayList de Java)

JavaDoc

JavaDoc permet de génèrer automatiquement une documentation du code à partir de commentaires associés aux classes, méthodes, propriétés, etc.

La documentation contient :

- Une description des membres (publics et protégés) des classes
- Une description des classes, interfaces, etc.
- Des liens permettant de naviguer entre les classes
- Des informations sur les implémentations et extensions

La documentation :

- peut être générée au format HTML
- est visible dans Eclipse ou NetBeans lors de l’auto-complétion

JavaDoc

Overview Package Class Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS | FRAMES NO FRAMES All Classes

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

Java™ Platform Standard Ed. 6

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

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.

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

PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes

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

Java™ Platform Standard Ed. 6

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.

JavaDoc

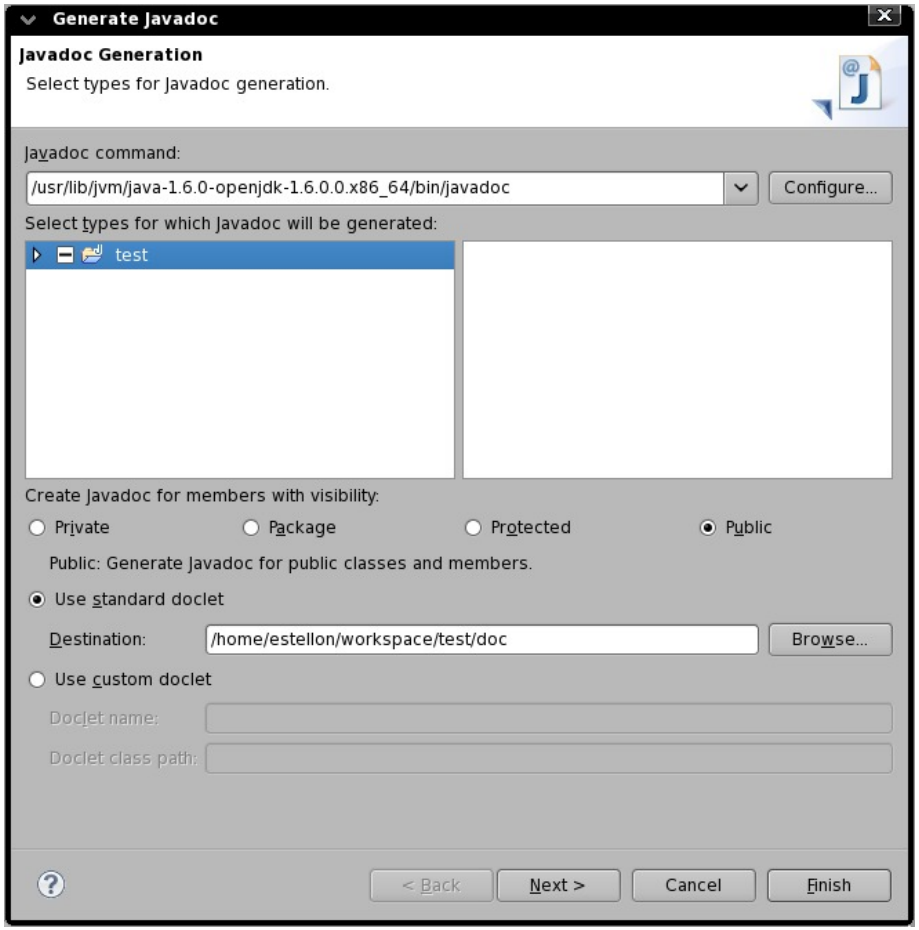
@author	1.0	pour préciser l'auteur de la fonctionnalité
@docRoot	1.3	chemin relatif qui mène à la racine de la doc.
@deprecated	1.0	indique que le membre ou la classe est dépréciée
@link	1.2	permet de créer un lien
@param	1.0	pour décrire un paramètre
@return	1.0	pour décrire la valeur de retour
@see	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)

JavaDoc

```java
/**
 *
 * The {@code Byte} class wraps a value of primitive type {@code byte}
 * in an object.  An object of type {@code Byte} contains a single
 * field whose type is {@code byte}.
 *
 * <p>In addition, this class provides several methods for converting
 * a {@code byte} to a {@code String} and a {@code String} to a {@code
 * byte}, as well as other constants and methods useful when dealing
 * with a {@code byte}.
 *
 * @author  Nakul Saraiya
 * @author  Joseph D. Darcy
 * @see     java.lang.Number
 * @since   JDK1.1
 */
public final class Byte extends Number implements Comparable<Byte> {
 /* ... */
}
```

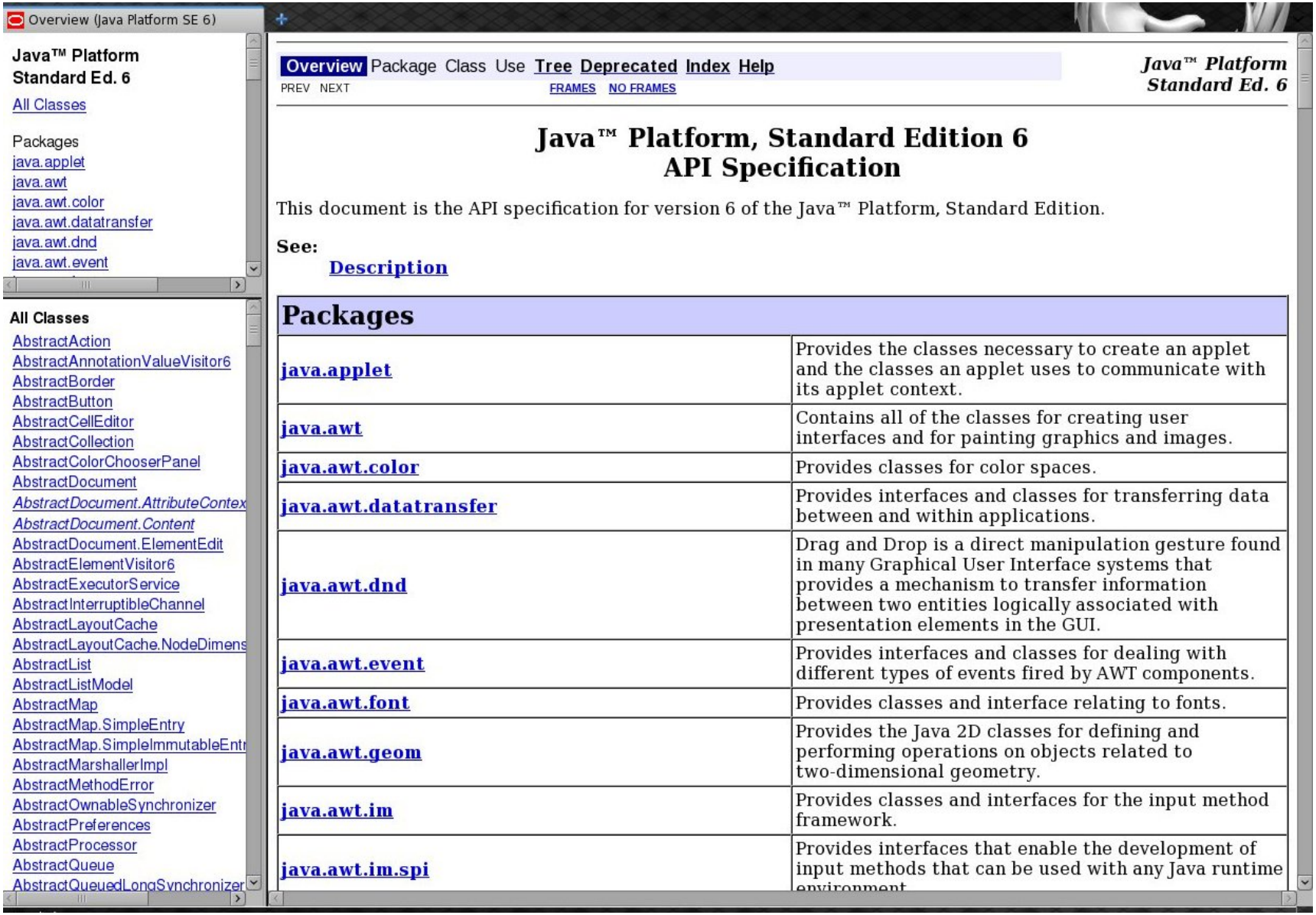
JavaDoc

- Génération avec Eclipse ou NetBeans :



- Ou en ligne de commande à l’aide de la commande `javadoc`

JavaDoc


Overview (Java Platform SE 6)
Java™ Platform
Standard Ed. 6
All Classes
Packages
java.applet
java.awt
java.awt.color
java.awt.datatransfer
java.awt.dnd
java.awt.event
All Classes
AbstractAction
AbstractAnnotationValueVisitor6
AbstractBorder
AbstractButton
AbstractCellEditor
AbstractCollection
AbstractColorChooserPanel
AbstractDocument
AbstractDocument.AttributeContex
AbstractDocument.Content
AbstractDocument.ElementEdit
AbstractElementVisitor6
AbstractExecutorService
AbstractInterruptibleChannel
AbstractLayoutCache
AbstractLayoutCache.NodeDimens
AbstractList
AbstractListModel
AbstractMap
AbstractMap.SimpleEntry
AbstractMap.SimpleImmutableEntr
AbstractMarshallerImpl
AbstractMethodError
AbstractOwnableSynchronizer
AbstractPreferences
AbstractProcessor
AbstractQueue
AbstractQueuedLongSynchronizer
Overview Package Class Use Tree Deprecated Index Help
PREV NEXT
FRAMES NO FRAMES
Java™ Platform
Standard Ed. 6
Java™ Platform, Standard Edition 6
API Specification
This document is the API specification for version 6 of the Java™ Platform, Standard Edition.
See:
Description
Packages
java.applet
Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
java.awt
Contains all of the classes for creating user interfaces and for painting graphics and images.
java.awt.color
Provides classes for color spaces.
java.awt.datatransfer
Provides interfaces and classes for transferring data between and within applications.
java.awt.dnd
Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI.
java.awt.event
Provides interfaces and classes for dealing with different types of events fired by AWT components.
java.awt.font
Provides classes and interface relating to fonts.
java.awt.geom
Provides the Java 2D classes for defining and performing operations on objects related to two-dimensional geometry.
java.awt.im
Provides classes and interfaces for the input method framework.
java.awt.im.spi
Provides interfaces that enable the development of input methods that can be used with any Java runtime

Structure d'un projet

En Java, un projet peut être découpé en paquets.

Les paquets permettent de :

- associer des classes afin de mieux organiser le code
- de créer des "modules indépendants" réutilisables
- d'avoir plusieurs classes qui possèdent le même nom

Un paquet (package) :

- est une collection de classes
- peut contenir des sous-paquets

Lors de l'exécution...

Java utilise l'arborescence de fichier pour retrouver les fichiers `.class` :

- Une classe (ou une interface) correspond à un fichier `.class`
- Un dossier correspond à un paquet

Les `.class` du paquet `com.univ_amu` doivent :

- être dans le sous-dossier `com/univ_amu`
- le dossier `com` doit être à la racine d'un des dossiers du `ClassPath`

Le `ClassPath` inclut :

- le répertoire courant ;
- les dossiers de la variable d'environnement `CLASSPATH` ;
- des archives `JAR`
- des dossiers précisés sur le ligne de commande de `java` ;

Lors de la compilation...

Le mot-clé `package` permet de préciser le paquet des classes ou interfaces définies dans le fichier :

```
package com.univ_amu;

public class MyClass { /* ... */ }
```

Java utilise l'arborescence pour retrouver le code des classes ou interfaces :

- Une classe (ou une interface) `A` est cherchée dans le fichier `A.java`
- Le fichier `A.java` est cherché dans le dossier associé au paquet de `A`

Dans l'exemple précédent, il est donc conseillé que le fichier :

- se nomme `MyClass.java`
- se trouve dans le dossier `com/univ_amu`

(Par défaut, la compilation crée `MyClass.class` dans `com/univ_amu`)

Utilisation d'une classe à partir d'un autre paquet

Accessibilité :

- Seules les classes publiques sont utilisable à partir d'un autre paquet
- Un fichier ne peut contenir qu'une seule classe publique

On peut désigner une classe qui se trouve dans un autre paquet :

```
package com.my_project;

public class Main {
  public static void main(String[] args) {
    com.univ_amu.MyClass myClass =
                    new com.univ_amu.MyClass();
  }
}
```

Importer une classe

Vous pouvez également importer une classe :

```java
package com.my_project;

import com.univ_amu.MyClass;

public class Main {
  public static void main(String[] args) {
    MyClass myClass = new MyClass();
  }
}
```

Deux classes de deux paquets différents peuvent avoir le même nom :

- Exemple : java.util.List et java.awt.List
- Attention de choisir le bon `import`

Importer un paquet

Vous pouvez également importer toutes les classes d'un paquet :

```
package com.my_project;

import com.univ_amu.*;

public class Main {
  public static void main(String[] args) {
    MyClass myClass = new MyClass();
  }
}
```

Remarques :

- Les classes des sous-paquets ne sont pas importées
- Le paquet `java.lang` est importé par défaut

Un exemple

Le fichier com/univ_amu/HelloWorld.java :

```java
package com.univ_amu;

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

```
$ javac com/univ_amu/HelloWorld.java
$ ls com/univ_amu
HelloWorld.java HelloWorld.class
$ java com.univ_amu.HelloWorld
Hello world !
```

Quelques remarques

Nommage des paquets :

- Les noms de paquets sont écrits en minuscules
- Pour éviter les collisions, on utilise le nom du domaine à l'envers
  ⇒ `com.google.gson`, `com.oracle.jdbc`
- Si le nom n'est pas valide, on utilise des underscores :
  ⇒ `com.univ_amu`, `com.example._123name`

Fichier JAR (Java Archive) :

- est une archive ZIP pour distribuer un ensemble de classes Java
- contient un manifest (qui peut préciser la classe qui contient le main)
- peut également faire partie du `ClassPath`
- peut être généré en ligne de commande (`jar`) ou avec un IDE

Accessibilité : modificateur public

Une classe ou un membre est accessible :

- de n'importe où s'il est précédé du modificateur `public` ;
- des classes de son paquet si rien n'est précisé.

```
public class MyClass {
  public int myPublicField;
  int myPackageField;

  public void doPublicAction() { }
  void doPackageAction() { }
}
```

Rappels :

- Seules les classes publiques sont utilisable à partir d'un autre paquet
- Un fichier ne peut contenir qu'une seule classe publique

Accessibilité : modificateur private

Une membre est accessible :

- de n'importe où s'il est précédé du modificateur `public` ;
- des classes de son paquet si rien n'est précisé ;
- **des méthodes de la classe s'il est précédé de private** ;

```
public class MyClass {
 private int privateField;
 private void doPrivateAction() { }
}
```

Afin de limiter les conséquences d'une modification :

- Les méthodes ou propriétés définies afin pour rendre lisible l'implémentation des fonctionnalités doivent être privées ;
- Seule l'interface de la classe doit être publique.

Getters et Setters

```
public class Time {
 private int hour, minute;

 public Time(int hour, int minute) {setTime(hour, minute);}

 public void setTime(int hour, int minute) {
   assert hour >= 0 && hour<=23;
   assert minute >= 0 && minute<=59;
   this.hour = hour; this.minute = minute;
 }

 public int getHour() { return hour; }
 public int getMinute() { return minute; }
}
```

Getters et Setters

Certaines propriétés peuvent être publics si elles sont pertinentes dans l'interface de la classe :

```
public class Point {
 public int x, y;

 public Point(int x, int y) { this.x = x; this.y = y; }

 public void translate(int dx, int dy) {
   this.x += dx; this.y += dy;
 }
}
```

Utilisation

Une méthode peut utiliser les propriétés et méthodes d'une autre instance :

```
public class Point {
 public int x, y;

 public Point(int x, int y) {
   this.x = x; this.y = y;
 }

 public double distanceTo(Point p) {
   int dx = x - p.x;
   int dy = y - p.y;
   return Math.sqrt(dx*dx + dy*dy);
 }
}
```

Composition

Afin d'implémenter ses services, une instance peut créer des instances et conserver leurs références dans des champs :

```
public class Circle {
 private Point center;
 private int radius;

 public Circle(int x, int y, int radius) {
   center = new Point(x,y);
   this.radius = radius;
 }

 public int getX() { return center.x; }
 public int getY() { return center.y; }
 public int getRadius() { return radius; }
}
```

Agrégation

Une instance peut simplement posséder des références vers des instances :

```
public class Circle {
 private Point center, point;

 public Circle(Point center, Point point) {
   this.center = center; this.point = point;
 }

 public Point getCenter() { return center; }

 public int getRadius() {
   int dx = x - p.x, dy = y - p.y;
   return Math.sqrt(dx*dx + dy*dy);
 }
}
```

Agrégation et délégation

Délégation du calcul de la distance à l'instance `center` de la classe `Point` :

```
public class Circle {
 private Point center, point;

 public Circle(Point center, Point point) {
   this.center = center; this.point = point;
 }

 public Point getCenter() { return center; }

 public int getRadius() {
   return center.distanceTo(point);
 }
}
```

Agrégation récursive

Il est possible de créer des structures récursives :

```
public class Node {
  private Node[] nodes;
  private String name;

  public Node(Node[] nodes, String name) {
    this.nodes = nodes; this.name = name;
  }

  public Node(String name) {
    this.nodes = new Node[0]; this.name = name;
  }
}
```

Agrégation récursive

Il est ensuite possible d'implémenter des méthodes de façon récursive :

```
public class Node {
  private Node[] nodes;
  private String name;

  /* Constructeurs du slide précédent. */

  public void print() {
    System.out.print("["+name+"]");
    for (int i = 0; i < nodes.length; i++)
      nodes[i].print();
  }
}
```

Agrégation récursive

Exemple d'utilisation de la classe précédente :

```
Node a = new Node("a");
Node b = new Node("b");
Node c = new Node("c");
Node d = new Node("d");
Node ab = new Node(new Node[] {a,b}, "ab");
Node cd = new Node(new Node[] {c,d}, "cd");
Node abcd = new Node(new Node[] {ab, cd}, "abcd");
abcd.print();
```

Cet exemple produit la sortie suivante :

```
[abcd] [ab] [a] [b] [cd] [c] [d]
```

Classe interne statique

Il est possible de définir une classe à l'intérieur d'une autre :

```
public class LinkedList {
  public static class Node {
    private String data;
    private Node next;

    public Node(String data, Node next) {
      this.data = data; this.next = next;
    }
  }
}
```

Il est possible d'instancier la classe interne sans qu'une instance de `LinkedList` existe car elle est statique :

```
LinkedList.Node node = new LinkedList.Node("a", null);
```

Classe interne statique

Il est également possible de la rendre privée à la classe `LinkedList` :

```
public class LinkedList {
  private static class Node {
    private String data;
    private /*LinkedList.*/Node next;

    public Node(String data, Node next) {
      this.data = data; this.next = next;
    }
  }
}
```

Dans ce cas, seules les méthodes de `LinkedList` pourront l'utiliser. Notez que des méthodes statiques définies dans `LinkedList` peuvent également utiliser cette classe interne du fait qu'elle soit statique.

Classe interne statique

Exemple d'implémentation de méthodes dans la classe `LinkedList` :

```
public class LinkedList {
  /* Code de la classe interne statique Node. */

  private Node first = null;

  public void add(String data) {
    first = new Node(data, first);
  }

  public void print() {
    Node node = first;
    while (node!=null) {
      System.out.print("["+node.data+"]");
      node = node.next;
    }
  }
}
```

Classe interne statique

Exemple d'utilisation de la classe précédente :

```
LinkedList list = new LinkedList();
list.add("a");
list.add("b");
list.add("c");
list.print();
```

Cet exemple produit la sortie suivante :

```
[c] [b] [a]
```

Classe interne statique

Une classe interne statique ne peut accéder qu'aux champs et méthodes statiques de la classe qui la contient :

```
public class LinkedList {

  private static class Node {
    /* Champs et méthodes de Node. */
    boolean isFirst() {
      return this==first;  // interdit !
    }
  }

  private Node first;
  /* Autres champs et méthodes de LinkedList. */
}
```

Classe interne

En revanche, si la classe interne n'est pas statique, elle peut accéder aux champs de classe qui la contient :

```
public class LinkedList {

  private class Node {
    /* Champs et méthodes de Node. */
    private boolean isFirst() {
      return this==first;  // autorisé !
    }
  }

  private Node first;
  /* Autres champs et méthodes de LinkedList. */
}
```

Classe interne

Java insère dans l'instance de `Node` une référence vers l'instance de `LinkedList` qui a permis de la créer :

```
public class LinkedList {
  private class Node {
    /* Champs et méthodes de Node. */
    private boolean isFirst() {
      return this==/*référenceVersLinkedList.*/first;
    }
  }

  public void add(String data) {
    first = new Node(data, first);
  }

  /* Autres champs et méthodes de la classe LinkedList. */
}
```

Classe interne

Il est possible d'utiliser la méthode `isFirst` dans `LinkedList` :

```
public class LinkedList {
  /* Code de la classe interne statique Node
     et champs et méthodes de la classe LinkedList. */

  public void print() {
    Node node = first;
    while (node!=null) {
      System.out.print("["+node.data
                         +","+node.isFirst()+"]");
      node = node.next;
    }
  }
}
```

Classe interne

Exemple d'utilisation de la classe précédente :

```
LinkedList list = new LinkedList();
list.add("a");
list.add("b");
list.add("c");
list.print();
```

Cet exemple produit la sortie suivante :

```
[c,true][b,false][a,false]
```

Exemple : itérateurs

Nous ajoutons une classe interne afin de pouvoir parcourir la liste :

```
public class LinkedList {
  /* Classe interne Node, autres champs et méthodes. */
  public Iterator iterator() { return new Iterator(); }

  public class Iterator {
    private Node node;
    public Iterator() { node = first; }
    public boolean hasNext() { return node!=null; }
    public String next() {
      String data = node.data; node = node.next;
      return data;
    }
  }
}
```

Exemple : itérateurs

Exemple d'utilisation de l'itérateur :

```
LinkedList list = new LinkedList();
list.add("a");
list.add("b");
list.add("c");
LinkedList.Iterator iterator = list.iterator();
while (iterator.hasNext())
  System.out.print("["+iterator.next()+"]");
```

Cet exemple produit la sortie suivante :

```
[c][b][a]
```

Notez que la classe interne `Node` n'est accessible que par les méthodes de `LinkedList` et de ses classes internes car elle est privée.

Rappel : Les objets et les classes

Un **objet** :

- peut être **construit**
- est **structuré** : il est constitué d'un ensemble de propriétés
- possède un **état** : la valeur de ses propriétés
- possède une **interface** : l'ensemble des méthodes et propriétés accessibles par les utilisateurs de l'objet

Dans les langages orientés objet, une **classe** (d'objets) définit :

- une façon de construire des objets (**constructeur**)
- la structure des objets de la classe (**propriétés**)
- le comportement des objets de la classe (**méthodes**)
- l'interface des objets de la classe (**méth. et prop. non-privées**)
- un type "référence vers des objets de cette classe"

Abstraction

Supposons que des classes implémentent un service de façons différentes :

```
class SimplePrinter {
  void print(String document) {
    System.out.println(document);
  }
}
```

```
class BracePrinter {
  void print(String document) {
    System.out.println("{"+document+"}");
  }
}
```

Toutes les instances de ces classes possèdent la méthode `print`.

Abstraction

Nous souhaiterions pouvoir facilement passer du code suivant :

```
Printer printer = new SimplePrinter();
printer.print("truc");    // → "truc";
```

au code suivant :

```
Printer printer = new BracePrinter();
printer.print("truc");    // → "{truc}";
```

Il nous faudrait définir un type `Printer` qui oblige la variable à contenir des références vers des objets qui implémentent la méthode `print`.

Abstraction

On peut vouloir traiter des objets en utilisant les services qu'ils partagent :

```
for (int index = 0; index < printers.length; index++)
 printers[index].print(document);
```

On peut aussi vouloir écrire un programme en supposant que les objets manipulés implémentent certains services :

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

Description d'une interface

Description d'une interface en Java :

```
public interface Printer {
    /**
     * Affiche la chaine de caracteres document.
     * @param document la chaine a afficher.
     */
    public void print(String document);
}
```

Une interface :

- peut contenir une ou plusieurs méthodes
- est un contrat

Implémentation d'une interface

`implements` permet d'indiquer qu'une classe implémente une interface :

```
class SimplePrinter implements Printer {
  void print(String document) {
    System.out.println(document);
  }
}
```

```
class BracePrinter implements Printer {
  void print(String document) {
    System.out.println("{"+document+"}");
  }
}
```

Java vérifie que toutes les méthodes de l'interface sont implémentées.

Références et interfaces

Déclaration d'une variable de type référence vers une instance d'une classe qui implémente l'interface `Printer` :

```
Printer printer;
```

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

```
Printer printer = new Printer(); // interdit !
printer.print("toto");           // que faire ici ?
```

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

```
SimplePrinter simplePrinter = new SimplePrinter();
```

et de mettre la référence dans une variable de type `Printer` :

```
Printer printer1 = simplePrinter;          // upcasting
Printer printer2 = new BracePrinter();
Printer string = new String("c")           // interdit !
```

Références et interfaces

Utilisation d'objets qui implémentent l'interface :

```
class Utils {
  static void printString(Printer[] printers, String doc) {
    for (unsigned int i = 0; i < printers.length; i++)
      printers[i].print(doc);
  }

  static void printArray(String[] array, Printer printer) {
    for (unsigned int i = 0; i < array.length; i++)
      printer.print(array[i]);
  }
}
```

Polymorphisme

Vérification à la compilation de l'existance des méthodes :

```
Printer printer = new SimplePrinter();
printer.println("msg"); // impossible !
```

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

```
Printer[] printers = new Printer[2];
printers[0] = new SimplePrinter();
printers[1] = new BracePrinter();
Random random = new Random(); // générateur aléatoire
int index = random.nextInt(2); // 0 et 1
printers[index].print("mon message");
```

`index=0` → méthode de la classe `SimplePrinter` → `mon message`
`index=1` → méthode de la classe `BracePrinter` → `{mon message}`

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 Printable {
  public void print();
}
```

La méthode `print` permet d'afficher l'objet.

```
interface Stack {
  public void push(int value);
  public int pop();
}
```

La méthode `push` permet d'empiler un entier.
La méthode `pop` dépile et retourne l'entier en haut de la pile.

Implémentations multiples

Implémentation des deux interfaces précédentes :

```
public class PrintableStack implements Stack, Printable {
  private int[] array; private int size;

  public PrintableStack(int capacity) {
    array = new int[capacity]; size = 0;
  }

  public void push(int v) { array[size] = v; size++; }
  public int pop() { size--; return array[size]; }

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

Implémentations multiples

Implémentation d'une des deux interfaces :

```
public class PrintableString implements Printable {

  private String string;

  public PrintableString(String string) {
    this.string = string;
  }

  public void print() {
    System.out.println(string);
  }

}
```

Implémentations multiples

Exemple d'utilisation des classes précédentes :

```
Printable[] printables = new Printable[3];
printables[0] = new PrintableString("bonjour");
PrintableStack stack = new PrintableStack(10);
printables[1] = stack;
printables[2] = new PrintableString("salut");
stack.push(10);
stack.push(30);
System.out.println(stack.pop());
stack.push(12);
for (int i = 0; i < printables.length; i++)
  printables[i].print();
```

Qu'écrit ce programme sur la sortie standard ?

Vérification des types

Vérification des types à la compilation :

```
Stack[] arrayStack = new Stack[2];
arrayStack[0] = new PrintableStack();
arrayStack[1] = new PrintableString("t"); // Erreur !
```

`PrintableString` n'implémente pas `Stack`.

```
Stack stack = new PrintableStack();
Printable printable = stack; // Erreur !
```

Le type `Stack` n'est pas compatible avec le type `Printable`.

Exemple : Arbre binaire

```java
public interface Node {
  public String convertToString();
}
```

```java
public class Leaf implements Node {
  private String name;

  public Leaf(String name) { this.name = name; }

  @Override
  public String convertToString() { return name; }
}
```

Exemple : Arbre binaire

```
public class InternalNode implements Node {
  private Node left, right; // Agrégation

  public InternalNode(Node left, Node right) {
    this.left = left;
    this.right = right;
  }

  @Override
  public String convertToString() {
    return "(" + left.convertToString() + "," +
           right.convertToString() + ")";
  }
}
```

Exemple : Arbre binaire

```
public class Exemple {
  public static void main(String[] args) {
    Node a = new Leaf("a");
    Node b = new Leaf("b");
    Node c = new Leaf("c");
    Node d = new Leaf("d");
    Node ab = new InternalNode(a, b);
    Node cd = new InternalNode(c, d);
    Node abcd = new InternalNode(ab, cd);
    System.out.println(abcd.convertToString());
  }
}
```

Qu'affiche ce programme ?

Exemple : Arbre binaire et Printer

```java
public interface Node {
  public void print(Printer printer);
}
```

```java
public class Leaf implements Node {
  private String name;

  public Leaf(String name) { this.name = name; }

  @Override
  public void print(Printer printer) {
    printer.print(name);
  }
}
```

Exemple : Arbre binaire et Printer

```
public class InternalNode implements Node {
  private Node left, right;

  public InternalNode(Node left, Node right) {
    this.left = left;
    this.right = right;
  }

  @Override
  public void print(Printer printer) {
    left.print(printer);
    right.print(printer);
  }
}
```

Exemple : Arbre binaire et Printer

```
public class Exemple {
  public static void main(String[] args) {
    Node a = new Leaf("a");
    Node b = new Leaf("b");
    Node ab = new InternalNode(a, b);
    ab.print(new SimplePrinter());
    ab.print(new BracePrinter());
  }
}
```

Qu'affiche ce programme ?

Classes anonymes

Il est possible d'implémenter directement une interface dans le code :

```
public class Exemple {
  public static void main(String[] args) {
    Node a = new Leaf("a");
    Printer printer = new Printer() {
       // implémentation des méthodes de l'interface
       public void print(String document) {
         System.out.println("("+document+")");
       }
    };
    a.print(printer);
  }
}
```

Une telle classe est dite *anonyme* car on ne lui associe pas de nom.

Classes anonymes

Bien évidemment, la variable intermédiaire n'est pas nécessaire :

```
public class Exemple {
  public static void main(String[] args) {
    Node a = new Leaf("a");
    a.print(new Printer() {
        // implémentation des méthodes de l'interface
        public void print(String document) {
          System.out.println("("+document+")");
        }
    });
  }
}
```

Transtypage

Il est possible de forcer une conversion de type :

```
Printer printer = new SimplePrinter();
// ↪ l'upcasting est toujours correct
//      donc nous n'avons pas besoin d'opérateur.
SimplePrinter simplePrinter = (SimplePrinter)printer;
// ↪ utilisation de l'opérateur de transtypage
//      car nous ne faisons pas un upcasting.
```

Attention, un transtypage peut échouer (à l'exécution) :

```
Printer printer = new BracePrinter();
SimplePrinter simplePrinter = (SimplePrinter)printer; // !!
```

```
String string = "toto";
Printer printer = (Printer)string; // !!
```

Problématique

Supposons que nous ayons la classe suivante :

```
public class ListSum {
  private int[] list = new int[10];
  private int size = 0;

  public void add(int value) {list[size] = value; size++;}

  public int eval() {
    int result = 0;
    for (int i = 0; i < size; i++)
      result += list[i];
    return result;
  }
}
```

Problématique

Supposons que nous ayons la classe suivante :

```
public class ListProduct {
  private int[] list = new int[10];
  private int size = 0;

  public void add(int value) {list[size] = value; size++;}

  public int eval() {
    int result = 1;
    for (int i = 0; i < size; i++)
      result *= list[i];
    return result;
  }
}
```

Problématique

Il est possible de “refactorer” les classes précédentes de façon à isoler les différences dans des méthodes :

```
public class ListSum {
  /* Propriétés, constructeur et méthode add. */

  public int eval() {
    int result = neutral();
    for (int i = 0; i < size; i++)
      result = compute(result, list[i]);
    return result;
  }

  private int neutral() { return 0; }
  private int compute(int a, int b) { return a+b; }
}
```

Problématique

Il est possible de “refactorer” les classes précédentes de façon à isoler les différences dans des méthodes :

```
public class ListProduct {
  /* Propriétés, constructeur et méthode add. */

  public int eval() {
    int result = neutral();
    for (int i = 0; i < size; i++)
      result = compute(result, list[i]);
    return result;
  }

  private int neutral() { return 1; }
  private int compute(int a, int b) { return a*b; }
}
```

Problématique

Après la refactorisation du code :

- seules les méthodes `neutral` et `compute` diffèrent ;
- il serait intéressant de pouvoir supprimer les duplications de code ;

Deux solutions :

- La délégation en utilisant une interface ;
- L'extension et les classes abstraites.

Délégation

Nous allons externaliser les méthodes `neutral` et `compute` dans deux nouvelles classes. Elles vont implémenter l'interface `Operator` :

```
public interface Operator {
  public int neutral();
  public int compute(int a, int b);
}
```

```
public class Sum implements Operator {
  public int neutral() { return 0; }
  public int compute(int a, int b) { return a+b; }
}
```

```
public class Product implements Operator {
  public int neutral() { return 1; }
  public int compute(int a, int b) { return a*b; }
}
```

Délégation

Les classes `ListSum` et `ListProduct` sont fusionnées dans une unique classe `List` qui délègue les calculs à un objet qui implémente `Operator` :

```
public class List {
 /* Propriétés et méthode add */
 private Operator operator;

 public List(Operator operator){this.operator = operator;}

 public int eval() {
  int result = operator.neutral();
  for (int i = 0; i < size; i++)
   result = operator.compute(result, list[i]);
  return result;
 }
}
```

Délégation

Utilisation des classes ListSum et ListProduct :

```
ListSum listSum = new ListSum();
listSum.add(2); listSum.add(3);
System.out.println(listSum.eval());
ListProduct listProduct = new ListProduct();
listProduct.add(2); listProduct.add(3);
System.out.println(listProduct.eval());
```

Utilisation après la refactorisation du code :

```
List listSum = new List(new Sum());
listSum.add(2); listSum.add(3);
System.out.println(listSum.eval());
List listProduct = new List(new Product());
listProduct.add(2); listProduct.add(3);
System.out.println(listProduct.eval());
```

Classes abstraites

Une classe est abstraite si des méthodes ne sont pas implémentées :

```
public abstract class List {
  private int[] list = new int[10];
  private int size = 0;

  public void add(int value) { list[size] = value; size++; }

  public int eval() {
    int result = neutral(); // util. d'une méthode abstraite
    for (int i = 0; i < size; i++)
      result = compute(result, list[i]); // idem
    return result;
  }

  public abstract int neutral(); // méthode abstraite
  public abstract int compute(int a, int b); // idem
}
```

Classes abstraites et extension

Évidemment, il n'est pas possible d'instancier une classe abstraite :

```
List list = new List(); // erreur
System.out.println(list.eval()) // car que faire ici ?
```

Nous allons étendre la classe `List` afin de récupérer les propriétés et les méthodes de `List` et définir le code des méthodes abstraites :

```
public class ListSum extends List {
  public int neutral() { return 0; }
  public int compute(int a, int b) { return a+b; }
}
```

La classe `ListSum` n'est plus abstraite, toutes ses méthodes sont définies :

```
ListSum listSum = new ListSum();
listSum.add(3); listSum.add(7);
System.out.println(listSum.eval());
```

Classes abstraites et extension

Évidemment, il n'est pas possible d'instancier une classe abstraite :

```
List list = new List(); // erreur
System.out.println(list.eval()) // car que faire ici ?
```

Nous allons étendre la classe `List` afin de récupérer les propriétés et les méthodes de `List` et définir le code des méthodes abstraites :

```
public class ListProduct extends List {
  public int neutral() { return 1; }
  public int compute(int a, int b) { return a*b; }
}
```

La classe `ListSum` n'est plus abstraite, toutes ses méthodes sont définies :

```
ListProduct listProduct = new ListProduct();
listProduct.add(3); listProduct.add(7);
System.out.println(listProduct.eval());
```

Généralisation aux classes non-abstraites

Plus généralement, l'extension permet de créer une classe en :

- conservant les services (propriétés et méthodes) d'une autre classe ;
- ajoutant de nouveaux services (propriétés et méthodes) ;
- redéfinissant certains services (méthodes).

En Java :

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

Il est toujours préférable de privilégier l'implémentation à l'extension.

Ajout de nouveaux services

Supposons que nous ayons la classe `Point` suivante :

```
public class Point {
  public int x, y;

  public void setPosition(int x, int y) { this.x = x; this.y = y; }
}
```

Il est possible d'ajouter de nouveaux services en utilisant l'extension :

```
public class Pixel extends Point {
  public int r, g, b;

  public void setColor(int r, int g, int b)
    { this.r = r; this.g = g; this.b = b; }
}
```

Ajout de nouveaux services

Les services de `Point` sont disponibles dans `Pixel` :

```
Pixel pixel = new Pixel();
pixel.setPosition(4,8);
System.out.println(pixel.x); // → 4
System.out.println(pixel.y); // → 8
pixel.setColor(200,200,120);
```

Évidemment, les services de `Pixel` ne sont disponibles dans `Point` :

```
Point point = new Point();
point.setPosition(4,8);
System.out.println(point.x); // → 4
System.out.println(point.y); // → 8
point.setColor(200,200,120); // impossible !
```

Redéfinition de méthode

Supposons que nous ayons la classe `Point` suivante :

```
public class Point {
  public int x, y;

  public void setPosition(int x, int y) { this.x = x; this.y = y; }
  public void clear() { x = 0; y = 0; }
}
```

Il est possible de redéfinir la méthode `clear` dans `Point` :

```
public class Pixel extends Point {
  public int r, g, b;

  public void setColor(int r, int g, int b)
    { this.r = r; this.g = g; this.b = b; }

  public void clear() { x = 0; y = 0; r = 0; g = 0; b = 0; }
}
```

Le mot-clé super

Supposons que nous ayons la classe `Point` suivante :

```
public class Point {
  public int x, y;

  public void setPosition(int x, int y) { this.x = x; this.y = y; }
  public void clear() { setPosition(0, 0); }
}
```

Le mot-clé super permet d'utiliser la méthode `clear` de `Point` :

```
public class Pixel extends Point {
  public int r, g, b;

  public void setColor(int r, int g, int b)
    { this.r = r; this.g = g; this.b = b; }

  public void clear() { super.clear(); setColor(0, 0, 0); }
}
```

Le mot-clé super

Supposons que nous ayons la classe `Point` suivante :

```
public class Point {
  public int x, y;

  public void setPosition(int x, int y) { this.x = x; this.y = y; }
}
```

Si la méthode n'a pas été redéfinie, le mot clé `super` est inutile :

```
public class Pixel extends Point {
  public int r, g, b;

  public void setColor(int r, int g, int b)
    { this.r = r; this.g = g; this.b = b; }

  public void clear() { /*super.*/setPosition(0, 0); setColor(0, 0, 0); }
}
```

Les constructeurs et le mot-clé super

Supposons que nous ayons la classe `Point` suivante :

```
public class Point {
  public int x, y;

  public Point(int x, int y) { this.x = x; this.y = y; }
}
```

La classe `Point` n'a pas de constructeur sans paramètre, il faut donc indiquer comment initialiser la partie de `Pixel` issue de la classe `Point` :

```
public class Pixel extends Point {
  public int r, g, b;

  public Pixel(int x, int y, int r, int g, int b) {
    super(x, y); // appel du constructeur de Point
    this.r = r; this.g = g; this.b = b;
  }
}
```

Les constructeurs

Supposons que nous ayons la classe `Point` suivante :

```
public class Point {
  public int x, y;

  public Point() { x = 0; y = 0; }
  public Point(int x, int y) { this.x = x; this.y = y; }
}
```

Par défaut, le constructeur sans paramètre est appelé :

```
public class Pixel extends Point {
  public int r, g, b;

  public Pixel(int r, int g, int b) {
    // appel du constructeur sans paramètre de Point
    this.r = r; this.g = g; this.b = b;
  }
}
```

Les constructeurs

Supposons que nous ayons la classe `Point` suivante :

```
public class Point {
  public int x, y;

  //public Point() { x = 0; y = 0; }
  public Point(int x, int y) { this.x = x; this.y = y; }
}
```

Ici, vous devez préciser les paramètres du constructeur avec `super` :

```
public class Pixel extends Point {
  public int r, g, b;

  public Pixel(int r, int g, int b) {
    // erreur de compilation (aucun constructeur sans paramètre)
    this.r = r; this.g = g; this.b = b;
  }
}
```

Transtypages et polymorphisme

Aucune méthode ou propriété ne peut être supprimée lors d'une extension. Par exemple, `Pixel` possède toutes les propriétés et méthodes de `Point` (même si certaines méthodes ont pu être redéfinies).

Par conséquent, l'upcasting est toujours autorisé :

```
Point point = new Pixel();
point.setPosition(2,4);
System.out.println(point.x + " " + point.y);
point.clear();
```

Remarques :

- Le code a exécuté lors d'un appel de méthode est déterminé à l'exécution en fonction de la référence présente dans la variable.
- Le typage des variables permet de vérifier à la compilation l'existence des propriétés et des méthodes.

La classe Object

Par défaut, les classes étendent la classe `Object` de Java. Par conséquent, l'upcasting vers la classe `Object` est toujours possible :

```
Pixel pixel = new Pixel();
Object object = pixel;
Object[] array = new Object[10];
for (int i = 0; i < t; i++) {
  if (i%2==0) array[i] = new Point();
  else array[i] = new Pixel();
}
```

Notez que `object.setPosition(2,3)` n'est pas autorisé dans le code précédent car la classe `Object` ne possède pas la méthode `setPosition` et seul le type de la variable compte pour déterminer si l'appel d'une méthode ou l'utilisation d'une propriété est autorisé.

La méthode toString() de la classe Object

Par transitivité de l'extension, toutes les méthodes et propriétés de la classe `Object` sont disponibles sur toutes les instances :

```
Object object = new Object(); Point point = new Point(2,3);
System.out.println(object.toString());
                              // ↪ java.lang.Object@19189e1
System.out.println(point.toString());
                              // ↪ test.Point@7c6768
```

La méthode `toString` est utilisée par Java pour convertir une référence en chaîne de caractères :

```
Object object = new Object(); Point point = new Point(2,3);
String string = object+";"+point;
System.out.println(string);
// ↪ java.lang.Object@19189e1;test.Point@7c6768
```

La méthode toString() de la classe Object

Évidemment, il est possible de redéfinir la méthode `toString` :

```
public class Point {
  public int x, y;

  public Point(int x, int y) { this.x = x; this.y = y; }

  public String toString() { return "("+x+","+y+")"; }
}
```

Le polymorphisme fait que cette méthode est appelée si la variable contient une référence vers un `Point` :

```
Point point = new Point(2,3); Object object = point;
System.out.println(point);  // → (2,3)
System.out.println(object); // → (2,3)
```

Extension d'interface

Supposons que nous ayons l'interface suivante :

```
public interface List {
  public int size();
  public void get(int index);
}
```

En Java, il est également possible d'étendre une interface :

```
public interface ModifiableList extends List {
  public void add(int value);
  public void remove(int index);
}
```

Une classe qui implémente l'interface `ModifiableList` doit implémenter les méthodes `size`, `get`, `add` et `remove`.

Extension d'interface

Supposons que la classe `ArrayModifiableList` implémente l'interface `ModifiableList`. Dans ce cas, nous pouvons écrire :

```
ModifiableList modifiableList = new ArrayModifiableList();
modifiableList.add(2);
modifiableList.add(5);
modifiableList.remove(0);
List list = modifiableList;
System.out.println(list.size());
```

En revanche, il n'est pas possible d'écrire :

```
list.remove(0);
  /* ↪ Cette méthode n'existe pas */
  /*    dans l'interface List. */
```

Extension d'interface

Supposons que nous avons l'interface suivante :

```
public interface Printable {
  public void print();
}
```

En Java, une interface peut étendre plusieurs interfaces :

```
public interface ModifiablePrintableList
                        extends ModifiableList, Printable {
}
```

Notons que nous ne définissons pas de nouvelles méthodes dans l'interface `ModifiablePrintableList`. Cette interface ne représente que l'union des interfaces `ModifiableList` et `Printable`. Bien évidemment, de nouvelles méthodes auraient pu être définies dans `ModifiablePrintableList`.

Accessibilité : modificateur public

Une classe ou un membre est accessible :

- de n'importe où s'il est précédé du modificateur `public` ;
- des classes de son paquet si rien n'est précisé.

```
public class MyClass {
  public int myPublicField;
  int myPackageField;

  public void doPublicAction() { }
  void doPackageAction() { }
}
```

Rappels :

- Seules les classes publiques sont utilisable à partir d'un autre paquet
- Un fichier ne peut contenir qu'une seule classe publique

Accessibilité : modificateur private

Une membre est accessible :

- de n'importe où s'il est précédé du modificateur `public` ;
- des classes de son paquet si rien n'est précisé ;
- des méthodes de la classe s'il est précédé de private ;

```
public class MyClass {
 private int privateField;
 private void doPrivateAction() { }
}
```

Afin de limiter les conséquences d'une modification :

- Les méthodes ou propriétés définies afin pour rendre lisible l'implémentation des fonctionnalités doivent être privées ;
- Seule l'interface de la classe doit être publique.

Accessibilité : modification protected

Un membre `protected` n'est accessible que par les méthodes des classes du paquet et par les classes qui l'étendent.

Le modificateur `protected` permet de protéger un membre :

```
public class MyClass {
 protected int protectedField;
 protected void doProtectedAction() { }
}
```

	Classe	Paquet	Extension	Extérieur
Private	✓			
Default	✓	✓		
Protected	✓	✓	✓	
Public	✓	✓	✓	✓

Surcharge de méthode

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 Adder {
  public static int add(int intVal1, int intVal2) {
    System.out.println("integer"); return intVal1+intVal2;
  }
  public static double add(double doubleVal1, double doubleVal2) {
    System.out.println("double"); return doubleVal1+doubleVal2;
  }
}

int intValue = 1; double doubleValue = 2.2;
double result1 = Adder.add(doubleValue, doubleValue); // → double
double result2 = Adder.add(intValue, doubleValue);    // → double
int result3 = Adder.add(intValue, intValue);          // → integer
```

Surcharge de méthode

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 Printer {
  static void print(Object object) {
    System.out.println("Object : "+object);
  }
  static void print(String string) {
    System.out.println("String : "+string);
  }
}

String string = "message";
Object object = string;
Printer.print(string); // → String : message
Printer.print(object); // → Object : message
```

Redéfinition de méthode – Pas de contravariance

```
class Card {}
class PrettyCard extends Card {}

class Deck {
  public void add(PrettyCard card) { /* 1 */ }
}

class BeloteDeck extends Deck {
  public void add(Card card) { /* 2 */ }
}

BeloteDeck deck = new BeloteDeck();
deck.method(new PrettyCard()); // -> méthode 1
deck.method(new Card()); // -> méthode 2
```

Redéfinition de méthode – Covariance

```
class Card {}
class PrettyCard extends Card {}

class Deck {
  public Card top() { /* 1 */ }
}

class BeloteDeck extends Deck {
  public PrettyCard top() { /* 2 */ }
}

BeloteDeck beloteDeck = new BeloteDeck();
PrettyCard prettyCard = beloteDeck.top(); // -> 2
Deck deck = beloteDeck;
Card card = deck.top(); // -> 2
```

Les types paramétrés

Supposons que nous ayons la classe suivante :

```
public class Stack {
  private Object[] stack = new Object[100];
  private int size = 0;

  public void push(Object object) {
    stack[size] = object; size++;
  }

  public Object pop() {
    size--; Object object = stack[size];
    stack[size]=null; // Pour le Garbage Collector.
    return object;
  }
}
```

Types paramétrés

Nous rencontrons le problème suivant :

```
Stack stack = new Stack();
String string = "truc";
stack.push(string);
string = (String)stack.pop(); // Transtypage obligatoire !
```

Nous avons également le problème suivant :

```
Stack stack = new Stack();
Integer intValue = new Integer(2);
stack.push(intValue);
String string = (String)stack.pop();// Erreur à l'exécution
```

Types paramétrés

Par conséquent, on souhaiterait pouvoir préciser le type des éléments :

```
Stack<String> stack = new Stack<String>();
String string = "truc";
stack.push(string);  // Le paramètre doit être un String.
String string = stack.pop(); // retourne un String.
```

Java nous permet de définir une classe `Stack` qui prend en paramètre un type. Ce type paramétré va pouvoir être utilisé dans les signatures des méthodes et lors de la définition des champs de la classe.

Lors de la compilation, Java va utiliser le type paramétré pour effectuer :

- des vérifications de type ;
- des transtypages automatiques ;
- des opérations d'emballage ou de déballage de valeurs.

Classes paramétrées

La nouvelle version de la classe Stack :

```
public class Stack<T> {
  private Object[] stack = new Object[100];
  private int size = 0;

  public void push(T element) {
    stack[size] = element; size++;
  }

  public T pop() {
    size--;
    T element = (T)stack[size];
    stack[size] = null;
    return element;
  }
}
```

Emballage et déballage

Les types simples ne sont pas des classes :

```
Stack<int> stack = new Stack<int>(); // interdit !
```

Dans ce cas, on doit utiliser la classe d'emballage `Integer` :

```
Stack<Integer> stack = new Stack<Integer>();
int intValue = 2;
Integer integer = new Integer(intValue);
        // ↪ emballage du int dans un Integer.
stack.push(integer);
Integer otherInteger = stack.pop();
int ohterIntValue = otherInteger.intValue();
   // ↪ déballage du int présent dans le Integer.
```

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)`

Emballage et déballage automatique

Depuis Java 5, il existe l'emballage et le déballage automatique :

```
Stack<Integer> stack = new Stack<Integer>();
int intValue = 2;
stack.push(intValue);
  // ↪ emballage automatique du int dans un Integer.
int otherIntValue = stack.pop();
  // ↪ déballage automatique du int.
```

Attention : il est important de noter que des allocations sont effectuées lors des emballages sans que des `new` soient présents dans le code.

Exemple : liste chaînée générique

Dans le troisième cours, nous avions défini la classe suivante :

```
public class LinkedList {
  private class Node {
    private String data;
    private Node next;
    public Node(String data, Node next) {
      this.data = data; this.next = next;
    }
  }

  private Node first = null;

  public void add(String data) {
   first = new Node(data, first);
  }
}
```

Exemple : liste chaînée générique

Nous la transformons en classe paramétrée de la façon suivante :

```
public class LinkedList<T> {
  private class Node {
    private T data;
    private Node next;

    public Node(T data, Node next) {
      this.data = data; this.next = next;
    }
  }

  private Node first = null;

  public void add(T data) {
    first = new Node(data, first);
  }
}
```

Exemple : itérateurs génériques

Il est possible de définir un itérateur générique :

```
public class LinkedList<T> {
  /* Classe interne Node, autres champs et méthodes. */
  public Iterator iterator() { return new Iterator(); }

  public class Iterator {
    private Node node;
    public Iterator() { node = first; }
    public boolean hasNext() { return node!=null; }
    public T next() {
      String data = node.data; node = node.next;
      return data;
    }
  }
}
```

Intégration des itérateurs par Java

Les deux interfaces paramétrées suivantes sont définies en Java :

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

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

Intégration des itérateurs par Java

Il est possible de définir un itérateur générique :

```
public class LinkedList<T> implements Iterable<T> {
  /* Classe interne Node, autres champs et méthodes. */

  public Iterator iterator() { return new Iterator(); }
            // ↪ covariance

  private class Iterator implements java.util.Iterator<T> {
    /* Voir le slide suivant. */
  }
}
```

Intégration des itérateurs par Java

Implémentation de la classe `Iterator` :

```
public class LinkedList<T> implements Iterable<T> {
  /* Classe interne Node, autres champs et méthodes. */

  private class Iterator implements java.util.Iterator<T> {
    private Node prev, current, next;
    public Iterator() { next = first; }
    public boolean hasNext() { return next!=null; }
    public T next() {
      prev = current; current = next;
      T data = next.data; next = next.next; return data;
    }
    public void remove() { /* Slide suivant. */ }
  }
}
```

Intégration des itérateurs par Java

Implémentation de la méthode `remove` :

```
public class LinkedList<T> implements Iterable<T> {
  /* Classe interne Node, autres champs et méthodes. */
  private Node first;

  private class Iterator implements java.util.Iterator<T> {
    private Node prev, current, next;
    /* et les méthodes du slide précédent. */

    public void remove() {
      if (current == null) return;
      if (prev==null) first = next; else prev.next = next;
    }
  }
}
```

Intégration des itérateurs par Java

Exemple d'utilisation de l'itérateur :

```
LinkedList<String> list = new LinkedList<String>();
list.add("a");
list.add("b");
list.add("c");
Iterator<String> iterator = list.iterator();
while (iterator.hasNext())
  System.out.print("["+iterator.next()+"]");
```

Cet exemple produit la sortie suivante :

```
[c][b][a]
```

Intégration des itérateurs par Java

En Java 5, il est possible d'écrire directement :

```
LinkedList<String> list = new LinkedList<String>();
list.add("a");
list.add("b");
list.add("c");
for (String string : list)
  System.out.print("["+string+"]");
```

Cet exemple produit la sortie suivante :

```
[c][b][a]
```

Pour utiliser ces boucles, il faut que :

- l'objet à droite du `for` implémente l'interface `Iterable<T>` ;
- les références vers des objets de type `T` soit affectables à la variable définie à gauche dans le `for`.

Condition sur les paramètres – Problématique

```
public interface Comparable<T> {
  public int compareTo(T element);
}
```

```
class Greatest {
  private String element;

  public void add(String element) {
    if (this.element==null || element.compareTo(this.element)>0)
      this.element = element;
  }

  public String get() { return element; }
}
```

Comment rendre la classe `Greatest` générique ?

Condition sur les paramètres

```
class Greatest<T extends Comparable<T>> {
  private T element;

  public void add(T element) {
    if (this.element==null || element.compareTo(this.element)>0)
      this.element = element;
  }

  public T get() { return element; }
}
```

? super – Problématique

Supposons que nous ayons les classes suivantes :

```
class Greatest<T extends Comparable<T>> {
  /* ... */
  public void add(T element) { /* ... */ }
  public T get() { return element; }
}

class Card implements Comparable<Card> { /* ... */ }
class PrettyCard extends Card { /* ... */ }
```

Il n'est pas possible d'écrire les lignes suivantes car `PrettyCard` n'implémente pas l'interface `Comparable<PrettyCard>` :

```
Greatest<PrettyCard> greatest = new Greatest<PrettyCard>();
greatest.add(new PrettyCard(Card.diamond, 7));
```

? super

Supposons que nous ayons les classes suivantes :

```
class Greatest<T extends Comparable<? super T>> {
  /* ... */
  public void add(T element) { /* ... */ }
  public T get() { return element; }
}

class Card implements Comparable<Card> { /* ... */ }
class PrettyCard extends Card { /* ... */ }
```

Il est possible d'écrire les lignes suivantes car `PrettyCard` implémente l'interface `Comparable<Card>` :

```
Greatest<PrettyCard> greatest = new Greatest<PrettyCard>();
greatest.add(new PrettyCard(Card.diamond, 7));
```

? extends – Problématique

Supposons que nous ayons les classes suivantes :

```
class Greatest<T extends Comparable<? super T>> {
  /* ... */
  public void add(T element) { /* ... */ }
  public void addAll(List<T> list) {
    for (T element : list) add(element);
  }
}
```

Il n'est pas possible d'écrire les lignes suivantes :

```
List<PrettyCard> list = new ArrayList<PrettyCard>();
Greatest<Card> greatest = new Greatest<Card>();
/* ... */
list.addAll(list);
```

? extends

Supposons que nous ayons les classes suivantes :

```
class Greatest<T extends Comparable<? super T>> {
  /* ... */
  public void add(T element) { /* ... */ }
  public void addAll(List<? extends T> list) {
    for (T element : list) add(element);
  }
}
```

Il est maintenant possible d'écrire les lignes suivantes :

```
List<PrettyCard> list = new ArrayList<PrettyCard>();
Greatest<Card> greatest = new Greatest<Card>();
/* ... */
list.addAll(list);
```

Méthodes paramétrées et surcharge

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

Exemple :

```
String[] array = { "ezjf", "aaz", "zz" };
System.out.println(Tools.isSorted(array));
```

Plusieurs paramètres

```
public class Pair<A, B> {
  public A first;
  public B second;

  public Pair(A first, B second) {
    this.first = first;
    this.second = second;
  }

  public static <A, B> Pair<A,B> makePair(A first,
                                          B second) {
    return new Pair<A,B>(first, second);
  }
}
```

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> list = new ArrayList<Integer>();
/* code qui utilise list. */
```

car on peut changer la structure de données facilement :

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

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 (éléments comparables).

Structures de données Java

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.

Les exceptions

Un 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 ne doit pas être considérée comme un bug).

Quelques exemples de situations exceptionnelles :

- 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 de celles-ci afin d'en faciliter leur gestion.

En Java, une exception est concrétise par 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("error");
```

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

```
try { /* Problème possible */ }
catch (MyException e) { /* traiter l'exception. */ }
```

Définir son exception

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

```
public class MyException extends Exception {

  private int number;

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

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

}
```

La syntaxe try/catch

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

```
public static void test(int value) {
  System.out.print("A ");
  try {
    System.out.println("B ");
    if (value > 12) throw new MyException(value);
    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: Error 13
D
```

Exceptions et signatures des méthodes

Une méthode doit préciser dans sa signature toutes les exceptions qu'elle peut générer et qu'elle n'a pas traitées avec un bloc `try/catch` :

```
public class Test {

  public static void runTestValue(int value) throws MyException {
    testValue(value);
  }

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

  public static void main(String args[]) {
    try { runTestValue(13); }
    catch (MyException e) { e.printStackTrace(); }
  }
}
```

Pile d'appels

La méthode printStackTrace permet d'afficher la pile d'appels :

```
public class Test {
 public static void runTestValue(int value) throws MyException {
   testValue(value);
 }

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

 public static void main(String args[]) {
  try { runTestValue(13); } catch (MyException e) {e.printStackTrace();}
 }
}
```

```
MyException: Error 13
        at Test.testValue(Test.java:5)
        at Test.runTestValue(Test.java:2)
        at Test.main(Test.java:9)
```

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

Notez que ces exceptions s'apparentent le plus souvent à des bugs.

Capturer une exception en fonction de son type

Il est possible de capturer une exception en fonction de son type :

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

`divide(12,0)` :

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

`divide(null,12)` :

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

Le mot-clé finally

Le bloc associé au mot-clé `finally` est toujours exécuté :

```
public static void readFile(String fileName) {
  try {
    FileReader fileReader = new FileReader(fileName);
          /* peut déclencher une FileNotFoundException. */
    try {
      int character = fileReader.read(); /* IOException ? */
      while (character!=-1) {
        System.out.println(character);
        character = fileReader.read(); /* IOException ? */
      }
    } finally { fileReader.close(); /* à faire dans tous les cas. */ }
  } catch (IOException exception) { exception.printStackTrace(); }
}
```

`FileNotFoundException` étend `IOException` donc elle est capturée.

Le mot-clé finally

il est toujours possible de capturer les exceptions en fonction de leur type :

```
public static void readFile(String fileName) {
  try {
    FileReader fileReader = new FileReader(fileName);
          /* peut déclencher une FileNotFoundException. */
    try {
      int character = fileReader.read(); /* une IOException ? */
      while (character!=-1) {
        System.out.println(character);
        character = fileReader.read(); /* une IOException ? */
      }
    } finally { /* à faire dans tous les cas. */
          fileReader.close();
    }
  } catch (FileNotFoundException exception) {
    System.out.println("File "+fileName+" not found.");
  } catch (IOException exception) { exception.printStackTrace(); }
}
```

Exemple

```
public class Stack<T> {
  private Object[] stack;
  private int size;

  public Stack(int capacity) {
    stack = new Object[capacity]; size = 0;
  }

  public void push(T object) throws FullStackException {
    if (taille == stack.length) throw new FullStackException();
    pile[taille] = object; taille++;
  }

  public T pop() throws EmptyStackException {
    if (size == 0) throw new EmptyStackException();
    size--; T object = (T)stack[size]; stack[size]=null;
    return object;
  }
}
```

Exemple

Définition des exceptions :

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

```
public class FullStackException extends StackException {
    public FullStackException() { super("Full stack."); }
}
```

```
public class EmptyStackException extends StackException {
    public EmptyStackException() { super("Empty stack."); }
}
```

Exemple

Exemples d'utilisation :

```
Stack<Integer> stack = new Stack<Integer>(2);
```

```
try {
  stack.push(1);
  stack.push(2);
  stack.push(3);
} catch (StackException e) {
  e.printStackTrace();
}
```

```
FullStackException: Full stack.
        at Stack.push(Stack.java:8)
        at Test.main(Test.java:4)
```

```
try {
  stack.push(1);
  stack.pop();
  stack.pop();
} catch (StackException e) {
  e.printStackTrace();
}
```

```
EmptyStackException: Empty stack.
        at Stack.pop(Stack.java:14)
        at Test.main(Test.java:4)
```

Les énumérations

Il est possible de définir des énumérations :

```
enum Suit {
    SPADES,
    HEARTS,
    DIAMONDS,
    CLUBS;
}
```

Une énumération est une classe avec des éléments prédéfinis et statiques :

```
Suit suit = Suit.SPADES;
/* ... */
if (suit == Suit.SPADES) { /* .... */ }
```

Les énumérations

Définition de champs, de méthodes et d'un constructeur :

```
enum Suit {
  SPADES("Pique", "Pi"),
  HEARTS("Coeur", "Co"),
  DIAMONDS("Carreau", "Ca"),
  CLUBS("Trèfle", "Tr");

  private final String name;
  private final String symbol;

  Suit(String name, String symbol) {
    this.name = name;
    this.symbol = symbol;
  }

  public String name()   { return name; }
  public String symbol() { return symbol; }
}
```

Les énumérations

Un exemple d'utilisation de l'énumération précédente :

```
public static void main(String[] args) {
  for (Suit suit : Suit.values())
    System.out.printf("Le symbole de %s est %s",
                      suit.name(),
                      suit.symbol());
}
```

Un autre exemple :

```
public static void main(String[] args) {
  for (Suit suit : Suit.values())
    System.out.printf("La position de %s est %d",
                      suit.name(),
                      suit.ordinal());
}
```

Le mot-clé final

Première utilisation : interdire l'extension d'une classe

```
final public class Integer {

}
```

Deuxième utilisation : interdire la redéfinition d'une méthode

```
public class VariadicOperator {
  /* ... */
  final public double value() { /* ... */ }
  /* ... */
}
```

Le mot-clé final

Troisième utilisation : interdire la modification d'un attribut

```
final public class Integer {
  private final int value;

  public Integer(int value) { this.value = value; }
}
```

- L'attribut doit être initialisé après la construction de l'instance ;
- La valeur de l'attribut ne peut plus être modifiée ensuite.

Le mot-clé final

Quatrième utilisation : interdire la modification d'une variable

```
public class Stack<T> {
  /* ... */
  public T pop() {
    final T top = array[size-1];
    array[size-1] = null; size--;
    return top;
  }
  /* ... */
}
```

Les classes anonymes

Supposons que nous ayons l'interface suivante :

```
public interface ActionListener {
  public void actionPerformed(ActionEvent event);
}
```

Il est possible de :

- définir une classe anonyme qui implémente cette interface ;
- d'obtenir immédiatement une instance de cette classe

```
ActionListener listener = new ActionListener() {
  public void actionPerformed(ActionEvent event) {
    counter++;
  }
});
```

Les classes anonymes

```
public class Window {
  private int counter;

  public Window() {
    Button button = new Button("count");
    button.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent event) {
        counter++;
      }
    });
  }
}
```

Les classes anonymes

Il est possible d'utiliser des attributs de la classe "externe" :

```
public class Window {
  private Counter counter = new Counter();

  public Window() {
    Button button = new Button("count");
    button.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent event) {
        counter.count();
      }
    });
  }
}
```

Les classes anonymes

Il est possible d'utiliser des variables **finales** de la méthode :

```
public class Window {

  public Window() {
    final Counter counter = new Counter();
    Button button = new Button("count");
    button.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent event) {
        counter.count();
      }
    });
  }
}
```

Java 8 : Lambda expressions

```
public interface ActionListener {
  public void actionPerformed(ActionEvent event);
}
```

Avec Java 8, il est possible d'écrire directement :

```
public class Window {
  public Window() {
    Button button = new Button("button");
    button.addActionListener(
        event -> System.out.println(event)
    );
  }
}
```

Explication : `ActionListener` possède une seule méthode donc on peut affecter une lambda expression à une variable de type `ActionListener`.

Java 8 : Lambda expressions

Les trois interfaces suivantes sont définies :

```
public interface Predicate<T> {
  public boolean test(T t);
}

public interface Function<T,R> {
  public R apply(T t);
}

public interface Consumer<T> {
  void accept(T t);
}
```

Java 8 : Lambda expressions

On peut écrire directement :

```
persons
   .stream()
   .filter(person -> person.getAge() >= 18)
   .map(person -> person.getName())
   .forEach(name -> System.out.println(name));
```

Types des paramètres et retours des méthodes :

- `stream()` → `Stream<Person>`
- `filter(Predicate<Person>)` → `Stream<Person>`
- `map(Function<Person, String>)` → `Stream<String>`
- `forEach(Consumer<String>)`

Java 8 : Faire référence à une méthode

On peut faire référence à une méthode :

```
persons
   .stream()
   .map(Person::getName) /* person->person.getName() */
   .forEach(name -> System.out.println(name));
```

Un autre exemple :

```
String[] strings = { "Truc", "Machin", "Bidule" };
Arrays.sort(strings, String::compareToIgnoreCase);
                  /* (s1,s2)->s1.compareToIgnoreCase(s2)); */
```

Il est également possible de (Java Tutorials) :

- faire référence à une méthode d'une instance ;
- faire référence à un construction.

Révision – Les mots réservés de Java

- Gestions des paquets :

  `package`
  `import`

- Définitions des classes, interfaces et énumérations :

  `class`
  `interface`
  `enum`

- Héritage :

  `extends`
  `implements`

Révision – Les classes

```
class Counter {

  int counter;

  Counter(int initialValue) {
    counter = initialValue;
  }

  void count() {
    counter++;
  }

}
```

Révision – Interfaces

```
interface Printer {
  /**
   * Imprime la chaîne s.
   * @param s chaîne à imprimer
   */
  void print(String document);

  /**
   * Permet de savoir si l'instance est prête.
   * @return true si l'instance est prête, false sinon
   */
  boolean isReady();
}
```

Révision – Implémentation d'une interface

```
class MyPrinter implements Printer {
  String name;

  MyPrinter(String name) { this.name = name; }

  @Override
  void print(String document) {
    System.out.println(name+" : "+document);
  }

  @Override
  boolean isReady() { return true; }
}
```

Révision – Extension

```
class MyBracePrinter extends MyPrinter {

 MyBracePrinter(String name) {
   super(name);
 }

 @Override
  void print(String document) {
    System.out.println("{"+super.print(document)+"}");
  }

}
```

Révision – Polymorphisme

Qu'affiche ce programme ?

```
public static void main(String args[]) {
  Printer printer;
  if (args[0].length()>0)
    printer = new MyBracePrinter("bracePrinter");
  else printer = new MyPrinter("printer");
  printer.print("document");
}
```

Révision – Les mots réservés

Modificateurs pour les membres et les variables :

```
public
protected
private
static
final
abstract
throws
```

Révision – Modification de la visibilité

	Classe	Paquet	Extension	Extérieur
Private	✓			
Default	✓	✓		
Protected	✓	✓	✓	
Public	✓	✓	✓	✓

Révision – Données et méthodes statiques

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

```
class Counter {
 private static int counter = 0;
 static void count() { counter++; }
 static int value() { return counter; }
}
```

Un exemple d'utilisation de la classe précédente :

```
Counter.count();
Counter.count();
Counter.count();
System.out.println(Counter.value());
```

Classes abstraites

Une classe est abstraite si des méthodes ne sont pas implémentées :

```
public abstract class List {
  private int[] list = new int[10];
  private int size = 0;

  public void add(int value) { list[size] = value; size++; }

  public int eval() {
    int result = neutral(); // utilisation d'une méthode abstraite
    for (int i = 0; i < size; i++)
      result = compute(result, list[i]); // idem
    return result;
  }

  public abstract int neutral(); // méthode abstraite
  public abstract int compute(int a, int b); // idem
}
```

Classes abstraites et extension

Évidemment, il n'est pas possible d'instancier une classe abstraite :

```
List list = new List(); // erreur
System.out.println(list.eval()) // car que faire ici ?
```

Nous allons étendre la classe `List` afin de récupérer les propriétés et les méthodes de `List` et définir le code des méthodes abstraites :

```
public class ListSum extends List {
  public int neutral() { return 0; }
  public int compute(int a, int b) { return a+b; }
}
```

La classe `ListSum` n'est plus abstraite, toutes ses méthodes sont définies :

```
ListSum listSum = new ListSum();
listSum.add(3); listSum.add(7);
System.out.println(listSum.eval());
```

Classes abstraites et extension

Évidemment, il n'est pas possible d'instancier une classe abstraite :

```
List list = new List(); // erreur
System.out.println(list.eval()) // car que faire ici ?
```

Nous allons étendre la classe `List` afin de récupérer les propriétés et les méthodes de `List` et définir le code des méthodes abstraites :

```
public class ListProduct extends List {
  public int neutral() { return 1; }
  public int compute(int a, int b) { return a*b; }
}
```

La classe `ListSum` n'est plus abstraite, toutes ses méthodes sont définies :

```
ListProduct listProduct = new ListProduct();
listProduct.add(3); listProduct.add(7);
System.out.println(listProduct.eval());
```

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
instanceof
```

Révision – Les mots réservés et 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} 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 intValue = 12;
double doubleValue = 13.5;
boolean booleanValue = true;
```

Révision – Les autres mots réservés

```
assert
goto
native
assert
strictfp
volatile
transient
synchronized
```

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 Counter implements Runnable {

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

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

```
public class Main  {
    public static void main(String[] args) {
      Thread counter1 = new Thread(new Counter()); counter1.start();
      Thread counter2 = new Thread(new Counter()); counter2.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 Counter extends Thread {
    public void run() {
        int counter = 0;
        while (counter < 10) {
          System.out.print(counter+" "); counter++;
        }
    }
}
```

```
public class Main  {
    public static void main(String[] args) {
        Thread counter1 = new Counter(); counter1.start();
        Thread counter2 = new Counter(); counter2.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 Counter extends Thread {
    public void run() {
        int counter = 0;
        while (!interrupted()) counter++;
        System.out.println(counter);
    }
}
```

L'état d'interruption

Utilisation du compteur :

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

Compteur qui compte moins vite :

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

L'état d'interruption

Utilisation du compteur :

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

Compteur qui compte moins vite :

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

L'état d'interruption

Thread qui interrompt un autre thread après un certain temps :

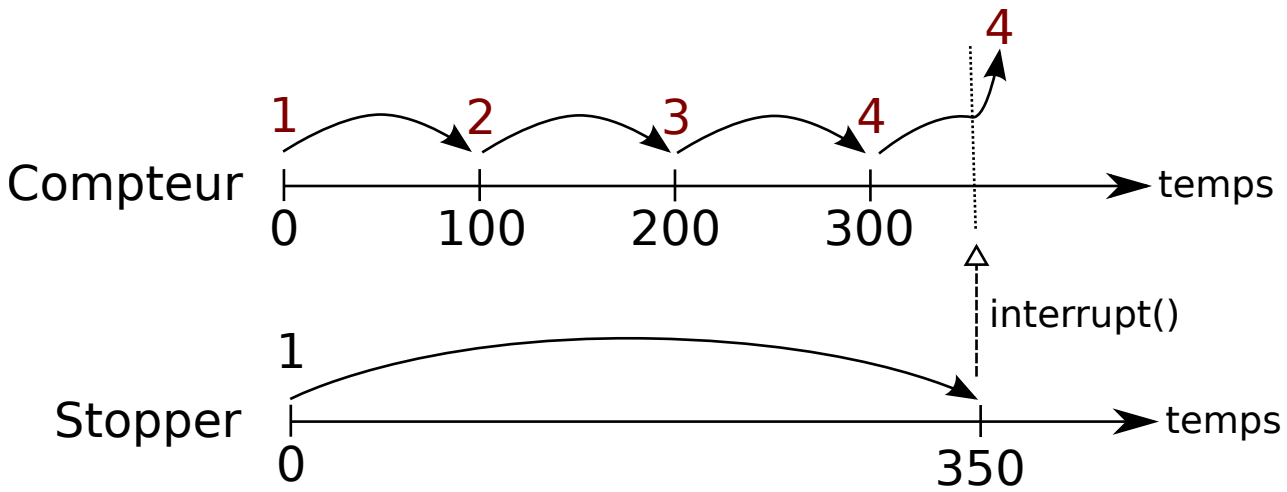
```
public class Stopper extends Thread {

    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 counter = new Counter();
        counter.start();
        Thread stopper = new Stopper(counter, 350);
        stopper.start();
}
```

Exécution :



Problème de synchronisation

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

```
public class SlowStack {
    private int[] array; private int size;

    public SlowStack() {
        array = new int[1000]; size = 0;
    }

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

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

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

Synchronisation

Un thread qui empile 10 entiers :

```
public class Stacker extends Thread {

    private SlowStack stack;

    public Stacker(SlowStack stack) {
        this.stack = stack;
    }

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

Synchronisation

Exécution en parallèle de deux empileurs sur une même pile :

```
public static void main(String[] args)
      throws InterruptedException {
    SlowStack stack = new SlowStack();
    Thread stacker1 = new Stacker(stack);
    Thread stacker2 = new Stacker(stack);
    stacker1.start();
    stacker2.start();
    stacker1.join(); // On attend la fin du thread 1
    stacker2.join(); // On attend la fin du thread 2
    while (stack.size()>0) {
        System.out.print(stack.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)	
array[s] = p ;	
sleep(100) ;	int s = size ; (s = x)
– –	array[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 SlowStack {
    ...
    synchronized void push(int value) throws InterruptedException {
        int s = size; tab[s] = value;
        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);
```

UML et diagramme de classes

l'Unified Modeling Language :

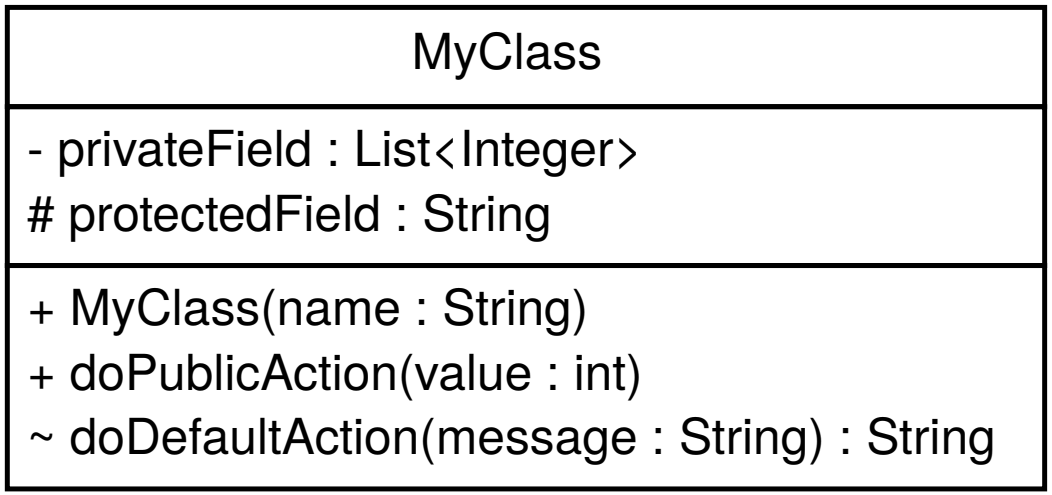
- est un langage de modélisation graphique ;
- est apparu dans le cadre de la conception orientée objet ;
- propose 13 types de diagrammes qui permettent la modélisation d'un projet durant tout son cycle de vie ;
- est souvent abrégé par le sigle UML.

Les diagrammes de classes :

- font partie d'UML ;
- sont des schémas utilisés pour représenter les classes et les interfaces ;
- permettent de représenter les interactions entre les classes ;
- sont statiques : on fait abstraction des aspects temporels ;
- permettent de réfléchir à la structure du programme ;
- permettent de décrire la structure du programme.

Schéma d'une classe

Un exemple de schéma représentant une classe :

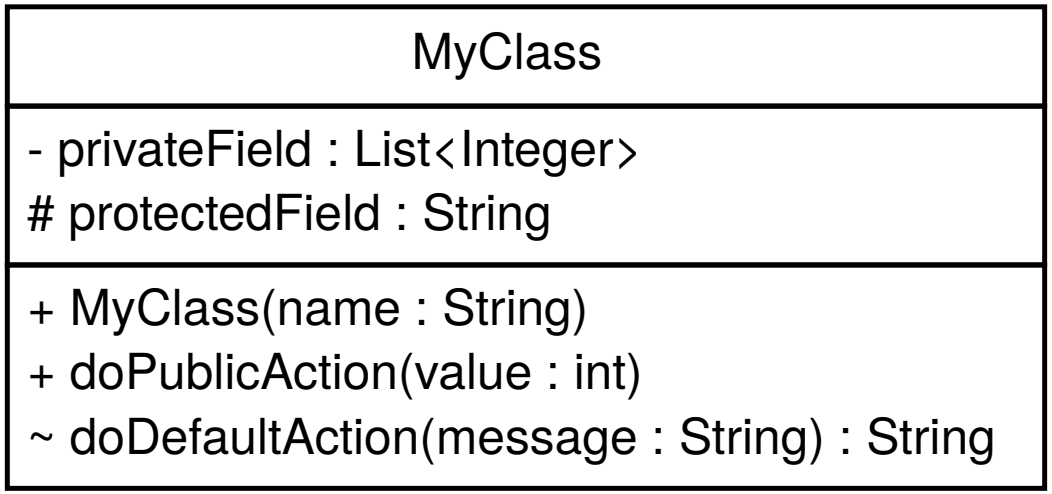


Les règles d'écriture des champs et des méthodes :

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

Droits d'accès (ou visibilité)

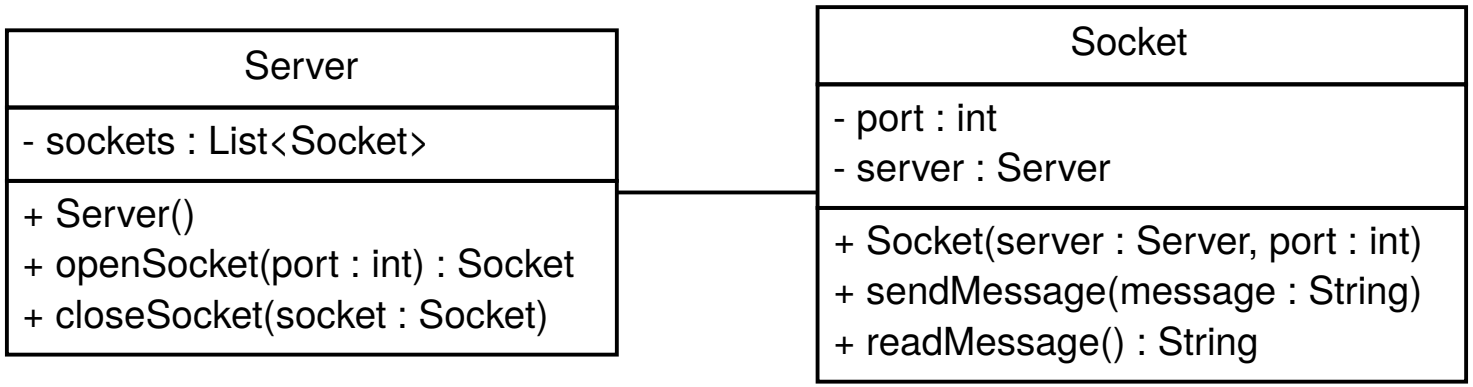
Un exemple illustrant les différents droits d'accès :



		Classe	Paquet	Extension	Extérieur
Private	-	✓			
Default	~	✓	✓		
Protected	#	✓	✓	✓	
Public	+	✓	✓	✓	✓

Associations

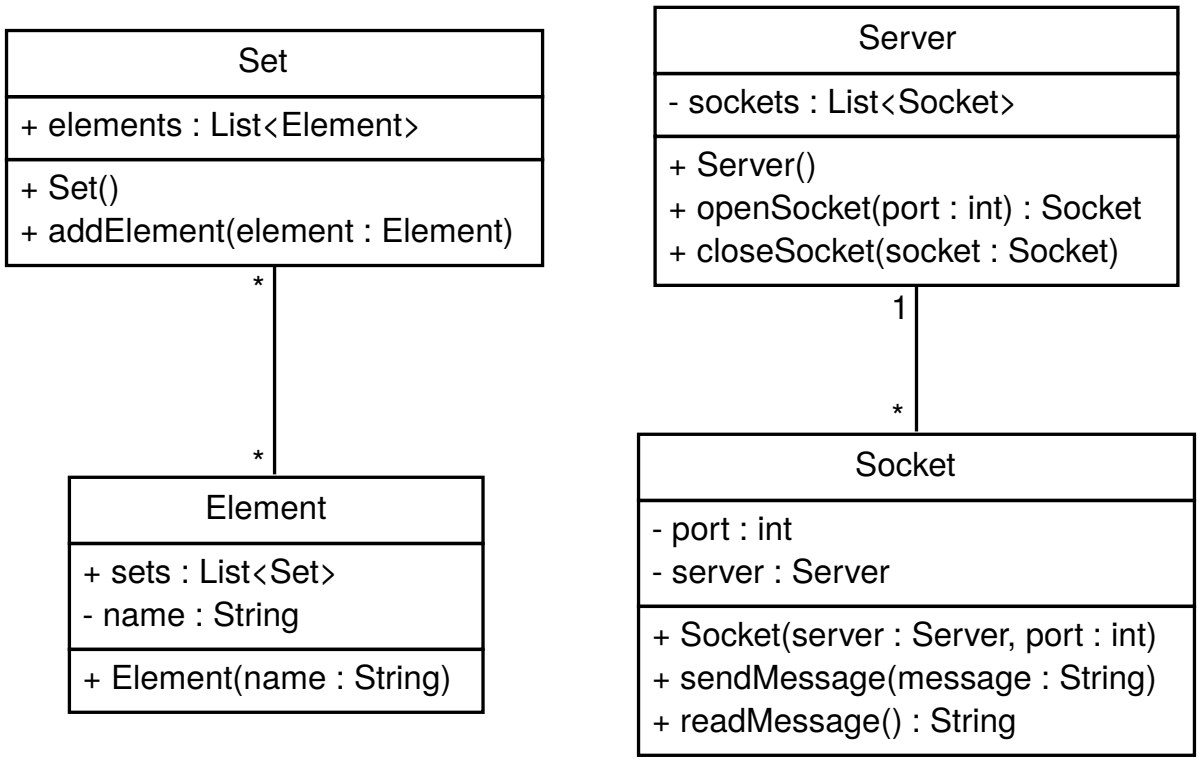
Schéma représentant une association :



Une association exprime une relation sémantique entre deux classes

Cardinalités

Cardinalités :



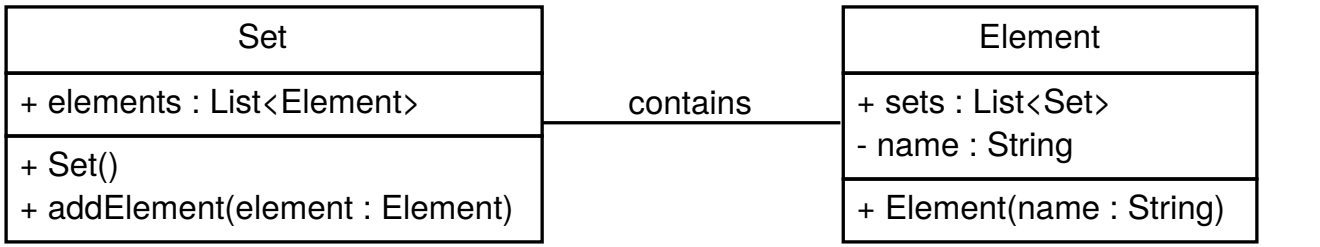
- Permet d'indiquer le nombre d'instances qui participent à l'association

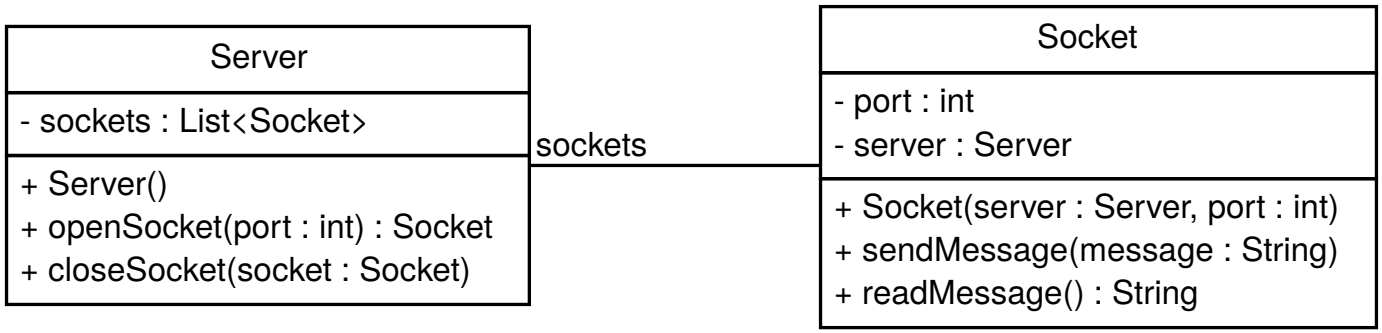
Cardinalités

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*

Nommage des relations

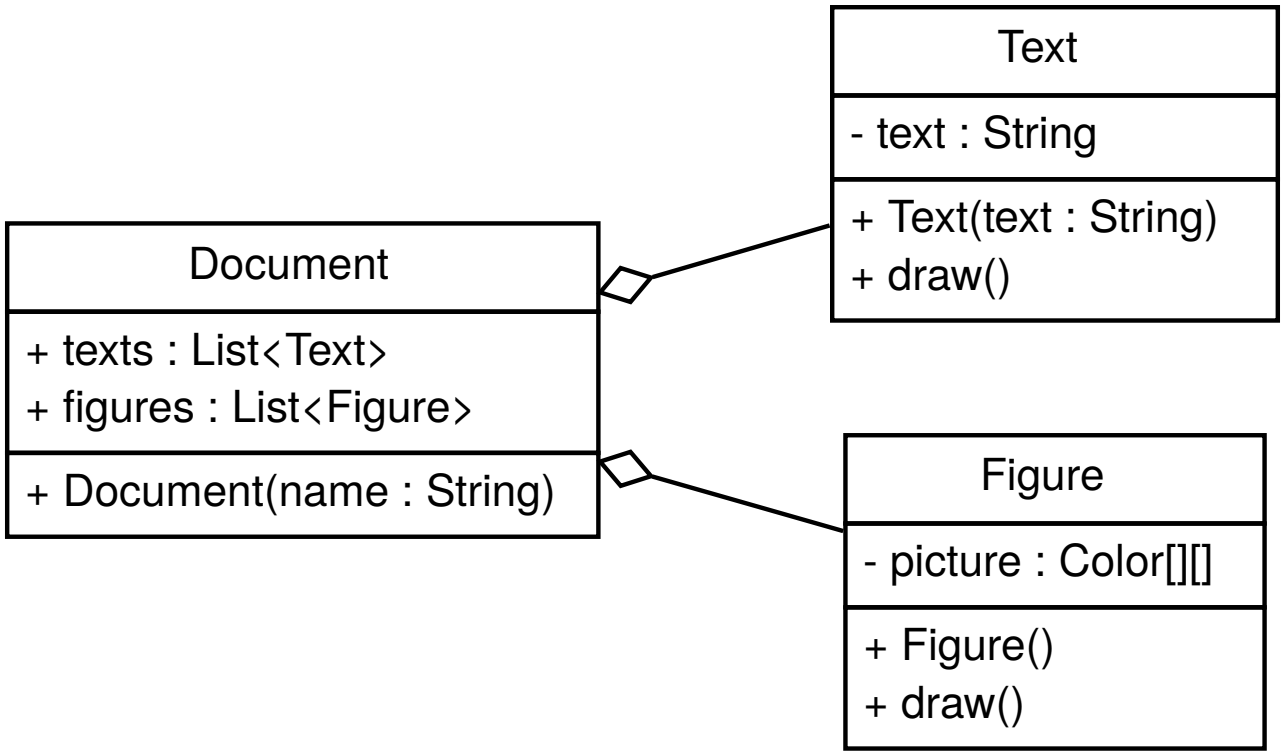
On peut préciser la sémantique d'une association :





Agrégations

Un schéma représentant une agrégation :

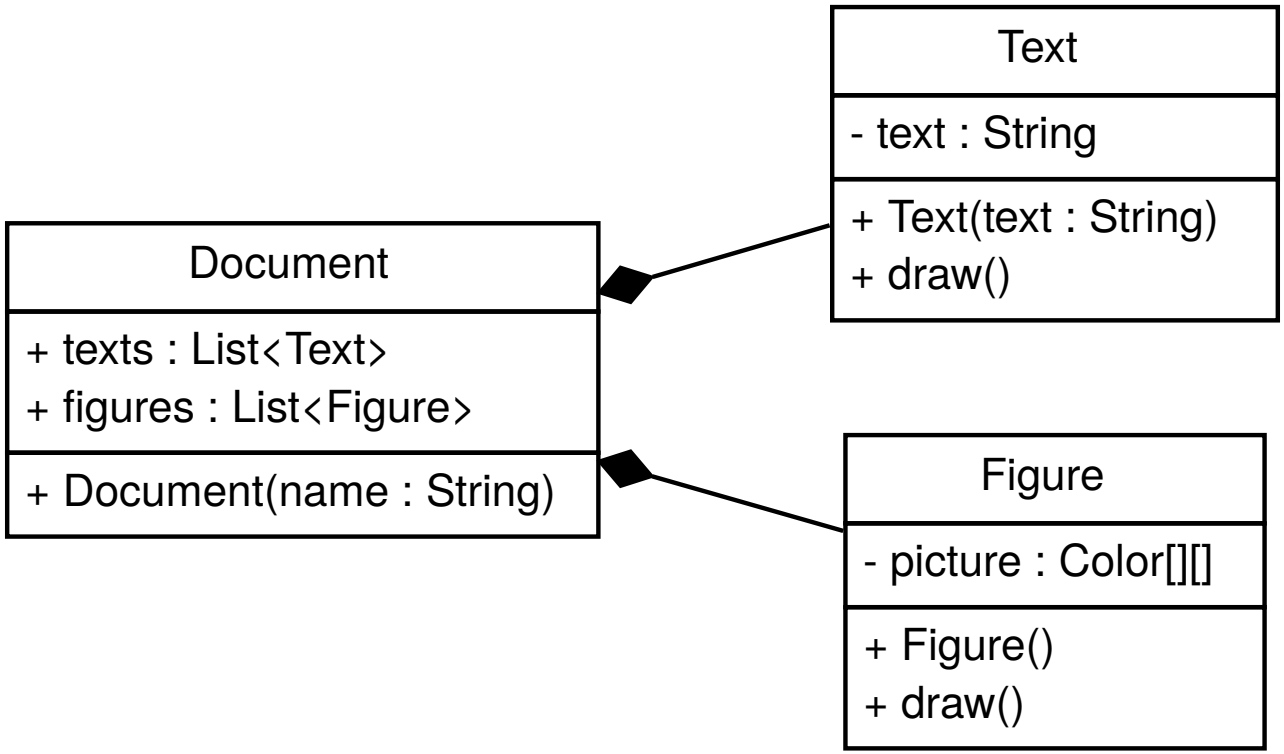


Signification :

- Le document possède des figures et des textes

Compositions

Un schéma représentant une composition :

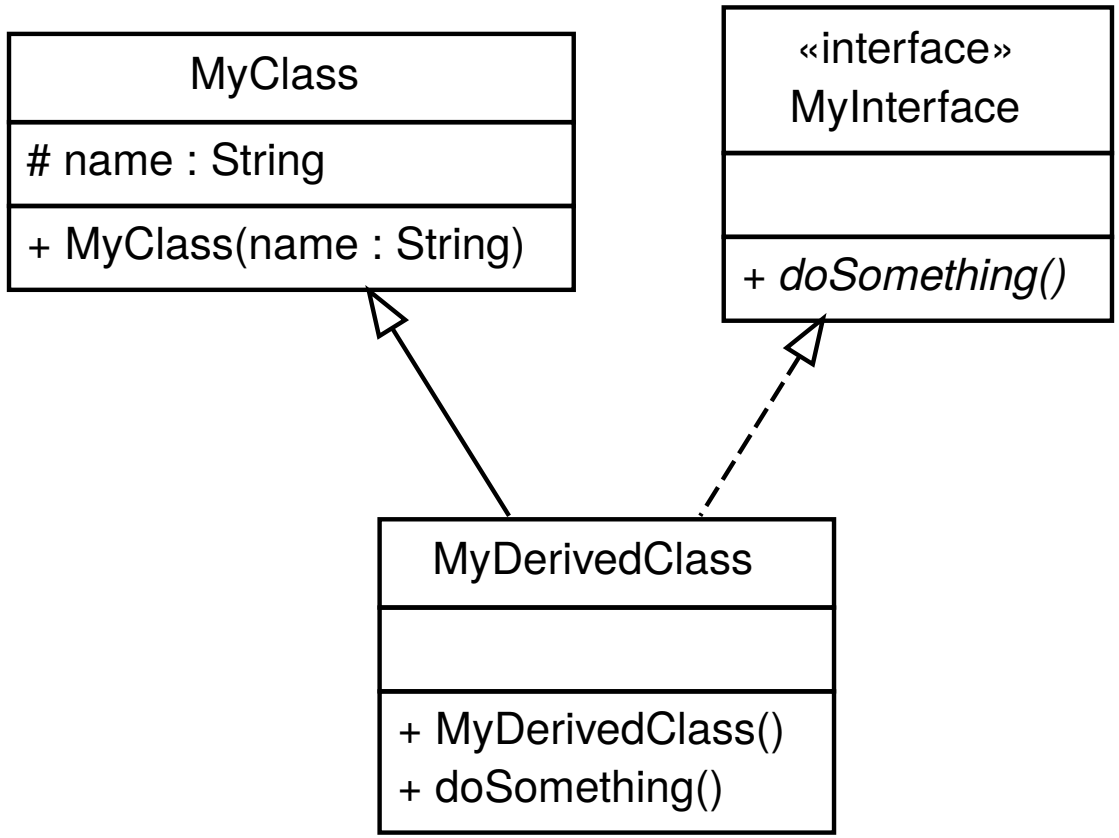


Signification :

- Le document est composé de figures et de textes
- Les figures et les textes sont détruits si le document est détruit

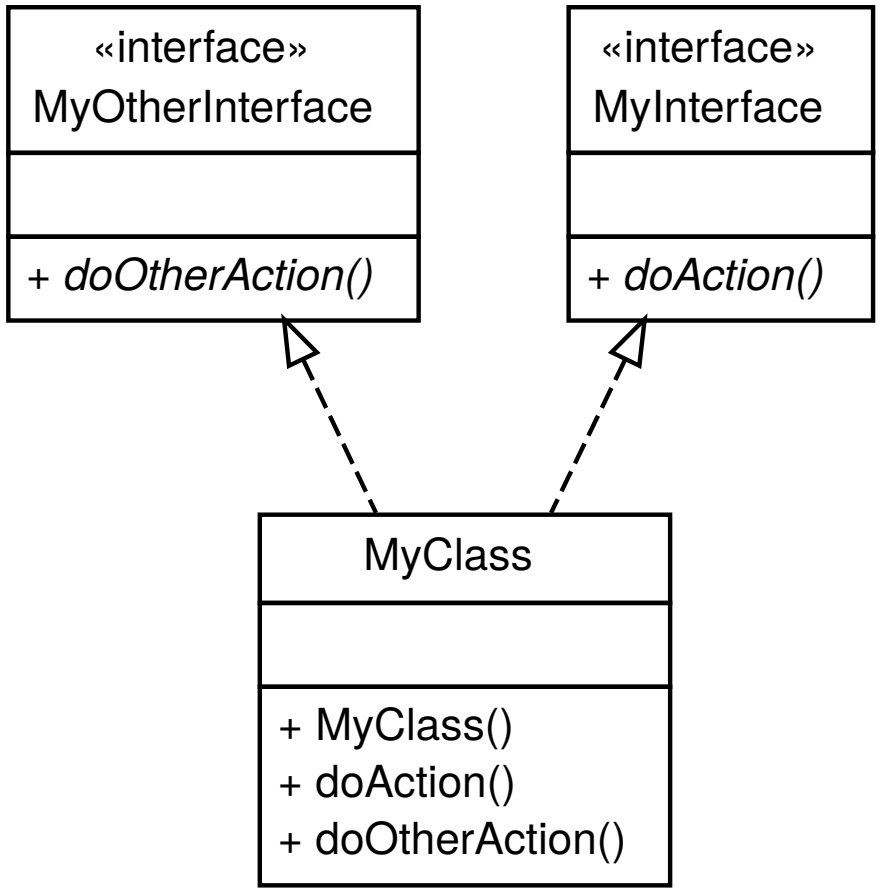
Interfaces, extensions et implémentations

Une interface, une extension et une implémentation :



Implémentation multiples

Une implémentation multiple :



Classes et méthodes abstraites

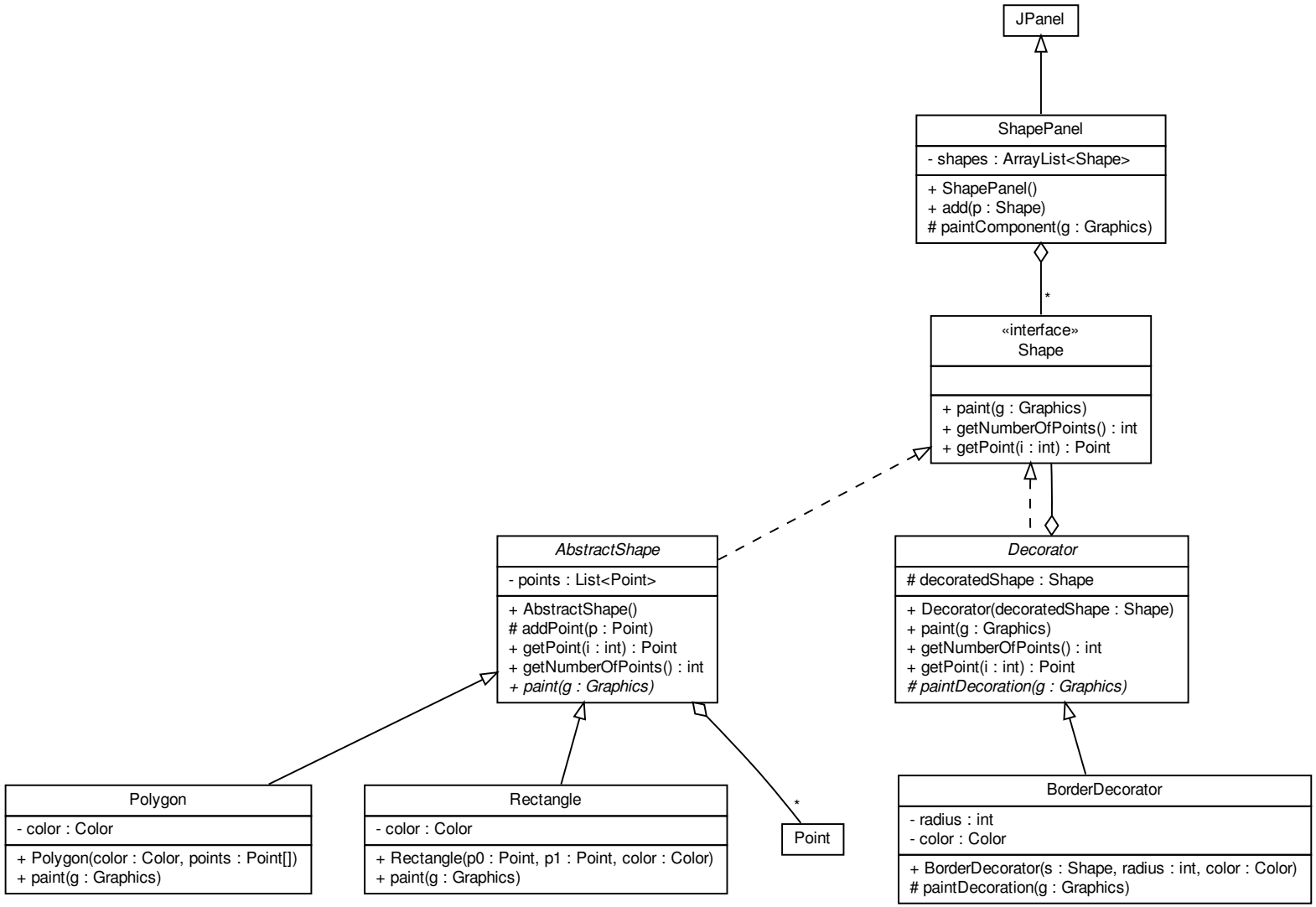
Un exemple de schéma avec une classe et une classe abstraite :

MyAbstractClass
- privateField : List<Integer> # protectedField : String
+ MyAbstractClass(name : String) + doPublicAction(value : int) *# doAbstractAction(message : String)*

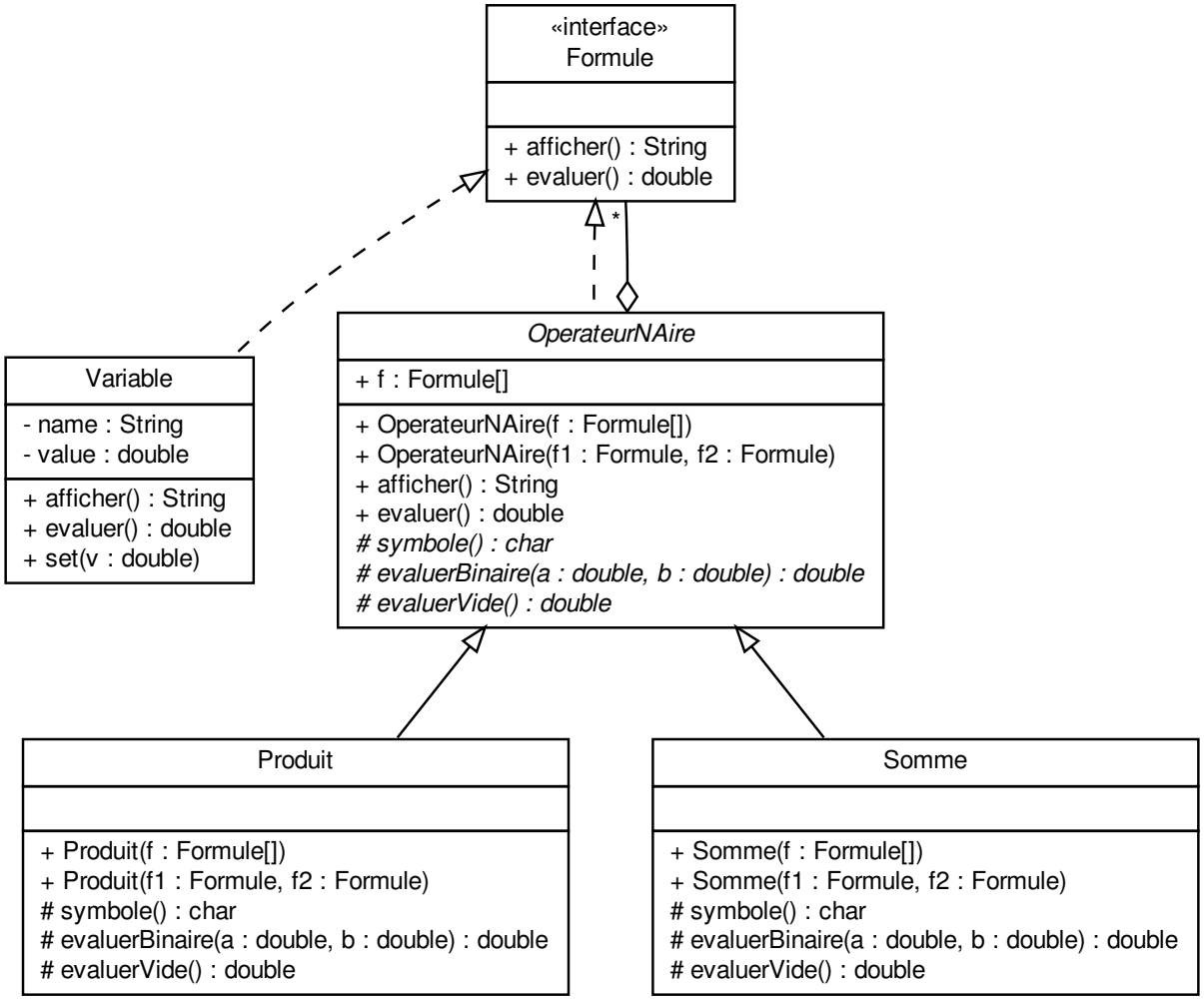
MyClass
- privateField : List<Integer> # protectedField : String
+ MyClass(name : String) + doPublicAction(value : int) ~ doDefaultAction(message : String) : String

À droite, la méthode `doAbstractAction` est écrit en italique car elle est abstraite. La classe est donc abstraite et son nom est aussi écrit en italique.

Exemple de diagramme de classes



Exemple de diagramme de classes



Un programme “propre”

Clean Code (Robert C. Martin)

Un programme “propre” :

- respecte les attentes des utilisateurs ;
- est fiable ;
- peut évoluer facilement/rapidement ;
- est compréhensible.

En résumé :

- Un programme informatique est de qualité si l’effort nécessaire à l’ajout d’une nouvelle fonctionnalité par un développeur extérieur au projet est faible.

Pourquoi ?

- Pour programmer les fonctionnalités les unes après les autres
- Pour ajouter des fonctionnalités à moindre coût
- Pour que les programmes soient utilisables plus longtemps
- Pour valoriser les codes écrits par les développeurs (car réutilisables)
- Pour effectuer des tests à toutes les étapes du développement
- Pour que les développeurs soient heureux de travailler

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
  - $\Rightarrow$ Rester conforme aux spécifications qui n’ont pas changé
  - $\Rightarrow$ Faciliter l’intégration
- Gagner du temps

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).

En d'autres termes :

- 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
  - de perdre du temps

Single Responsibility Principle (SRP)

Principe :

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

En d'autres termes :

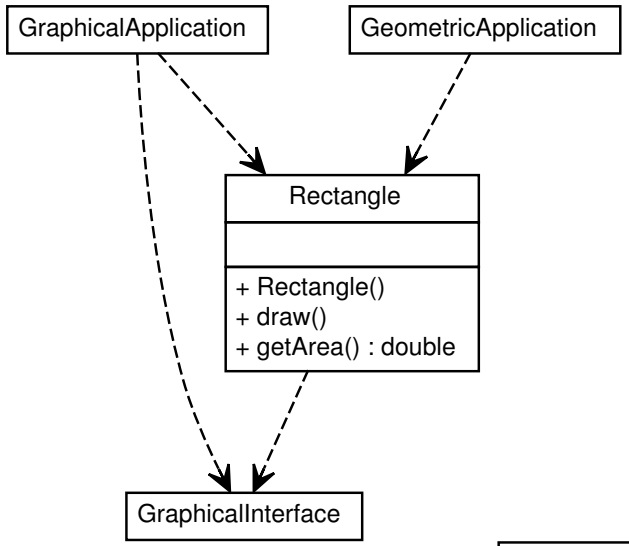
- Une responsabilité est une **"raison de changer"**
- Une classe ne doit avoir qu'une seule raison de changer

Avantages :

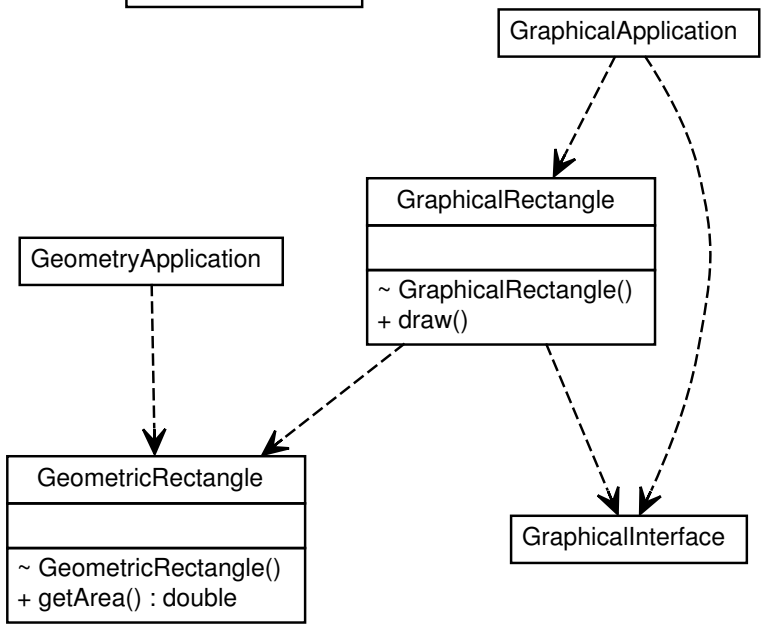
- Diminution de la complexité du code
- Amélioration de la lisibilité du code
- Meilleure organisation du code
- Modification locale lors des évolutions
- Augmentation de la fiabilité
- Les classes ont plus de chance d'être réutilisables

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)

Considérons les deux classes suivantes :

```
public class Circle {
  public Point center;
  public int radius;

  public Circle(Point center, int radius) {
    this.center = center;
    this.radius = radius;
 }
}
```

```
public class Rectangle {
  public Point point1, point2;

  public Rectangle(Point point1, Point point2) {
    this.point1 = point1;
   this.point2 = point2;
  }
}
```

Open/Closed Principle (OCP)

La méthode suivante ne respectent pas OCP :

```
public class GraphicTools {
  static void draw(Graphics graphics, List<Object> objects) {
    for (Object object : objects) {
      if (object instanceof Rectangle) {
        Rectangle rectangle = (Rectangle) object;
        int x = Math.min(rectangle.point1.x, rectangle.point2.x);
        int y = Math.min(rectangle.point1.y, rectangle.point2.y);
        int width = Math.abs(rectangle.point1.x - rectangle.point2.x);
        int height = Math.abs(rectangle.point1.y - rectangle.point2.y);
        graphics.drawRect(x, y, width, height);
      } else if (object instanceof Circle) {
        /* Voir le slide suivant */
      }
    }
  }
}
```

Open/Closed Principle (OCP)

La méthode suivante ne respectent pas OCP :

```
public class GraphicTools {
  static void draw(Graphics graphics, List<Object> objects) {
    for (Object object : objects) {
      if (object instanceof Rectangle) {
        /* Voir le slide précédent */
      } else if (object instanceof Circle) {
        Circle circle = (Circle) object;
        int x = circle.center.x - circle.radius;
        int y = circle.center.y - circle.radius;
        int width = circle.radius * 2;
        int height = circle.radius * 2;
        graphics.drawOval(x, y, width, height);
      }
    }
  }
}
```

Open/Closed Principle (OCP)

Première étape : simplifier le code de la méthode.

```
public class GraphicTools {
  static void draw(Graphics graphics, List<Object> objects) {
    for (Object object : objects) {
      if (object instanceof Rectangle) {
        Rectangle rectangle = (Rectangle)object;
        drawRectangle(graphics, rectangle);
      } else if (object instanceof Circle) {
        Circle circle = (Circle)object;
        drawCircle(graphics, circle);
      }
    }
  }

  /* Voir slide suivant */
}
```

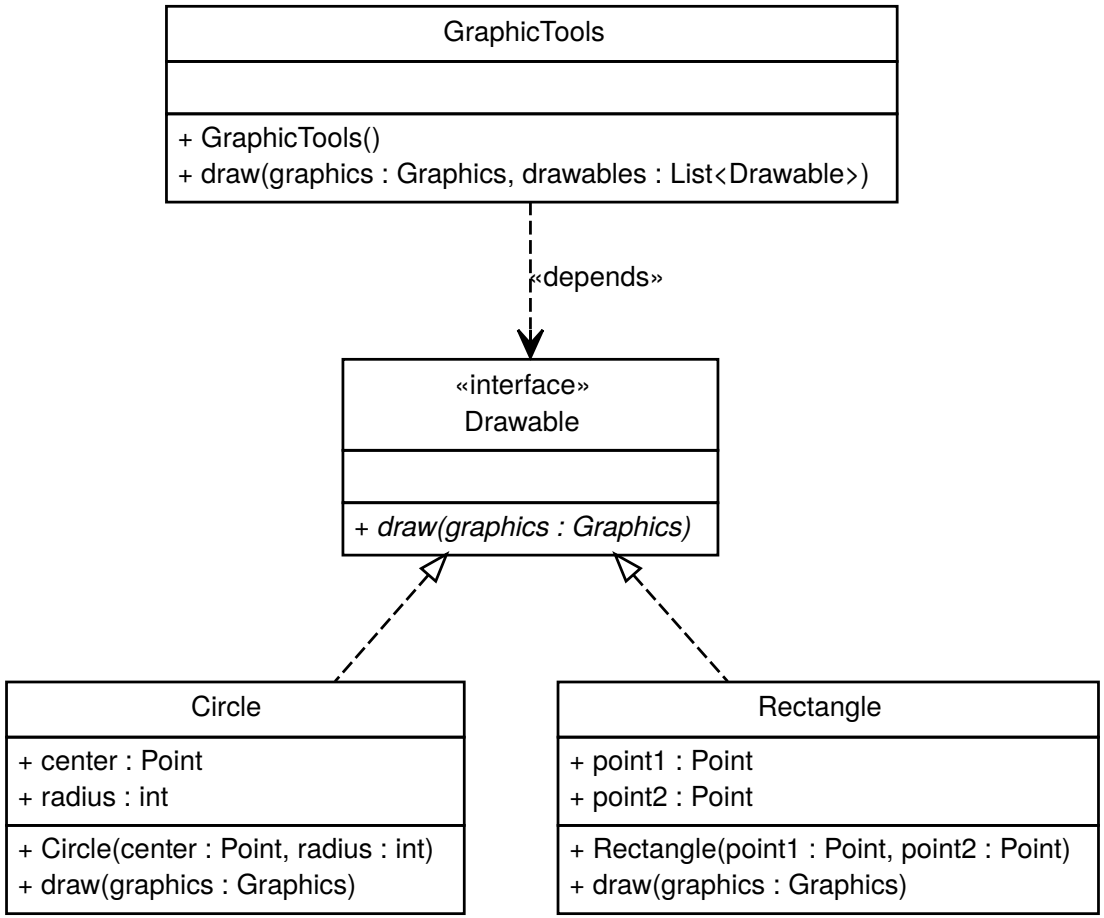
Open/Closed Principle (OCP)

Première étape : simplifier le code de la méthode.

```
public class GraphicTools { /* Voir slide précédent */

  static void drawRectangle(Graphics graphics, Rectangle rectangle) {
    int x = Math.min(rectangle.point1.x, rectangle.point2.x);
    int y = Math.min(rectangle.point1.y, rectangle.point2.y);
    int width = Math.abs(rectangle.point1.x -  rectangle.point2.x);
    int height = Math.abs(rectangle.point1.y -  rectangle.point2.y);
    graphics.drawRect(x, y, width, height);
  }

  static void drawCircle(Graphics graphics, Circle circle) {
    int x = circle.center.x - circle.radius;
    int y = circle.center.y - circle.radius;
    int width = circle.radius * 2;
    int height = circle.radius * 2;
    graphics.drawOval(x, y, width, height);
  }
}
```

Open/Closed Principle (OCP)

Première solution avec une interface et de l'abstraction :



Open/Closed Principle (OCP)

Implémentation de la première solution :

```
public interface Drawable {
  void draw(Graphics graphics);
}
```

```
public class GraphicTools {
  static void draw(Graphics graphics, List<Drawable> drawables) {
    for (Drawable drawable : drawables)
      drawable.draw(graphics);
      // ou drawables.forEach(drawable->drawable.draw(graphics));
  }
}
```

Open/Closed Principle (OCP)

Implémentation de la première solution :

```
public class Circle implements Drawable {
  public Point center;
  public int radius;

  public Circle(Point center, int radius) {
    this.center = center;
    this.radius = radius;
  }

  public void draw(Graphics graphics) {
    int x = center.x - radius;
    int y = center.y - radius;
    int width = radius * 2;
    int height = radius * 2;
    graphics.drawOval(x, y, width, height);
  }
}
```

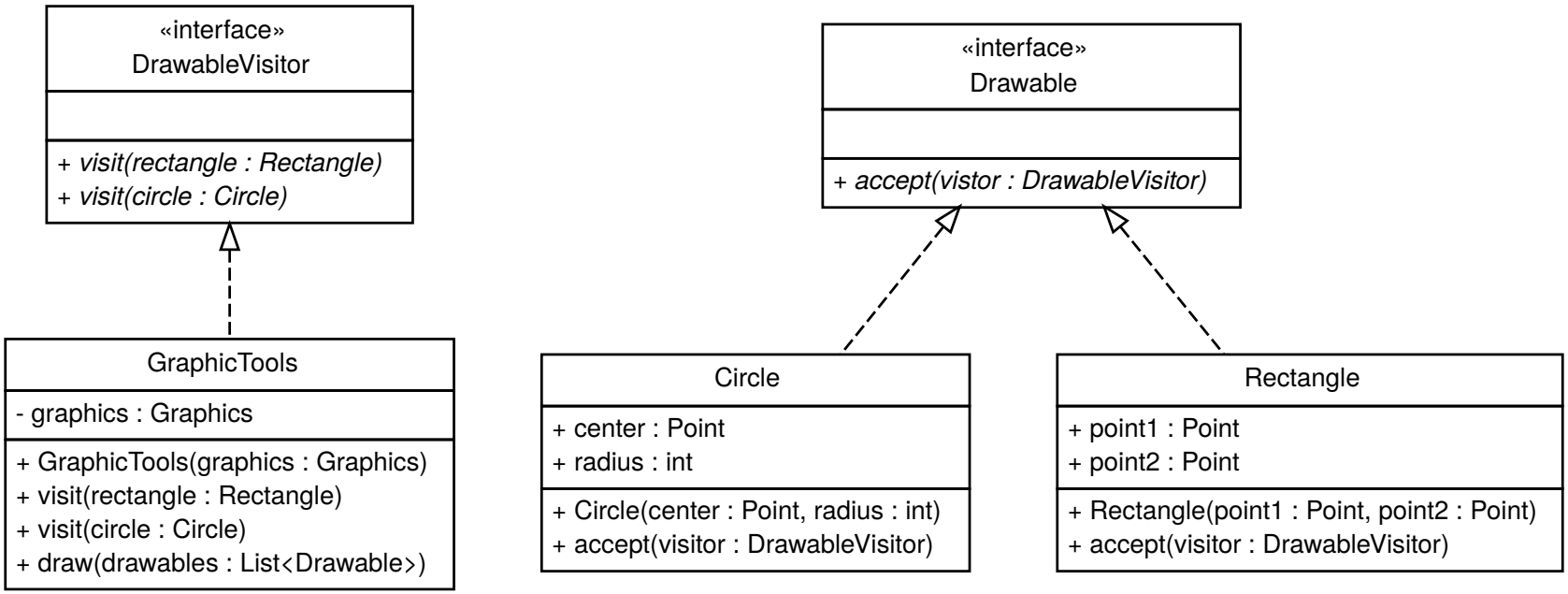
Open/Closed Principle (OCP)

Implémentation de la première solution :

```
public class Rectangle implements Drawable {
  public Point point1, point2;

  public Rectangle(Point point1, Point point2) {
    this.point1 = point1;
    this.point2 = point2;
  }

  public void draw(Graphics graphics) {
    int x = Math.min(point1.x, point2.x);
    int y = Math.min(point1.y, point2.y);
    int width = Math.abs(point1.x - point2.x);
    int height = Math.abs(point1.y - point2.y);
    graphics.drawRect(x, y, width, height);
  }
}
```

Open/Closed Principle (OCP)

Deuxième solution avec le patron de conception “visiteurs” :



- `draw` de `GraphicsTools` invoque `accept`
- `accept` de `Circle` invoque `visit(Circle circle)`
- `accept` de `Rectangle` invoque `visit(Rectangle rectangle)`
- On invoque toujours une méthode d’une interface

Open/Closed Principle (OCP)

Implémentation de la deuxième solution :

```
public interface Drawable {
  void accept(DrawableVisitor vistor);
}
```

```
public interface DrawableVisitor {
  public void visit(Rectangle rectangle);
  public void visit(Circle circle);
}
```

Open/Closed Principle (OCP)

Implémentation de la deuxième solution :

```
public class Circle implements Drawable {
  public Point center;
  public int radius;

  public Circle(Point center, int radius) {
    this.center = center;
    this.radius = radius;
  }

  public void accept(DrawableVisitor visitor) {
    visitor.visit(this);
  }
}
```

Open/Closed Principle (OCP)

Implémentation de la deuxième solution :

```
public class Rectangle implements Drawable {
  public Point point1, point2;

  public Rectangle(Point point1, Point point2) {
    this.point1 = point1;
    this.point2 = point2;
  }

  public void accept(DrawableVisitor visitor) {
    visitor.visit(this);
  }
}
```

Open/Closed Principle (OCP)

Implémentation de la deuxième solution :

```
public class GraphicTools implements DrawableVisitor {

  private Graphics graphics;

  public GraphicTools(Graphics graphics) {
    this.graphics = graphics;
  }

  public void draw(List<Drawable> drawables) {
    for (Drawable drawable : drawables)
      drawable.accept(this);
  }

  /* Voir slide suivant pour la suite */
}
```

Open/Closed Principle (OCP)

Implémentation de la deuxième solution :

```
public class GraphicTools implements DrawableVisitor {
  public void visit(Rectangle rectangle) {
    int x = Math.min(rectangle.point1.x, rectangle.point2.x);
    int y = Math.min(rectangle.point1.y, rectangle.point2.y);
    int width = Math.abs(rectangle.point1.x -  rectangle.point2.x);
    int height = Math.abs(rectangle.point1.y -  rectangle.point2.y);
    graphics.drawRect(x, y, width, height);
  }

  public void visit(Circle circle) {
    int x = circle.center.x - circle.radius;
    int y = circle.center.y - circle.radius;
    int width = circle.radius * 2;
    int height = circle.radius * 2;
    graphics.drawOval(x, y, width, height);
  }
}
```

Open/Closed Principle (OCP)

Comparaison des deux solutions :

- Première solution :
  - ✓ Simplicité du code
  - ✓ Simplicité lors de l'ajout d'une nouvelle forme
  - ✗ Couplage dessin/forme
  - ✗ Une seule façon de dessiner par forme
  - ✗ Méthodes de dessin éclatées dans de nombreuses classes
- Deuxième solution :
  - ✓ Les méthodes de dessin sont regroupées dans une classe
  - ✓ Possibilité d'avoir plusieurs façons de dessiner les formes
  - ✗ Code plus compliqué à lire
  - ✗ Nombreuses classes à modifier lors de l'ajout d'une forme

Le choix de l'implémentation d'une fonctionalité doit dépendre des probables évolutions de l'application.

Open/Closed Principle (OCP)

Un deuxième exemple qui viole SRP et OCP :

```
public class ItemList {
  private List<String> items;

  public ItemList() { items = new ArrayList<String>(); }

  public void add(String item) { items.add(item); }

  /* Slide suivant. */
}
```

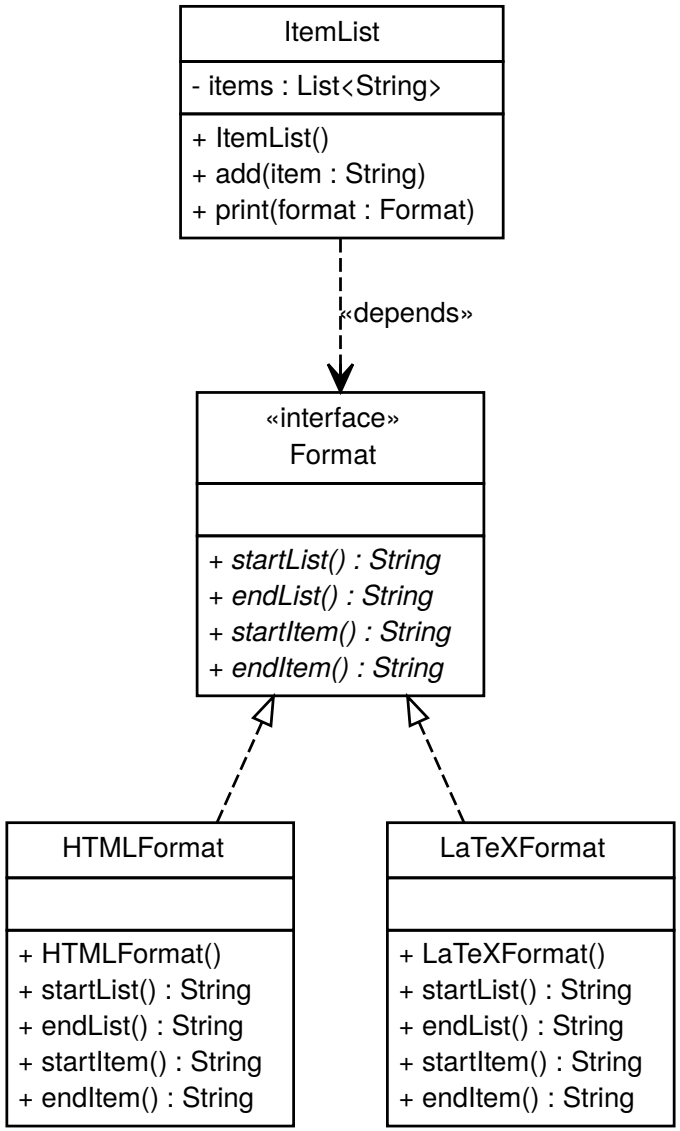
Open/Closed Principle (OCP)

Un deuxième exemple qui viole SRP et OCP :

```
public class ItemList {
  /* Slide précédent. */

  public void printHTML() {
    System.out.println("<ul>");
    for (String item : items)
      System.out.println("<li>"+item+"</li>");
    System.out.println("</ul>");
  }
  public void printLaTeX() {
    System.out.println("\\begin{itemize}");
    for (String item : items)System.out.println("\\item "+item);
    System.out.println("\\end{itemize}");
  }
}
```

Open/Closed Principle (OCP)

Une solution :



Open/Closed Principle (OCP)

Implémentation de la solution :

```
public interface Format {
  public String startList();
  public String endList();
  public String startItem();
  public String endItem();
}
```

Open/Closed Principle (OCP)

Implémentation de la solution :

```
public class ItemList {
  private List<String> items;

  public ItemList() { items = new ArrayList<String>(); }
  public void add(String item) { items.add(item); }

  public void print(Format format) {
    System.out.println(format.startList());
    for (String item : items)
          System.out.println(format.startItem()
                             +item
                             +format.endItem());
    System.out.println(format.endList());
  }

}
```

Open/Closed Principle (OCP)

Implémentation de la solution :

```
public class HTMLFormat implements Format {
  public String startList() { return "<ul>"; }
  public String endList()   { return "</ul>"; }
  public String startItem()   { return "<li>"; }
  public String endItem()   { return "</li>"; }
}
```

```
public class LaTeXFormat implements Format {
  public String startList() { return "\\begin{itemize}"; }
  public String endList()   { return "\\item"; }
  public String startItem()   { return " "; }
  public String endItem()   { return "\\end{itemize}"; }
}
```

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 width;
  private double height;

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

Liskov Substitution Principle (LSP)

Un carré “est” un rectangle, on devrait pouvoir écrire :

```
public class Square extends Rectangle {

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

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

}
```

Liskov Substitution Principle (LSP)

Violation de LSP :

```
public void test(Rectangle rectangle) {
  rectangle.setWidth(2);
  rectangle.setHeight(3);
  assert rectangle.getArea()==3*2;
}
```

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 width;
  private double height;

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

Liskov Substitution Principle (LSP)

Suite de la solution :

```
class Square extends RectangularShape {
  private double sideLength;

  public void setSideLength(double sideLength) {
    this.sideLength = sideLength;
  }

  public double getWidth()  { return sideLength; }
  public double getHeight() { return sideLength; }
}
```

Liskov Substitution Principle (LSP)

Utilisation :

```
public void testRectangle(Rectangle rectangle) {
  rectangle.setWidth(2); rectangle.setHeight(3);
  assert rectangle.getArea()==3*2;
}

public void testSquare(Square square) {
  square.setSideLength(2);
  assert square.getArea()==2*2;
}
```

Interface Segregation Principle (ISP)

Principe :

Éviter les interfaces qui contiennent beaucoup de méthodes

Signification et objectifs :

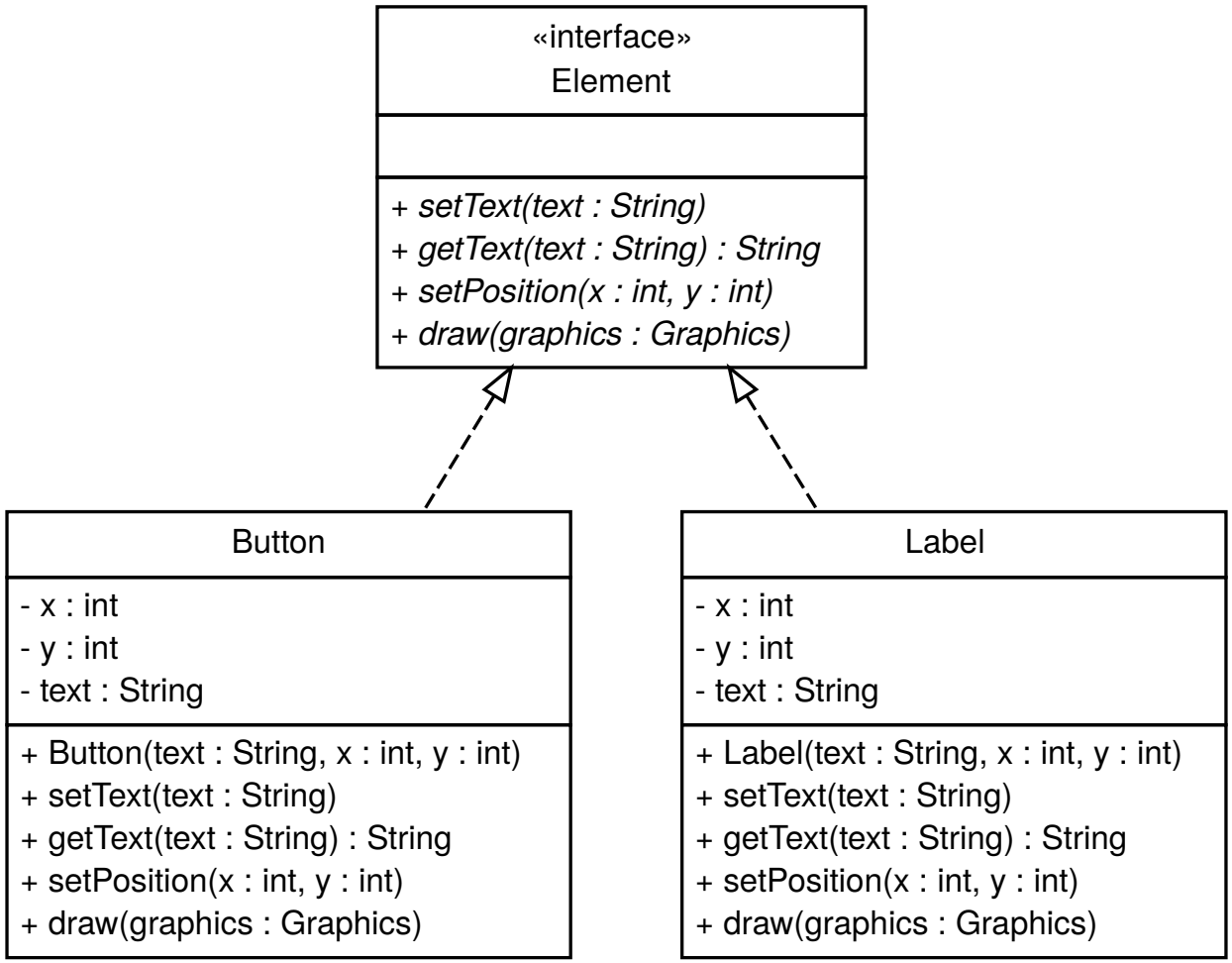
- Découper les interfaces en responsabilités distinctes (SRP)
- Quand une interface grossit, se poser la question du rôle de l'inferface
- Éviter de devoir implémenter des services qui n'ont pas à être proposés par la classe qui implémente l'interface
- Limiter les modifications lors de la modification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

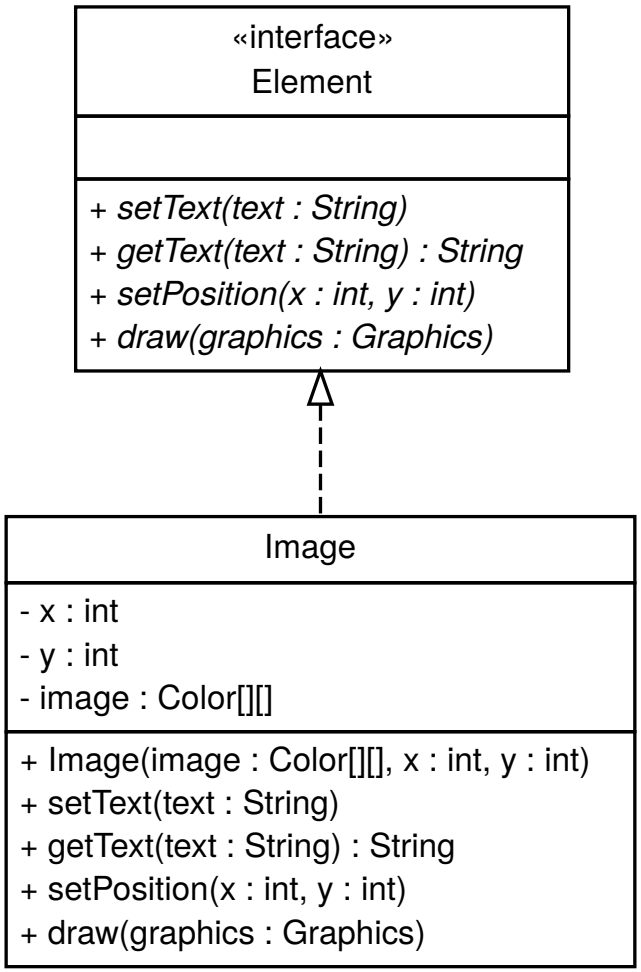
```java
public interface Element {
  public void setText(String text);
  public String getText(String text);
  public void setPosition(int x, int y);
  public void draw(Graphics graphics);
}
```

Interface Segregation Principle (ISP)

```
public class Label implements Element {
  private int x,y;
  private String text;

  public Label(String text, int x, int y) {
    this.text = text; this.x = x; this.y = y;
  }

  public void setText(String text)   { this.text = text; }
  public String getText(String text) { return text; }

  public void setPosition(int x, int y) {
    this.x = x; this.y = y;
  }

  public void draw(Graphics graphics) {
    graphics.drawString(text, x, y);
  }
}
```

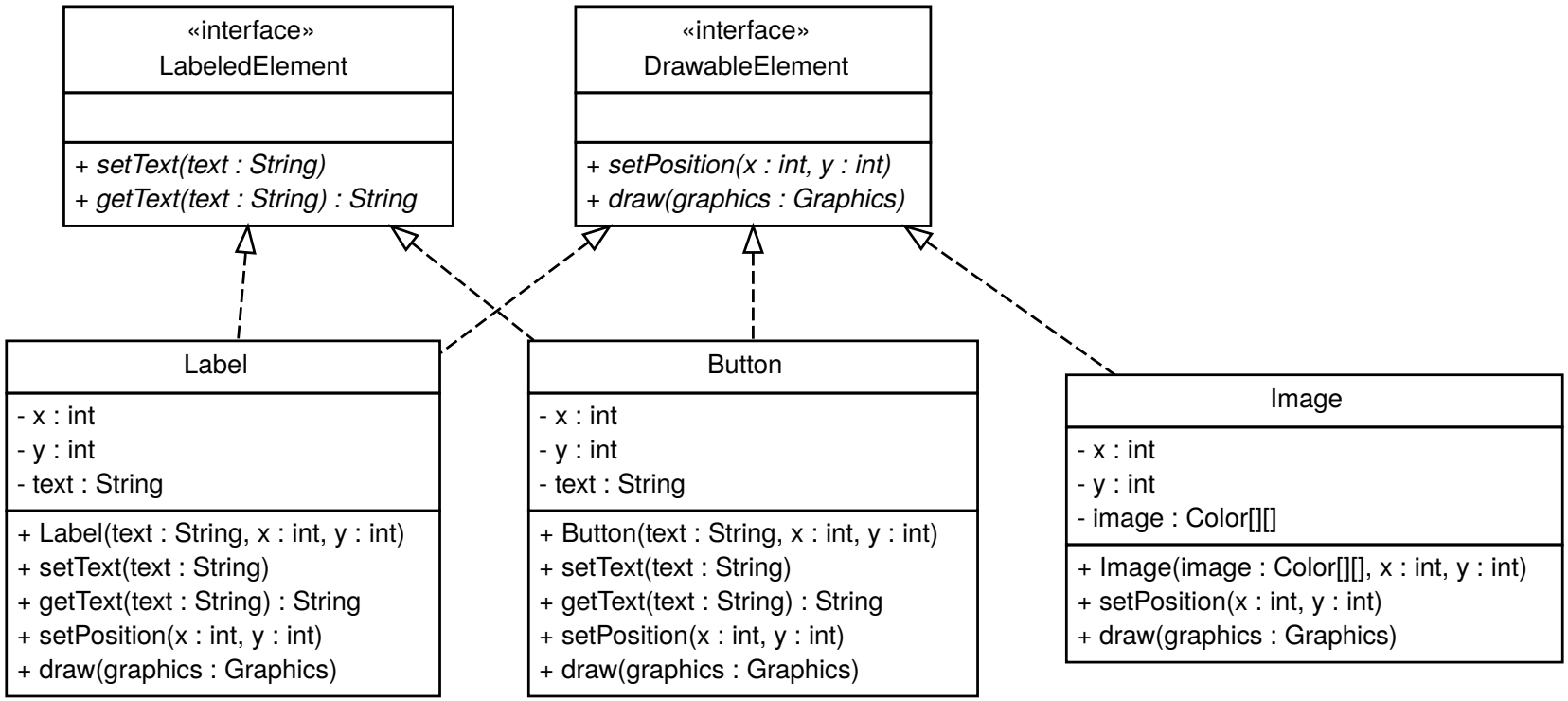
Interface Segregation Principle (ISP)

Une image n'a pas de texte : que faire dans `setText` et `getText` ?



Interface Segregation Principle (ISP)

Solution : découper l'interface.



Dependency Inversion Principle (DIP)

Principe :

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

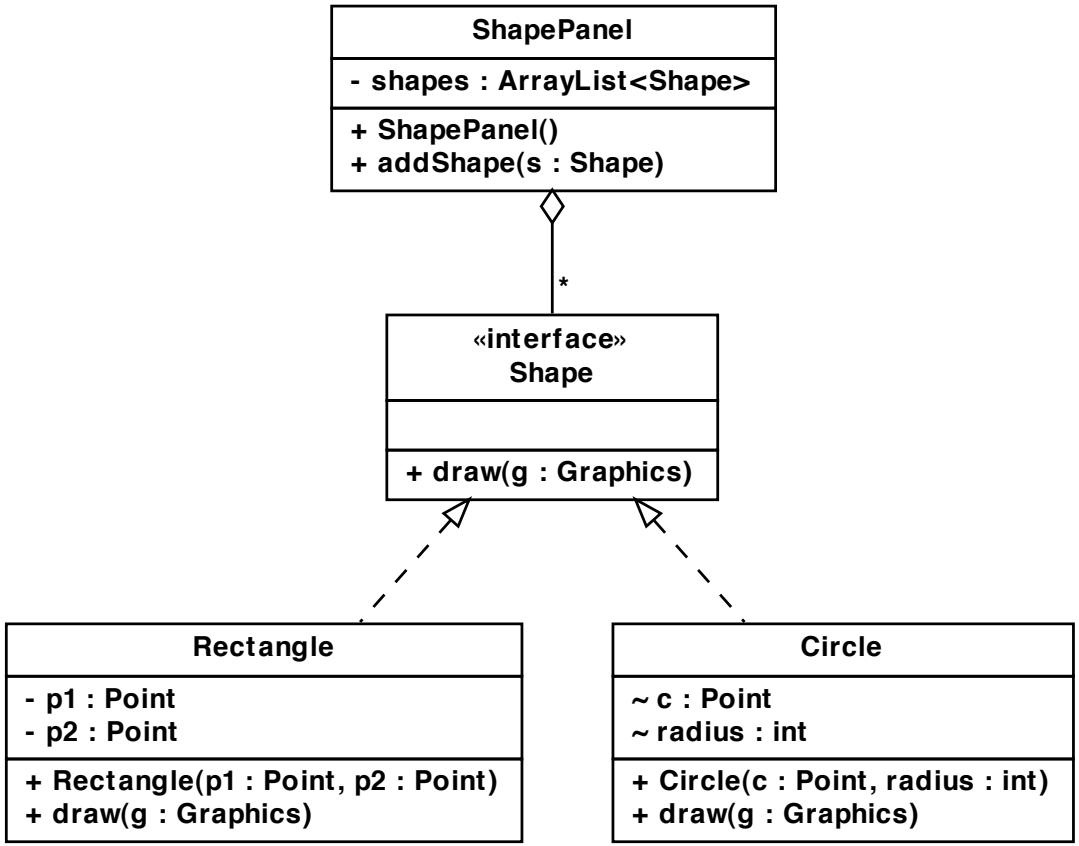
Signification et objectifs :

- 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

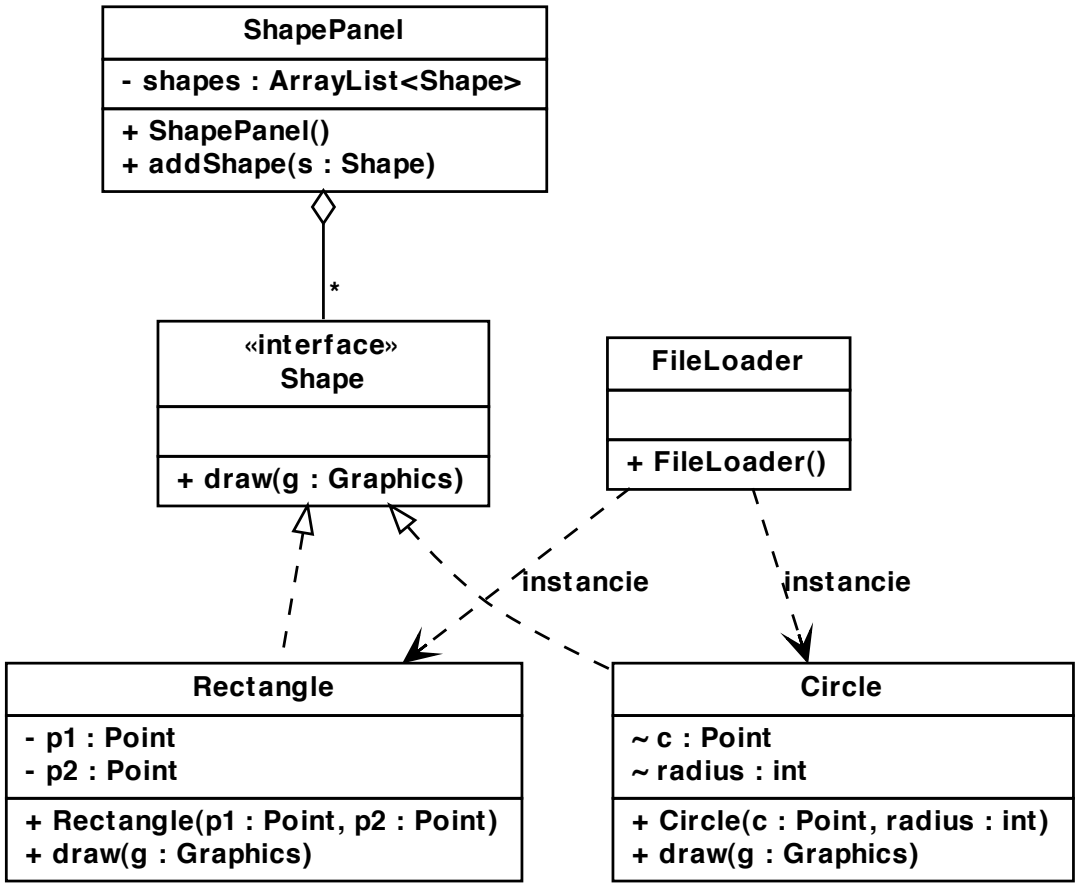
Avantages :

- 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)



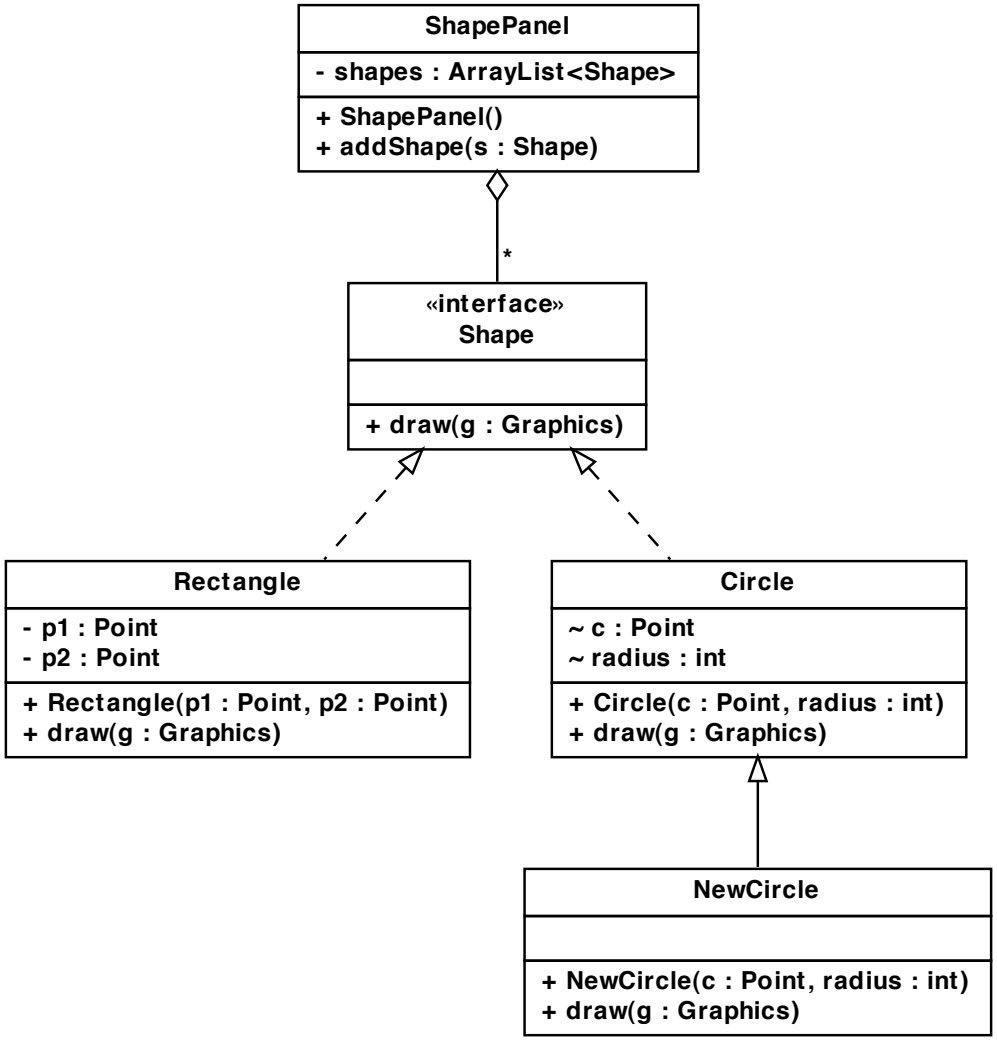
Dependency Inversion Principle (DIP)

```
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)));
  }

  /* ... */
```

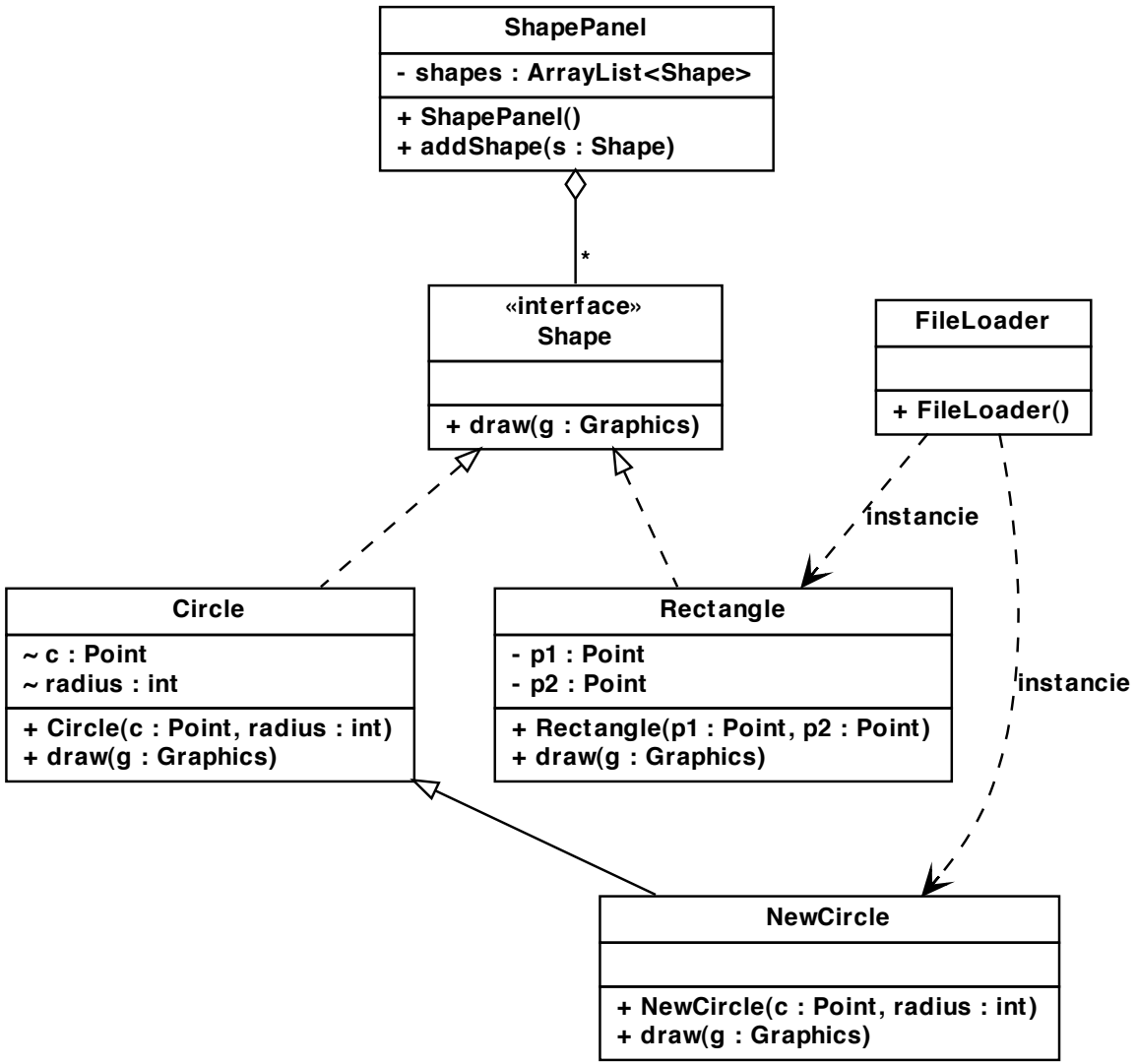
Dependency Inversion Principle (DIP)

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

Dependency Inversion Principle (DIP)



Dependency Inversion Principle (DIP)

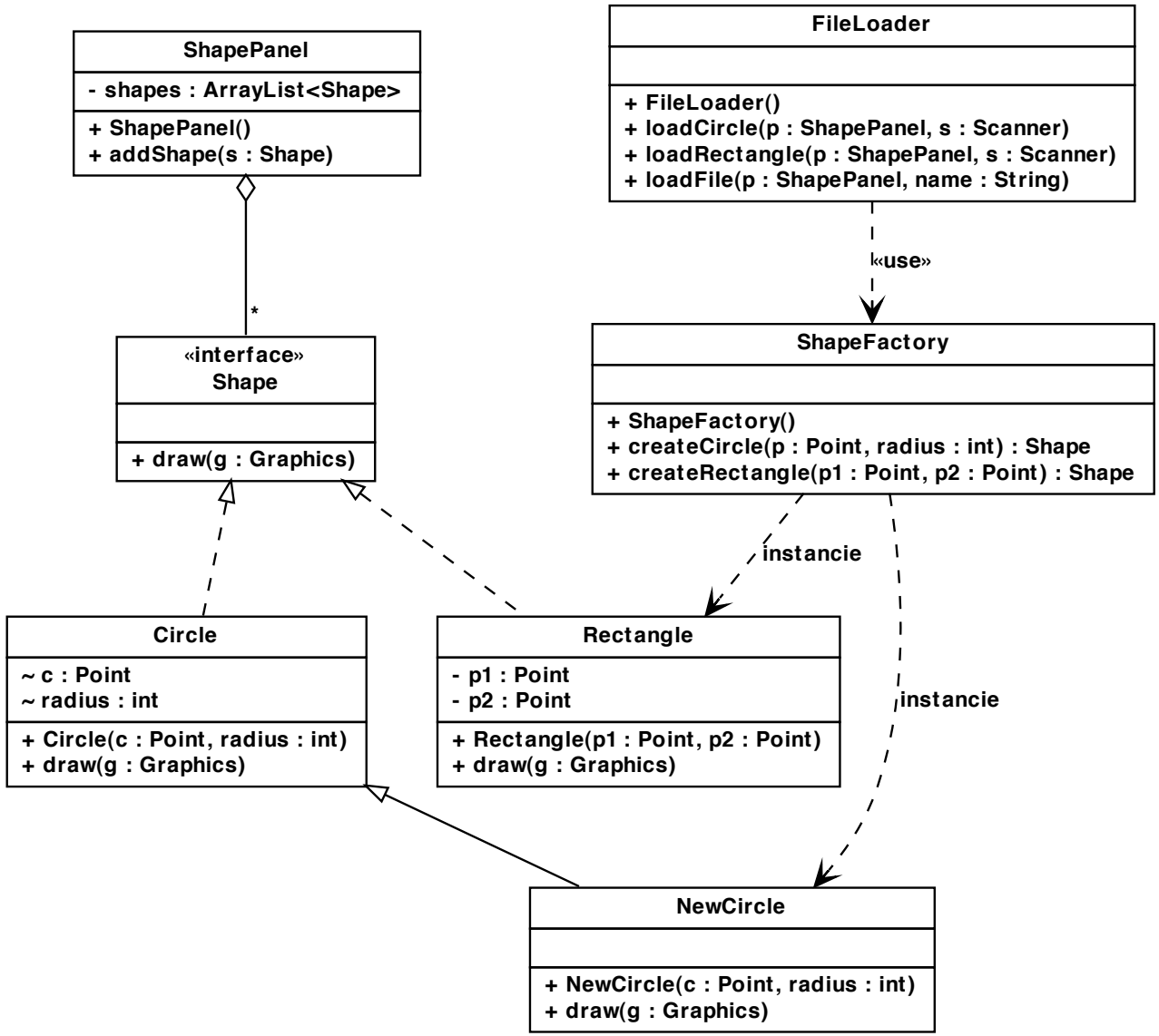


Dependency Inversion Principle (DIP)

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)));
  }

  /* ... */
```

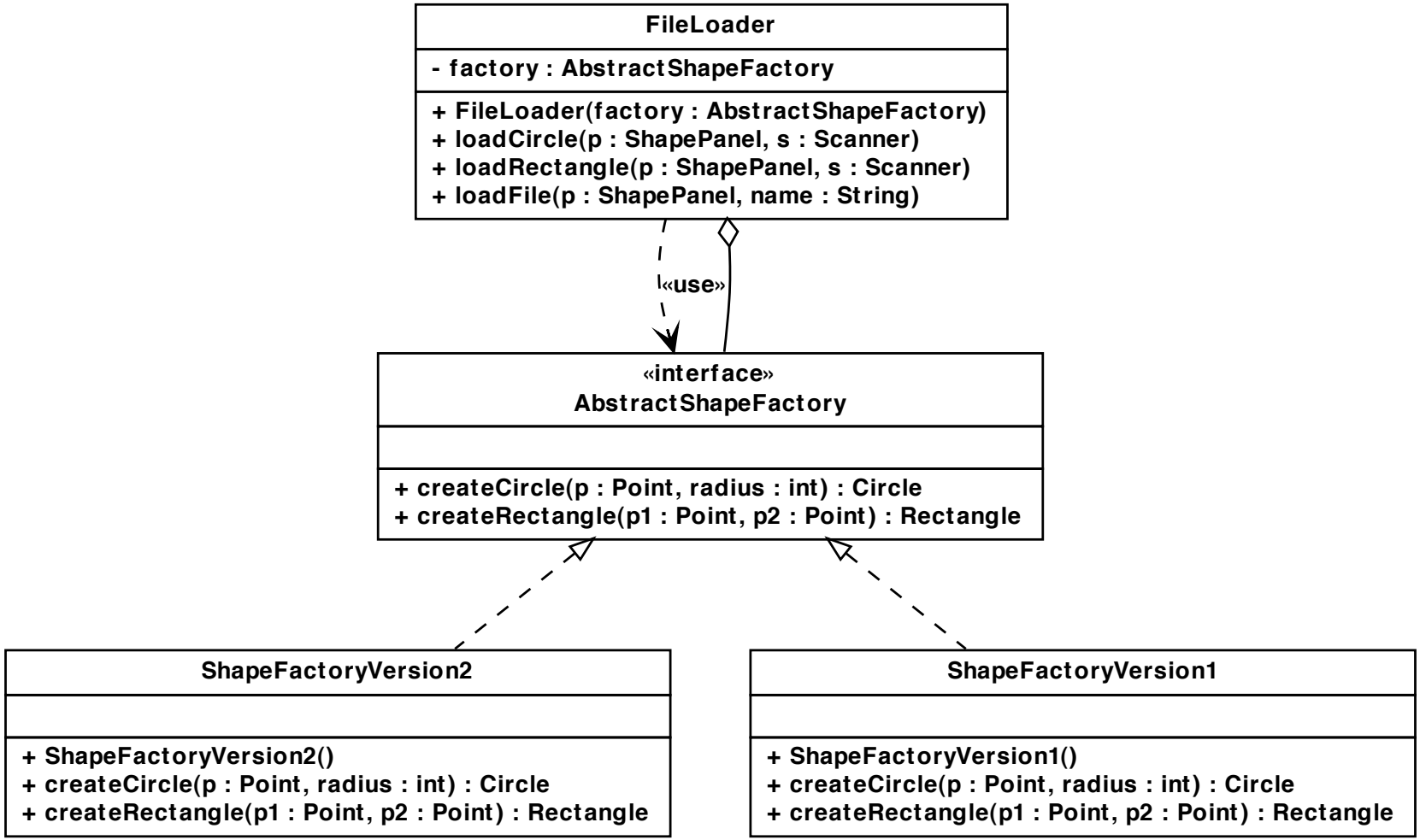
Dependency Inversion Principle (DIP)

Question : Comment faire cohabiter les deux fabriques suivantes ?

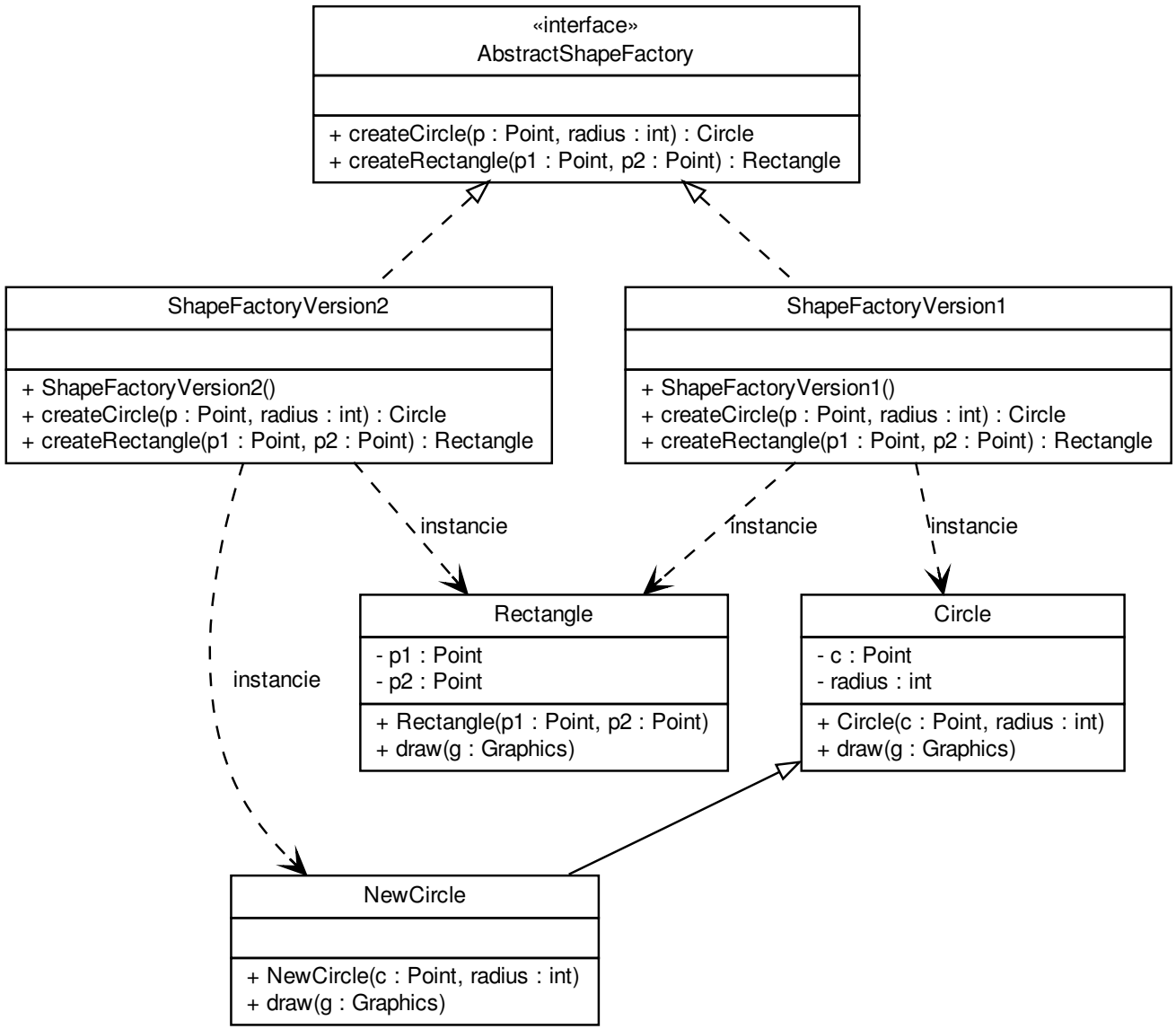
```
public class ShapeFactoryVersion1 {
  public Shape createCircle(Point p, int radius) {
    return new Circle(p, radius);
  }
  public Shape createRectangle(Point p1, Point p2) {
    return new Rectangle(p1, p2);
  }
}
```

```
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)

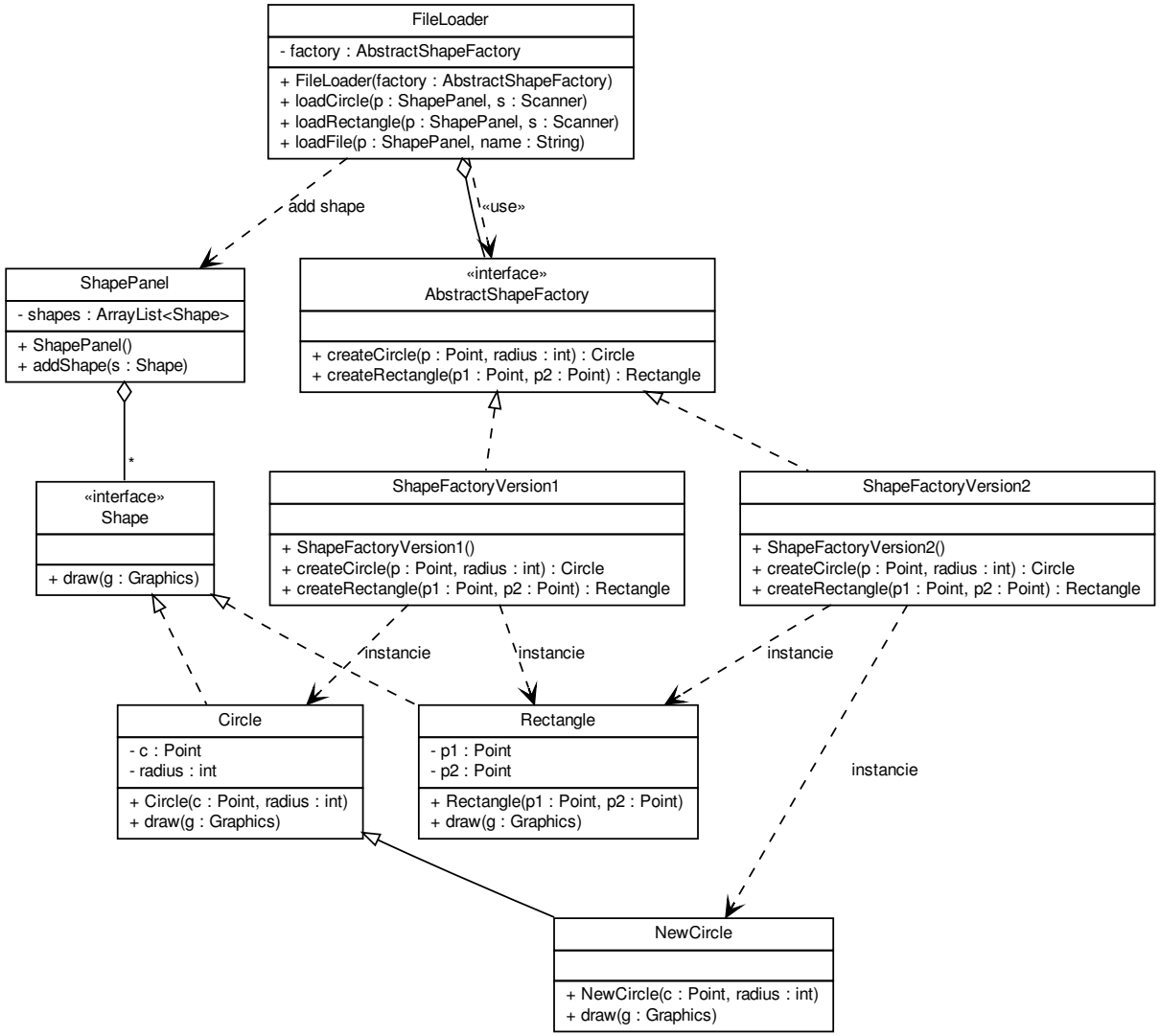


Dependency Inversion Principle (DIP)



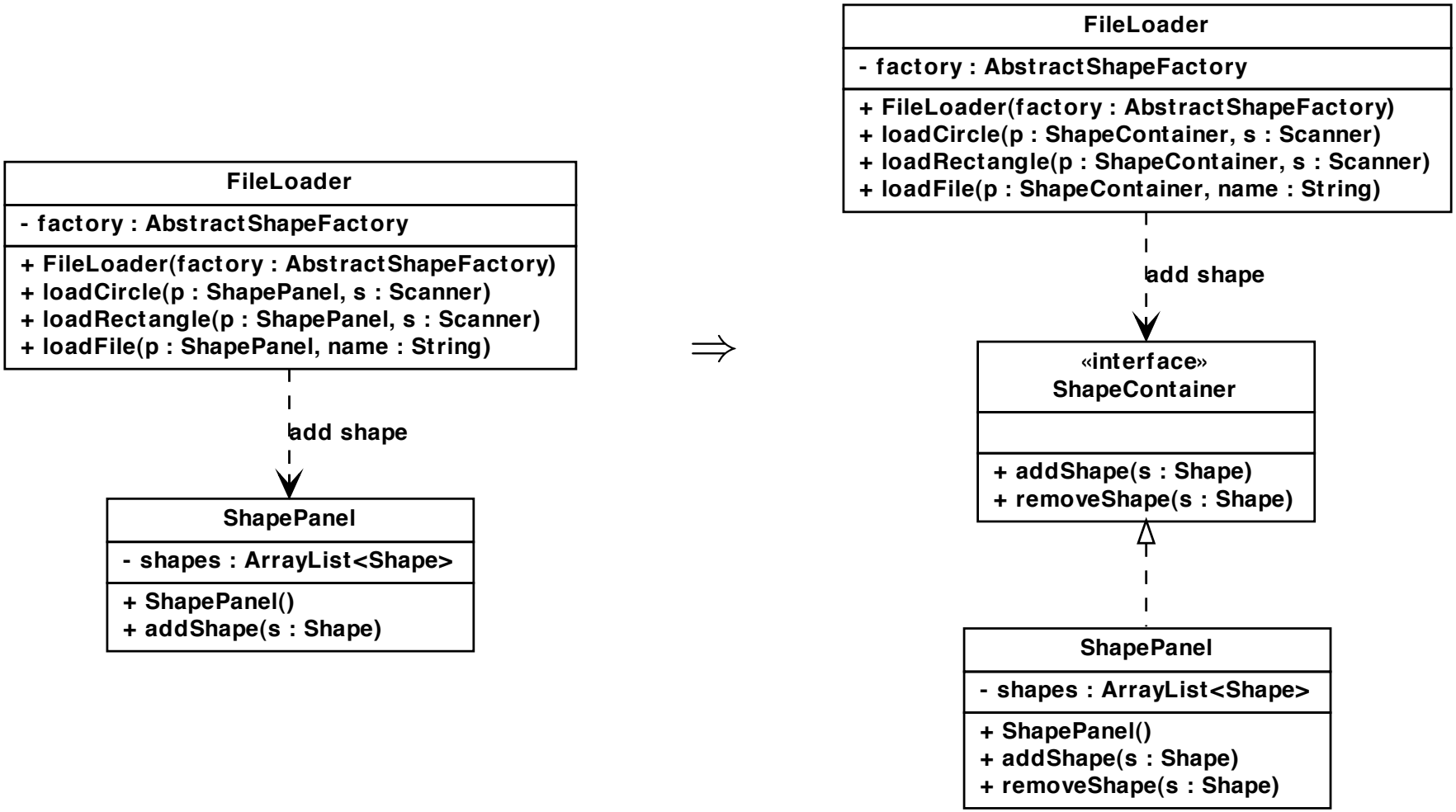
Dependency Inversion Principle (DIP)

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);
}
```

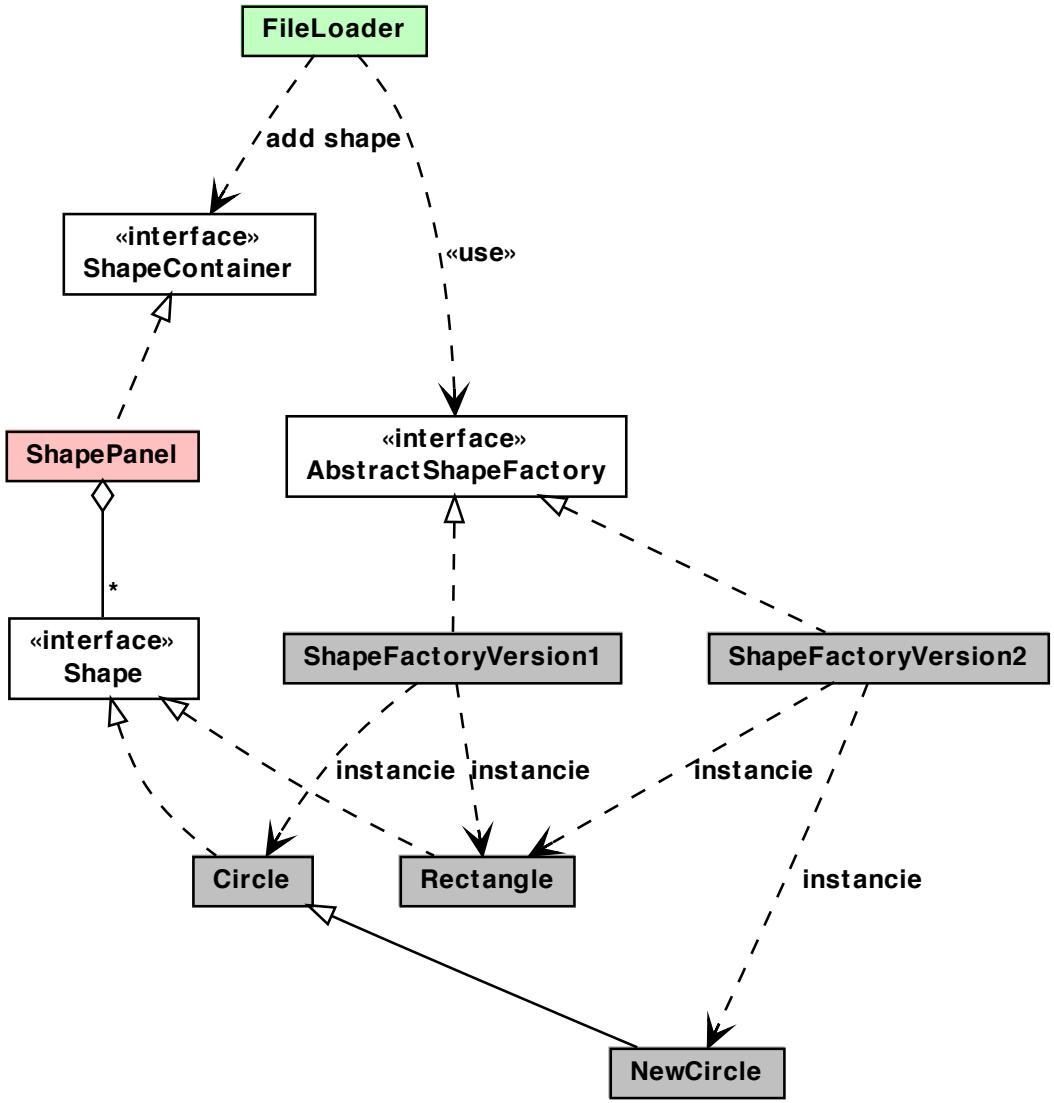
```
public class ShapePanel extends JPanel
                        implements ShapeContainer {

  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 anti-patterns) sont des erreurs courantes de conception.

Cétégories de patrons de conception

- **Création** :
  - Description de la fabrication des objets (création, initialisation)
  - Regroupement et séparation du code de fabrication des objets
- **Structure** :
  - Rendre les connexions indépendantes des évolutions futures
  - Découplage de l'application avec des interfaces
- **Comportement** :
  - Coder proprement le comportement et les interactions entre les objets

Cétégories de patrons de conception

Création

Factory Method
Abstract Factory
Builder
Prototype
Singleton

Structure

Adapter
Bridge
Composite
Decorator
Facade
Flyweight
Proxy

Comportement

Interpreter
Template Method
Chain of Responsibility
Command
Iterator
Mediator
Memento
Observer
State
Strategy
Visitor
Callback

Fabrique

Supposons que nous disposions de l'interface et la classe suivante :

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

```
public class SimpleButton implements Button  {
  public void draw() {
    System.out.println("Simple button.");
  }
}
```

La classe `SimpleButton` est instanciée dans de nombreuses autres classes.

Fabrique

Supposons que nous souhaitions ajouter un nouveau bouton sans modifier le précédent :

```
public class ModernButton implements Button  {
  public void draw() {
    System.out.println("Modern button.");
  }
}
```

Afin d'utiliser ce bouton, nous devons modifier toutes les instanciations présentes dans notre code $\Rightarrow$ violation de OCP.

Factory (Fabrique)

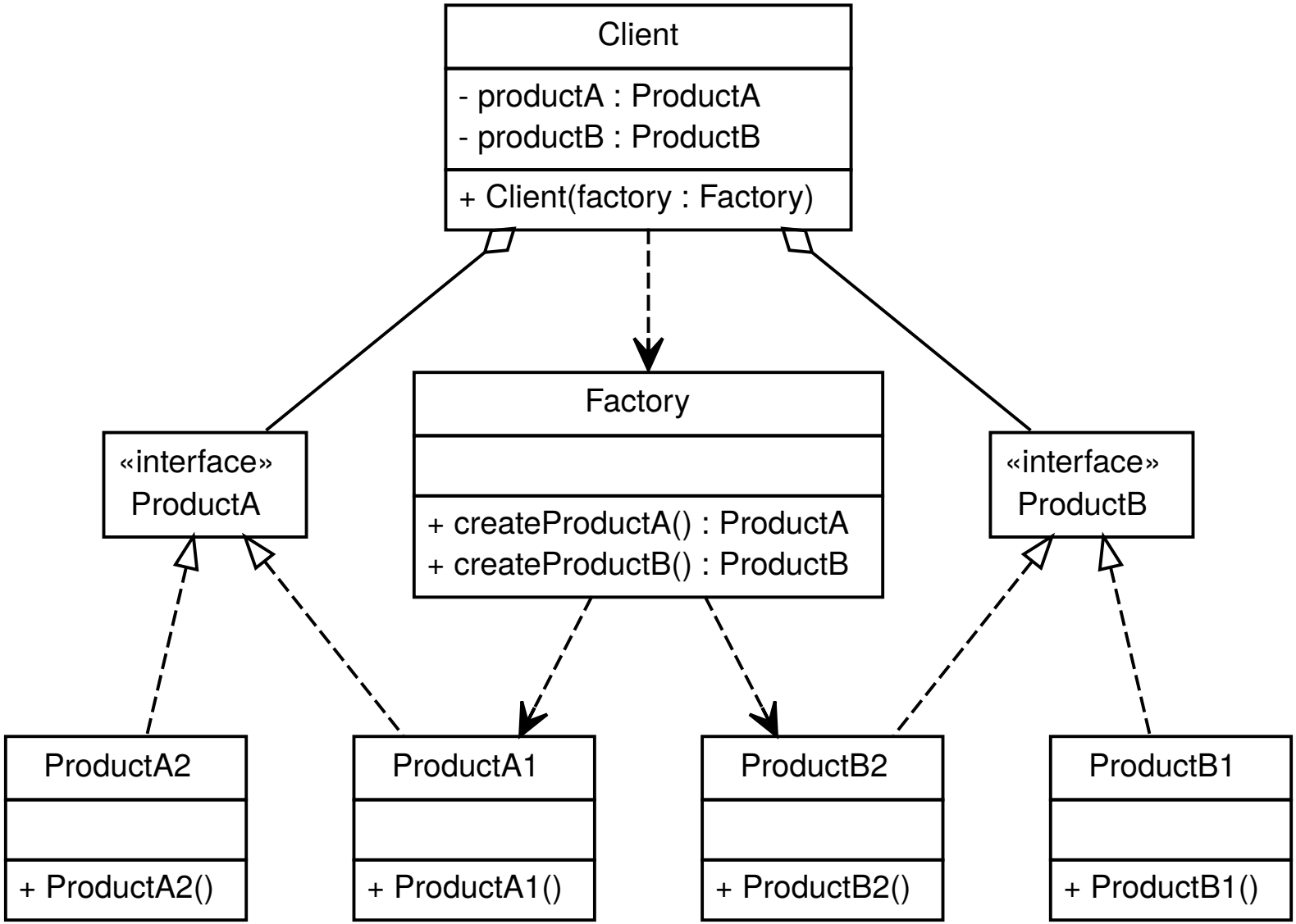
Une fabrique consiste à isoler la création des objets de leurs utilisations :

```
public class ButtonFactory {
  public Button createButton() {
    return new SimpleButton();
  }
}
```

Toutes les instanciations doivent se faire via cette classe. Les modifications nécessaires à l'utilisation de la classe `ModernButton` sont isolées :

```
public class ButtonFactory {
  public Button createButton() {
    return new ModernButton();
  }
}
```

Factory (Fabrique)



Abstract Factory (Fabrique abstraite)

Imaginons que la fabrique fournisse différents éléments :

```
public class Factory {
  public Button createButton() { return new SimpleButton(); }
  public TextBox createTextBox() { return new SimpleTextBox(); }
  public List createList() { return new SimpleList(); }
}
```

Pour passer du style simple au moderne, il nous faut changer toutes les méthodes de la fabrique, ce qui est une violation de OCP :

```
public class Factory {
  public Button createButton() { return new ModernButton(); }
  public TextBox createTextBox() { return new ModernTextBox(); }
  public List createList() { return new ModernList(); }
}
```

Abstract Factory (Fabrique abstraite)

Le code suivant semble résoudre le problème :

```
public class Factory {
  private String type;

  public Factory(String type) { this.type = type; }

  public Button createButton() {
    if (type.equals("modern")) return new ModernButton();
    else return new SimpleButton();
  }

  public TextBox createTextBox() {
    if (type.equals("modern")) return new ModernTextBox();
    else return new SimpleTextBox();
  }
  /* Idem pour createList. */
}
```

Abstract Factory (Fabrique abstraite)

Que se passe-t-il nous avons besoin d'introduire un nouveau type :

```
public class Factory {
  private String type;

  public Factory(String type) { this.type = type; }

  public Button createButton() {
    if (type.equals("modern")) return new ModernButton();
    else if (type.equals("simple")) return new SimpleButton();
    else return new OtherButton();
  }
  /* Idem pour createTextBox et createList. */
}
```

Toutes les méthodes de la fabrique doivent être modifiées, ce qui est une nouvelle violation de OCP.

Abstract Factory (Fabrique abstraite)

Les fabriques abstraites permettent de corriger ce défaut :

```
public interface AbstractFactory {
  public Button createButton();
  public TextBox createTextBox();
  public List createList();
}
```

Les classes qui devront instancier des boutons, des boites de texte ou des listes auront à leur disposition une instance d'une classe qui implémente l'interface `ButtonFactory` :

```
AbstractFactory factory = new SimpleFactory();
                        /* new ModernFactory(); */
Button button = factory.createButton();
/* ... */
```

Abstract Factory (Fabrique abstraite)

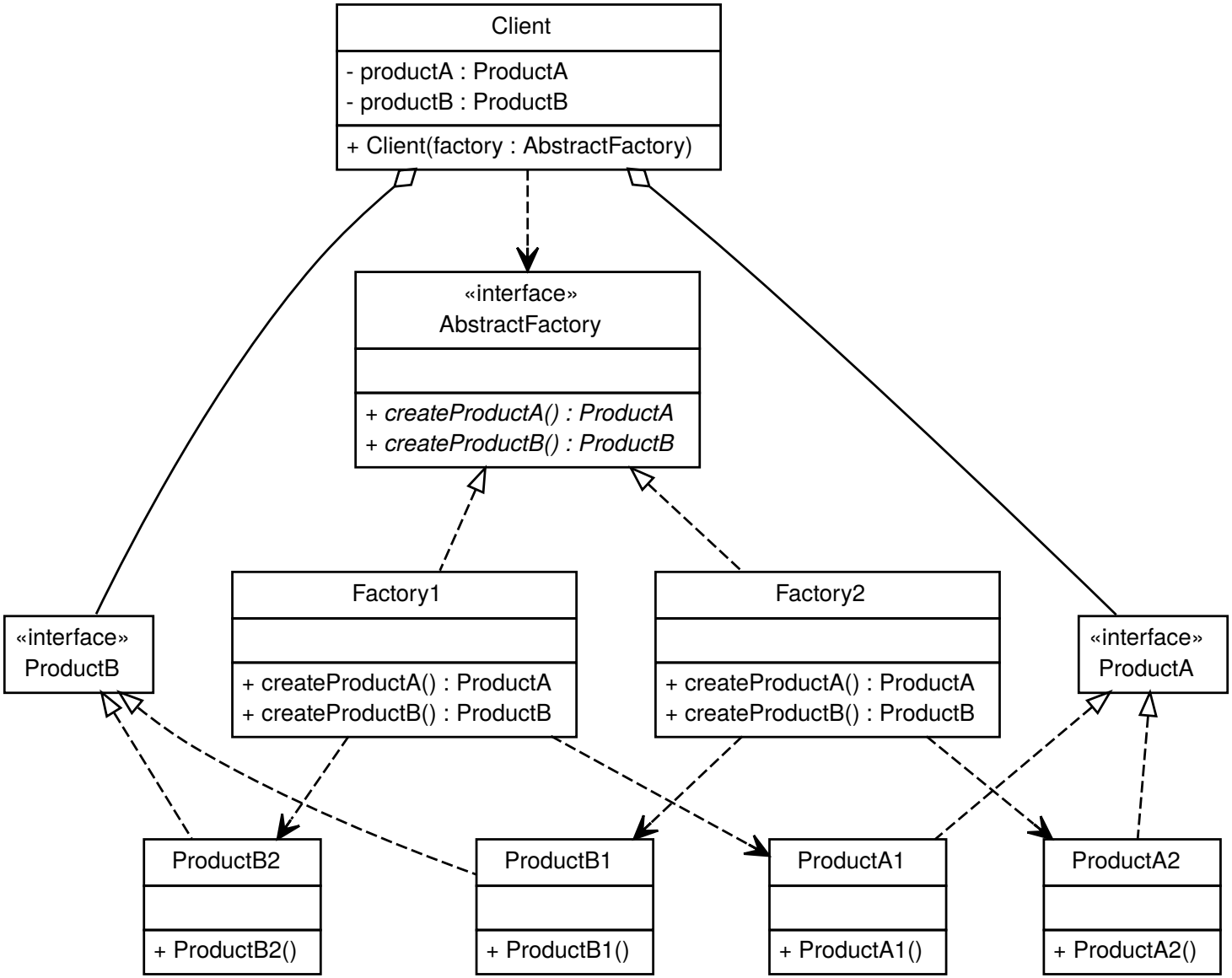
On implémente ensuite une fabrique par type :

```
public class SimpleFactory implements AbstractFactory {
  public Button createButton() { return new SimpleButton(); }
  public TextBox createTextBox() { return new SimpleTextBox(); }
  public List createList() { return new SimpleList(); }
}
```

```
public class ModernFactory implements AbstractFactory {
  public Button createButton() { return new ModernButton(); }
  public TextBox createTextBox() { return new ModernTextBox(); }
  public List createList() { return new ModernList(); }
}
```

L'ajout d'un nouveau type s'effectue par l'ajout d'une nouvelle classe, ce qui respecte OCP.

Factory (Fabrique)



Adaptateur

Supposons que la classe suivante existe :

```
public class Drawer {
  public void draw(Pencil pencil) {
    pencil.drawLine(0,0,10,10);
    pencil.drawCircle(5,5,5);
    pencil.drawLine(10,0,0,10);
  }
}
```

Nous avons également l'interface suivante :

```
public interface Pencil {
  public void drawLine(int x1, int y1, int x2, int y2);
  public void drawCircle(int x, int y, int radius);
}
```

Cette classe et cette interface ne peuvent pas être modifiées.

Adaptateur

Nous avons également la classe `Crayon` suivante :

```
public class Crayon {
  public void dessinerLigne(int x1, int y1, int x2, int y2) {
    /* ... */
  }

  public void dessinerCercle(Point center, int rayon) {
    /* ... */
  }
}
```

Nous souhaitons utiliser la classe `Crayon` comme un `Pencil` pour qu'elle puisse être utilisée par la classe `Drawer`.

Adaptateur

Pour ce faire, nous définissons l'adaptateur suivant :

```
public class CrayonAdapter implements Pencil {

  private Crayon crayon;

  public CrayonAdapter(Crayon c) { this.crayon = c; }

  public void drawLine(int x1, int y1, int x2, int y2) {
    crayon.dessinerLigne(x1, y1, x2, y2);
  }

  public void drawCircle(int x, int y, int radius) {
    crayon.dessinerCercle(new Point(x, y), radius);
  }
}
```

Adaptateur

Cette classe est utilisable de la façon suivante :

```
Pencil pencil = new CrayonAdapter(new Crayon());
Drawer drawer = new Drawer();
drawer.draw(pencil);
```

Proxy

Supposons que nous avons la classe suivante :

```
public class ArrayStack {
  private int[] stack = new int[10];
  private int size = 0;

  public void push(int value) {
    stack[size] = value;
    size++;
  }

  public int pop() { size--; return stack[size]; }
  public int size() { return size; }
  public int get(int index) { return stack[index]; }
}
```

Proxy

Faisons en sorte que la classe précédente implémente la classe suivante : s

```
public interface Stack {
  public void push(int value);
  public int pop();
  public int size();
  public int get(int index);
}
```

Nous souhaitons pouvoir rendre une pile non modifiable :

```
Stack stack = new ArrayStack(20);
stack.push(2);
stack.push(4);
stack = new UnmodifiableStack(stack);
/* stack.push et stack.pop interdit (à l'exécution). */
```

Proxy

Pour ce faire, nous définissons un proxy :

```
public class UnmodifiableStack implements Stack {
  private Stack stack;

  public UnmodifiableStack(Stack stack) { this.stack = stack; }

  public void push(int value) {
    throw new UnsupportedOperationException();
  }

  public int pop() { throw new UnsupportedOperationException(); }

  public int size() { return stack.size(); }
  public int get(int index) { return stack.get(i); }
}
```

Décorateur

Supposons que nous avons la classe suivante :

```
public class ArrayStack {
  private int[] stack = new int[10];
  private int size = 0;

  public void push(int value) {
    list[size] = value;
    size++;
  }

  public int pop() { size--; return list[size]; }
}
```

Décorateur

Nous souhaitons ajouter des logs pour débugger notre programme :

```
public class ArrayStack {
  private int[] stack = new int[10];
  private int size = 0;

  public void push(int value) {
    System.out.println("push("+value+")");
    list[size] = value; size++;
  }

  public int pop() {
     System.out.println("pop()");
     size--; return list[size];
  }
}
```

Décorateur

Cette modification a été réalisée en modifiant une classe existante. De plus, une nouvelle modification est nécessaire pour retirer les logs.

Définissons l'interface suivante :

```
public interface Stack {
  public void push(int value);
  public int pop();
}
```

Faisons en sorte que `ArrayStack` implémente cette interface :

```
public class ArrayStack implements Stack {
  /* ... */
}
```

Décorateur

Il suffit alors de définir un décorateur :

```
public class VerboseStack implements Stack {
  private Stack stack;

  public VerboseStack(Stack stack) { this.stack = stack; }

  public void push(int value) {
    System.out.println("push("+value+")");
    stack.push(value);
  }

  public int pop() {
     System.out.println("pop()");
     return stack.pop();
  }
}
```

Décorateur

Supposons que nous ayons le code suivant :

```
Stack stack = new ArrayStack(10);
stack.push(2);
stack.pop();
```

Il est très facile d'introduire le décorateur :

```
Stack stack = new ArrayStack(10);
stack = new VerboseStack(stack);
stack.push(2);
stack.pop();
```

Ce code produit la sortie suivante :

```
push(2);
pop();
```

Décorateur

Nous définissons un nouveau décorateur :

```
public class CounterStack implements Stack {
  private Stack stack;
  private int size;

  public VerboseStack(Stack stack) {
    this.stack = stack; size = 0;
  }

  public void push(int value) { size++; stack.push(value); }
  public int pop()            { size--; return stack.pop(); }
  public int getSize()        { return size; }
}
```

Décorateur

Un exemple d'utilisateur du décorateur précédent :

```
Stack stack = new ArrayStack(10);
CounterStack counterStack = new CounterStack(stack);
stack = counterStack;
stack.push(2); stack.push(3);
stack.pop();
System.out.println(counterStack.getSize());
```

Ce code produit la sortie suivante :

```
1
```

Décorateur

Il est possible d'utiliser plusieurs décorateurs simultanément :

```
Stack stack = new ArrayStack(10);
stack = new VerboseStack(stack);
CounterStack counterStack = new CounterStack(stack);
stack = counterStack;
stack.push(2); stack.push(3);
stack.pop();
System.out.println(counterStack.getSize());
```

Ce code produit la sortie suivante :

```
push(2);
push(3);
pop();
1
```

Patron de méthode

Le patron de méthode permet de faire partager un comportement par plusieures classes :

```
public abstract class List {
  private int[] list = new int[10]; private int size = 0;

  public void add(int value) { list[size] = value; size++; }

  public int eval() {
    int result = neutral(); // util. d'une méthode abstraite
    for (int i = 0; i < list.length; i++)
      result = compute(result, list[i]); // idem
    return result;
  }

  protected abstract int neutral(); // méthode abstraite
  protected abstract int compute(int a, int b); // idem
}
```

Patron de méthode

La classe abstraite est alors étendue de la façon suivante :

```
public class ListSum extends List {
  public int neutral() { return 0; }
  public int compute(int a, int b) { return a+b; }
}
```

```
public class ListProduct extends List {
  public int neutral() { return 1; }
  public int compute(int a, int b) { return a*b; }
}
```

Patron de méthode

Dans l'exemple précédent :

- la classe `List` est abstraite ;
- la méthode `eval` est une méthode socle qui utilise `neutral` et `compute` pour définir son comportement ;
- les méthodes `neutral` et `compute` sont abstraites ;
- elles sont définies dans les classes qui étendent `List`.

Stratégie

Il est également possible d'utiliser une stratégie :

```
public interface Strategy {
  public int neutral();
  public int compute(int a, int b);
}
```

```
public class SumStrategy implements Strategy {
  public int neutral() { return 0; }
  public int compute(int a, int b) { return a+b; }
}
```

```
public class ProductStrategy implements Strategy {
  public int neutral() { return 1; }
  public int compute(int a, int b) { return a*b; }
}
```

Stratégie

On délègue à la stratégie une partie du traitement :

```
public class List {
  /* Propriétés et méthode add */

private Stategy strategy;

  public List(Stategy strategy) {
   this.strategy = strategy;
  }

  public int eval() {
    int result = strategy.neutral();
    for (int i = 0; i < list.length; i++)
      result = strategy.compute(result, list[i]);
    return result;
  }
}
```

Stratégie

Utilisation des stratégies :

```
List listSum = new List(new SumStrategy());
listSum.add(2); listSum.add(3);
System.out.println(listSum.eval());
List listProduct = new List(new ProductStrategy());
listProduct.add(2); listProduct.add(3);
System.out.println(listProduct.eval());
```

État

Considérons la classe suivante :

```
public class Writer {
  private int state = 0;

  public void write(char character) {
    switch (state) {
    case 0 :
      if (character=='{') state = 1;
      else System.out.print(character);
      break;
    case 1:
      if (character=='}') state = 0;
      else System.out.print(Character.toUpperCase(character));
      break;
    }
  }
}
```

État

Un exemple d'utilisation de la classe précédente :

```
Writer writer = new Writer();
String string = "abc{def}ghi";
for (int index = 0; index < string.length(); index++)
  writer.write(string.charAt(index));
```

Le code précédent génère la sortie suivante :

```
abcDEFghi
```

Notez que si nous souhaitons ajouter de nouveaux états, nous devons modifier les méthodes de la classe `Writer` $\Rightarrow$ violation de OCP.

De plus, le fait qu'une classe implémente un grand nombre d'états peut être considéré comme une violation de SRP.

État

Pour corriger ces défauts, nous déclarons la classe et l'interface suivantes :

```
public class WriterContext {
  private WriterState state = new WriterState0();

  public void write(char character) {
    state.write(this, character);
  }

  public void changeState(WriterState state) {
    this.state = state;
  }
}
```

```
public interface WriterState {
  public void write(WriterContext context, char character);
}
```

État

Nous pouvons maintenant définir les différents états :

```
public class WriterState0 implements WriterState {
  public void write(WriterContext context, char c) {
    if (c=='{') {context.changeState(new WriterState1());}
    else System.out.print(c);
  }
}
```

```
public class WriterState1 implements WriterState {
  public void write(WriterContext context, char c) {
    if (c=='}') {context.changeState(new WriterState0());}
    else System.out.print(Character.toUpperCase(c));
  }
}
```

État

Un exemple d'utilisation de la classe précédente :

```
WriterContext writer = new WriterContext();
String string = "abc{def}ghi";
for (int index = 0; index < string.length(); index++)
  writer.write(string.charAt(index));
```

Le code précédent génère la sortie suivante :

```
abcDEFghi
```

Notez que l'ajout d'un nouvel état s'effectue en ajoutant une nouvelle classe qui implémente l'interface `WriterState`.

Itérateur

Considérons l'interface suivante :

```
public interface List {
  public void add(int value);
}
```

Considérons l'implémentation suivante de cette interface :

```
public class ArrayList implements List {
  private int[] list = new int[10];
  private int size = 0;

  public void add(int value) {list[size] = value;size++;}
  public int size() { return size; }
  public int get(int index) { return list[index]; }
}
```

Itérateur

Supposons que nous avons également cette implémentation :

```
public class Node {
  private int value; private Node next;
  public Node(int value, Node next) {
    this.value = value; this.next = next;
  }
  public getValue() { return value; }
  public getNext() { return next; }
}
```

```
public class LinkedList implements List {
  private Node first = null;
  public void add(int value) { first = new Node(value, first); }
  public Node getFirst() { return first; }
}
```

Itérateur

Nous voulons itérer de la même façon sur les deux implémentations :

```
List list = new ArrayList(); /* ou LinkedList() */
list.add(2); list.add(4);
Iterable iterable = list;
Iterator iterator iterator = iterable.iterator();
while (iterator.hasNext()) {
  int value = iterator.next();
  System.out.println(value);
}
```

Nous introduisons l'interface `Iterable` pour permettre que des objets autres que des listes puissent être parcourus en utilisant un itérateur.

$\Rightarrow$ Nous voulons séparer la notion de liste et d'objet "itérable".

Itérateur

Nous en déduisons les deux interfaces suivantes :

```
public interface Iterator {
  public boolean hasNext();
  public int next();
}
```

```
public interface Iterable {
  public Iterator iterator();
}
```

De plus, l'interface `List` doit étendre l'interface `Iterable` :

```
public interface List { /* ... */ } extends Iterable;
```

Itérateur

L'implémentation de l'itérateur pour les `ArrayList` :

```
public class ArrayListIterator implements Iterator {
  private ArrayList list;
  private int position;

  public ArrayListIterator(ArrayList list) {
    this.list = list; position = 0;
  }

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

  public int next() {
    int value = list.get(position);
    position++;
    return value;
  }
}
```

Itérateur

L'implémentation de l'itérateur pour les `LinkedList` :

```
public class LinkedListIterator implements Iterator {
  private Node next;

  public LinkedListIterator(LinkedList list) {
    next = list.getFirst();
  }

  public boolean hasNext() { return next!=null; }

  public int next() {
    int value = next.getValue();
    next = next.getNext();
    return value;
  }
}
```

Itérateur

Nous devons également implémenter la méthode `iterator` dans les deux classes `ArrayList` et `LinkedList` :

```
public class ArrayList implements List {
  /* ... */
  public Iterator iterator() {
    return new ArrayListIterator(this);
  }
}
```

```
public class LinkedList implements List {
  /* ... */
  public Iterator iterator() {
    return new LinkedListIterator(this);
  }
}
```

Fonction de rappel (Callback) – C/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

```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 MyClass {
  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#

```
public delegate void Callback(int c);

class MyClass {

  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);
  }
}
```