Fiche 1 : syntaxe de Java et programmation impérative

IPO – Introduction à la programmation objet

Thibaut Balabonski @ Université Paris-Saclay
http://www.lri.fr/~blsk/IPO/

V1, automne 2022

Structure minimale. Un *programme* Java est un texte, écrit dans un fichier d'extension `.java`. Voici la structure minimale pour un programme `hello`, à écrire dans un fichier nommé `hello.java`, où les points de suspension doivent être remplacés par le code à exécuter.

```
public class hello {
  public static void main(String[] args) {
    ...
  }
}
```

On verra plus tard ce que signifient tous ces mots, vous pouvez les considérer comme une formule magique en attendant. Note : le nom du programme, écrit après **`public class`**, doit correspondre exactement au nom du fichier, majuscules comprises.

On applique le *compilateur* Java à un tel programme avec la commande

```
javac hello.java
```

(à faire dans un terminal). Ceci produit un nouveau fichier `hello.class`, qui est une traduction du programme dans le format de la *machine virtuelle* Java (JVM). On peut ensuite exécuter le programme avec la commande

```
java hello
```

(toujours dans le terminal). Pour que notre programme `hello` affiche `Hello, world!` lors de son exécution, il suffit de placer à la place des points de suspension la ligne

```
    System.out.println("Hello, world!");
```

Types, données et opérations. En java, on manipule des données qui peuvent être :

- des *valeurs de base*, comme des nombres entiers (**`int`**), des nombres décimaux (**`double`**), des booléens (**`boolean`**), des caractères (**`char`**) ; et
- des *valeurs complexes*, comme des chaînes de caractères (`String`), des tableaux (**`int`**`[]`), ou d'autres objets à définir soi-même.

Chaque type de données est associé à ses opérations. En voici quelques unes.

- Pour les entiers (**`int`**) : opérations arithmétiques `+`, `-`, `*`, `/`, `%`, comparaisons `==`, `!=`, `<`, `<=`, `>`, `>=`.
- Pour les booléens : conjonction `&&` (« et »), disjonction `||` (« ou »), négation `!` (« non »).
- Pour les chaînes de caractères : l'opérateur `+` est la concaténation, c'est-à-dire la mise bout à bout de deux chaînes.

Variables. Une variable est un *nom*, faisant référence à une *valeur stockée en mémoire*. On déclare une variable en donnant un nom et le type de la valeur à stocker, et éventuellement une valeur initiale. On peut déclarer plusieurs variables d'un même type sur une seule ligne en les séparant par des virgules.

```
int x = 3;
String s1, s2;
```

On peut consulter la valeur d'une variable en utilisant simplement son nom, ou la modifier à l'aide de l'opérateur `=` d'*affectation*.

```
y = x + 1;
```

Attention à ne pas confondre :

- l'opérateur `==` qui teste l'égalité de deux valeurs (et produit un booléen),
- l'opérateur `=` qui modifie la valeur d'une variable (et ne produit rien).

Structures de contrôle. En programmation impérative, le corps d'un programme est une séquence d'instructions, à exécuter dans l'ordre. Les *structures de contrôle* permettent d'organiser la séquence.

Le *branchement conditionnel* sélectionne un bloc (c'est-à-dire une séquence d'instructions) à exécuter, en fonction du résultat d'un test. Le deuxième bloc, introduit par **else**, est facultatif.

```
if (score >= 100) {
  System.out.println("Gagné");
} else {
  System.out.println("Perdu");
}
```

La *boucle conditionnelle* (« **while** ») répète l'exécution d'un bloc tant qu'une condition est vérifiée.

```
int x = 1;
while (x < n) {
  x = 2*x;
}
System.out.println(x);
```

La *boucle inconditionnelle* (« **for** ») répète l'exécution d'un bloc pour chaque valeur d'un intervalle entier.

```
for (int i = 0; i < n; i++) {
  System.out.println(i * i);
}
```

Fonctions. Les fonctions permettent d'*isoler* un fragment de programme en lui donnant un nom, que l'on s'efforce de choisir suffisamment explicite pour aider la compréhension du programme. La définition d'une fonction contient, dans l'ordre :

1. le type de la valeur qui sera renvoyée à la fin de l'exécution de la fonction (ou **void** si aucune valeur renvoyée),
2. le nom de la fonction,
3. la liste des arguments attentus, chacun donné par son nom et son type,
4. le bloc d'instructions à exécuter.

Les trois premiers éléments forment la *déclaration* de la fonction, le quatrième élément est son *corps*. En java, on ajoutera pour l'instant les deux mots-clés **public static** devant la déclaration de chaque fonction (dans le contexte de java, ceci ne s'appelle pas une « fonction » mais une « méthode », nous y reviendrons plus tard).

```
public static int binaryLog(int n) {
  int x = 1;
  while (x < n) {
    x = 2*x;
  }
  return x;
}
```

On *appelle* une fonction en fournissant des valeurs concrètes pour chacun des arguments.

```
int a = binaryLog(1024);
```

Note : un argument entier est passé *par valeur*.

Tableaux. Un tableau permet de stocker un ensemble de données de même type, et d'accéder à chacune par son *indice*. La première donnée a l'indice 0. On note avec des crochets `[]` le type d'un tableau.

```
int[] tab;
```

On crée un tableau avec le mot-clé **new**. Il faut alors préciser le nombre d'éléments que contiendra le tableau.

```
tab = new int[12];
```

On accède à un élément d'un tableau, en lecture comme en écriture, en précisant son indice.

```
tab[2] = tab[0] + tab[1];
```

Les tableaux (et d'autres structures de données similaires) dispose d'un mécanisme d'*itération* (« **for** each »), consistant à énumérer tous les éléments et répéter un même bloc d'instruction pour chacun.

```
  for(int x: tab) { // tab contient des entiers
    System.out.println(x*x);
  }
```

En java, passer un tableau en argument à une fonction se fait uniquement (et implicitement) *par référence*. Une fonction s'appliquant à un tableau est donc susceptible de le modifier définitivement.

Décodage des arguments de la ligne de commande. La fonction principale `main` prend obligatoirement en argument un tableau de chaînes de caractères (type `String[]`), contenant les arguments donnés au programme sur la ligne de commande. Pour « décoder » ces arguments lorsqu'ils sont censés représenter des nombres, on utilise des fonctions notées `Interger.parseInt` ou `Double.parseDouble`.

```
  public static void main(String[] args) {
    int x = Integer.parseInt(args[0]);
    double d = Double.parseDouble(args[1]);
    System.out.println(x + d);
  }
```

Pour passer des arguments sur la ligne de commande, on les place après le nom du programme.

```
  java hello 42 1.0001
```

Un programme. Ce programme doit être donné dans un fichier `mandelbrot.java`.

```
public class mandelbrot {
    public static boolean escapeTime(double cx, double cy) {
        double zx = 0;
        double zy = 0;
        for (int i = 0; i < 1000; i++) {
            double z = zx * zx - zy * zy;
            zy = 2 * zx * zy + cy;
            zx = z + cx;
            if (zx*zx + zy*zy > 4) { return false; }
        }
        return true;
    }

    public static void printMandelbrot(double x0, double y0, double radius, int res) {
        double xMin = x0 - radius;
        double yMin = y0 - radius;
        double pixelSize = radius / res;
        for (int y = 0; y < 2*res+1; y++) {
            for (int x = 0; x < 2*res+1; x++) {
                if (escapeTime(xMin + x*pixelSize, yMin + y*pixelSize)) {
                    System.out.print("@@");
                } else {
                    System.out.print("..");
                }
            }
            System.out.println();
        }
    }

    public static void main(String[] args) {
        double x0 = Double.parseDouble(args[0]);
        double y0 = Double.parseDouble(args[1]);
        double radius = Double.parseDouble(args[2]);
        int res = Integer.parseInt(args[3]);
        printMandelbrot(x0, y0, radius, res);
    }
}
```

On peut l'exécuter par exemple avec

```
  java mandelbrot -.5 0 1 20
```

Fiche 2 : création d'objets et bibliothèque standard

IPO – Introduction à la programmation objet

Thibaut Balabonski @ Université Paris-Saclay
http://www.lri.fr/~blsk/IPO/

V1, automne 2022

Classes et objets. Au-delà des types de données de base comme **`int`**, **`double`** ou **`boolean`** et des tableaux primitifs (`String[]`, **`int`**`[][]`...), Java permet la définition et la manipulation d'objets plus complexes. C'est là que va apparaître la *programmation objet*.

On distingue :

— les *classes* : des types pouvant être définis par l'utilisateur ou par la bibliothèque standard,
— les *objets* : les éléments des classes.

Un objet `o` d'une classe `C` est également appelé une *instance* de la classe `C`.

Les mécanismes de la programmation objet sont liés à un principe d'*encapsulation* : l'utilisateur d'un objet n'a pas besoin de connaître la structure exacte de l'objet manipulé ou les détails internes de son fonctionnement, et n'agit normalement pas directement sur l'objet lui-même comme on le ferait sur un tableau ou une structure de C/C++. À la place, l'utilisateur envoie des « messages » à un objet, pour lui demander de réaliser certaines opérations, les messages possibles étant définis par la classe à laquelle appartient l'objet.

Une classe définit donc :

— en premier lieu des opérations, appelées *méthodes*, permettant d'utiliser les objets et représentant les « messages » qui peuvent être envoyés aux objets de cette classe,
— et accessoirement une structure de données, représentant l'état interne, ou « mémoire », d'un objet.

Si on considère une classe `C` et deux objets `o1` et `o2` de cette classe : on manipule `o1` et `o2` avec les mêmes méthodes, définies par la classe `C`. En revanche, `o1` et `o2` peuvent avoir des états internes différents, reflétant le fait que ces objets ont été créés et utilisés indépendamment l'un de l'autre (chacun à sa vie propre !).

Création et utilisation d'un objet. Le nom d'une classe est également un type en Java. Ainsi, si l'on souhaite associer un objet de la classe `Scanner` à une variable `sc`, on peut déclarer la variable ainsi :

```
Scanner sc;
```

Pour créer un objet, on combine trois éléments :

— le mot-clé **`new`**,
— un appel à un *constructeur*, qui est une fonction destinée à créer et initialiser l'objet, et dont le nom est précisément le nom de la classe,
— des paramètres fournis au constructeur pour définir l'état initial de l'objet.

Pour créer un objet de la classe `Scanner` qui permettra de récupérer les éléments donnés par l'utilisateur sur l'entrée standard, appelée `System.in` en Java, on pourra donc écrire :

```
new Scanner(System.in)
```

On peut combiner ces deux éléments pour, d'un même geste, déclarer la variable `sc` et créer l'objet qu'elle doit contenir :

```
Scanner sc = new Scanner(System.in);
```

Note : les deux occurrences du nom de classe `Scanner` ici ont des rôles différents. La première est comprise comme le type d'une variable, la deuxième commen un constructeur, c'est-à-dire une fonction particulière.

Une fois l'objet `sc` créé, on peut l'utiliser par l'intermédiaire des méthodes définies par la classe `Scanner`. Pour demander à un objet `o` d'exécuter une méthode `m` avec les paramètres `p1`, `p2`, ..., `pN`, on utilise la notation `o.m(p1, p2, ..., pN)`, avec un point entre l'objet et le nom de la méthode. Par exemple :

```
int i = sc.nextInt();    // lecture d'un entier
String s = sc.next();    // lecture d'un élément
sc.close();              // fermeture de la source
```

La création d'un objet `sc` de la classe `Scanner` définit l'état interne de l'objet, notamment en lui fournissant un lien vers la source depuis laquelle il doit lire (dans notre exemple, l'entrée standard, c'est-à-dire en pratique le terminal). Cet état interne contient également des éléments implicites, comme la manière dont les éléments sont séparés, ou la base à utiliser pour lire les nombres. Des méthodes particulières permettent de consulter ou personnaliser ces autres éléments.

Notez qu'il existe plusieurs constructeurs pour `Scanner`, qui ont tous le même nom mais prennent des paramètres de types différents. On dit que le constructeur est *surchargé*. Le choix de la version à appliquer effectivement est fait en fonction des types des paramètres fournis.

Bibliothèque standard. Les classes fournies par Java sont nombreuses (plusieurs milliers). Seules certaines classes basiques sont chargées automatiquement, et les autres doivent être mentionnées explicitement au début du fichier.

Pour notre classe `Scanner` il faut ainsi la ligne

```
import java.util.Scanner;
```

La notation avec plusieurs mots séparés par des points reflète l'organisation des différentes classes en différents groupes (des *packages*, on y reviendra plus tard). En l'occurrence, la classe `Scanner` est dans le groupe `util`, de la plateforme standard `java`.

Documentation. La documentation officielle de Java est disponible en ligne sur le site d'Oracle, la société qui maintient et édite ce langage. Voici le lien vers la page racine permettant de naviguer dans l'ensemble des classes.
`https://docs.oracle.com/en/java/javase/18/docs/api/index.html`
La plupart des éléments que vous utiliserez dans ce cours se trouveront dans les groupes `java.util`, `java.io` et `java.lang`, eux-mêmes appartenant au module `java.base`.

Prenez l'habitude de consulter ces pages pour chaque classe que vous utilisez. La page de documentation d'une classe contient notamment, dans l'ordre :

— une description générale de la classe et de la manière dont on l'utilise,
— une liste synthétique de tous les constructeurs et de toutes les méthodes proposées par la classe,
— une description détaillées de chaque constructeur et chaque méthode.

Du fait du nombre parfois très grand de méthodes, il n'est pas pertinent de connaître toute la liste par cœur (pour `Scanner` : 14 constructeurs et 58 méthodes). Il est utile de connaître :

— les principes généraux de fonctionnement décrits par la première partie de la page,
— les constructeurs et méthodes que vous utilisez le plus souvent.

Pour le reste, il suffit de retourner voir la documentation en fonction des besoins.

Collections. La bibliothèque standard de Java contient une variété de classes pour manipuler des collections d'objets (ces objets pouvant eux-mêmes être des instances d'autres classes). Comme les tableaux primitifs que vous connaissez déjà, ces classes sont paramétrées par le type des éléments qu'elles vont regrouper.

On a ainsi une classe `ArrayList<T>` pour les tableaux redimensionnables contenant des objets de type `T`, ou `HashMap<K, V>` pour des tables associant des objets de type `K` (les « clés ») et des objets de type `V` (les « valeurs »). Par exemple :

— `ArrayList<Integer>` : tableaux (redimensionnables) d'entiers,
— `ArrayList<String>` : tableaux (redimensionnables) de chaînes de caractères,
— `HashMap<String, Integer>` : tables représentant des associations entre des chaînes de caractères et des entiers.

Ces collections s'utilisent ensuite de la même manière que toute objet, en utilisant les constructeurs et méthodes associés.

```
ArrayList<String> t = new ArrayList<String>(); // création d'un tableau vide
t.add("Bjour ");
t.add("tout ");
t.add("monde.");      // ajout de trois éléments l'un à la suite de l'autre
t.set(0, "Bonjour "); // modification d'un élément
t.add(2, "le ");      // insertion d'un élément à l'indice 2
for (int i = 0; i < t.size(); i++) {  // pour chaque indice valide...
    System.out.print(t.get(i));       // ... affichage des éléments
}
```

Ne pas oublier d'ajouter au début du fichier la ligne `import java.util.ArrayList;`

Enfin, de nombreuses collections permettent l'itération à l'aide d'une boucle *for each*, où on ne passe pas par les indices, mais où à la place on nomme directement les éléments.

```
for (String s : t) {
  System.out.print(s);
}
```

Notez que les éléments successifs du tableau sont désignés tour à tout par une variable `s`, dont le type `String` est déclaré dans la boucle.

Un programme. Ce programme donne une variante du programme `mandelbrot`, qui construit un fichier `.ppm` (un format d'image) avec la fractale. Pour ajouter progressivement les éléments à notre fichier, on utilise un objet `b` de la classe `BufferedWriter`. Cet objet est lui-même lié avec un objet `w` de la classe `FileWriter`, qui interagit avec le fichier. Ce programme doit être écrit dans un fichier `mandelbrot.java`. Après compilation, on peut l'utiliser avec la ligne `java mandelbrot -0.7 0 1.2 400 dessin` pour créer un fichier `dessin.ppm`. Variez les trois premiers paramètres (coordonnées du centre et rayon) pour explorer la fractale. La ligne `static final int MAXTIME = 50;` définit une constante

`MAXTIME` valant 50 (on verra plus tard ce que cette ligne signifie précisément). Le code contient également un mécanisme **`try/catch`** de gestion d'erreur que l'on verra plus tard.

```
import java.io.FileWriter;
import java.io.BufferedWriter;
import java.io.IOException;

public class mandelbrot {

    static final int MAXTIME = 50; // augmenter la valeur pour plus de niveaux de gris

    public static int escapeTime(double cx, double cy) {
        double zx = 0, zy = 0;
        for (int i = 0; i < MAXTIME; i++) {
            double z = zx * zx - zy * zy ;
            zy = 2 * zx * zy + cy ;
            zx = z + cx ;
            if (zx * zx + zy * zy > 4) { return i ; }
        }
        return MAXTIME ;
    }

    public static int[][] computeMandelbrot(double x0, double y0, double radius, int res) {
        double xMin = x0-radius, yMin = y0-radius;
        double pixelSize = radius/res;
        int[][] map = new int[2*res+1][2*res+1];
        for (int y = 0; y < 2*res+1; y++) {
            for (int x = 0; x < 2*res+1; x++) {
                map[y][x] = escapeTime(xMin + x*pixelSize, yMin + y*pixelSize);
            }
        }
        return map;
    }

    public static void write(int[][] map, String filename) {
        try {
            FileWriter w = new FileWriter(filename);
            BufferedWriter b = new BufferedWriter(w);
            b.write("P3\n");
            b.write(map[0].length + " " + map.length + "\n");
            b.write(MAXTIME + "\n");
            for (int[] row : map) {
                for (int t: row) {
                    b.write(t + " " + t + " " + t + " ");
                }
                b.newLine();
            }
            b.close();
        } catch (IOException e) { e.printStackTrace(); }
    }

    public static void main (String[] args) {
        double x0 = Double.parseDouble (args[0]);
        double y0 = Double.parseDouble (args[1]);
        double radius = Double.parseDouble (args[2]);
        int res = Integer.parseInt (args[3]);
        int[][] map = computeMandelbrot (x0, y0, radius, res);
        String filename = args[4] + ".ppm";
        write(map, filename);
    }
}
```

Fiche 3 : définition de classes

IPO – Introduction à la programmation objet

Thibaut Balabonski @ Université Paris-Saclay
`https://www.lri.fr/~blsk/IPO/`

V1, automne 2022

Structure générale Une *classe* définit un type d'objets qui pourront être manipulés dans un programme. La définition d'une classe comporte :

- une description des données qui constituent l'état interne d'un objet (les *attributs*),
- une ou plusieurs fonctions dédiées à la création d'un nouvel objet (les *constructeurs*),
- l'ensemble des opérations que peut réaliser un objet de cette classe (les *méthodes*).

Syntaxe : la définition d'une classe est introduite par les mots-clés **`public class`**, suivis du nom donné à la classe. La définition elle-même est ensuite placée entre accolades, et énumère les attributs, constructeurs et méthodes.

```
public class Point {
  ...
}
```

Attributs On déclare les *attributs* comme on déclarerait des variables, en fournissant leur type et leur nom. On place généralement avant cette déclaration le mot-clé **`private`**.

```
  // deux attributs : coordonnées du point
  private double x, y;
```

Chaque objet d'une classe possède ses propres valeurs pour chaque attribut de la classe. Dans notre exemple, toute instance de la classe `Point` possède donc une valeur pour `x` et une valeur pour `y`. On accède à l'attribut `x` de l'objet `o` avec la notation pointée `o.x`, *lorsque cet accès est effectivement autorisé.*

Méthodes Une *méthode* est une opération qui peut être réalisée sur les objets de la classe que l'on est en train de définir. La définition d'une méthode se présente comme une définition de fonction, avec un nom, un type de retour, d'éventuels paramètres, et un code entre accolades. On place généralement le mot-clé **`public`** avant cela.

```
  // calcul de la distance au point de coordonnées (0, 0)
  public double norme() {
    ...
  }
```

La méthode ci-dessus, appelée `norme`, prend zéro paramètres et renvoie un nombre de type **`double`** représentant la distance à l'origine du point auquel on l'applique. On invoque cette méthode `norme` pour un objet `o` avec la notation pointée `o.norme()`. Dans une telle invocation, l'objet `o` est appelé le *paramètre implicite*. Les éventuels autres paramètres passés entre parenthèses sont les paramètres explicites.

Dans le corps d'une méthode, on peut faire référence à ses éventuels paramètres mais également à l'objet lui-même auquel la méthode est appliquée. Cet objet est désigné par le mot-clé **`this`**. On fait référence à l'attribut `x` de l'objet exécutant la méthode avec la notation **`this`**`.x`. On peut donc compléter le corps de la méthode `norme` par la ligne

```
    return Math.sqrt(this.x * this.x + this.y * this.y);
```

Pendant l'exécution d'une méthode déclenchée par un appel `p.norme()`, l'objet **`this`** auquel on fait référence est précisément le paramètre implicite `p`.

Constructeurs Un *constructeur* est une fonction dont le nom est précisément le nom de la classe. Son code a pour objectif d'initialiser les attributs de l'objet construit, en utilisant plus ou moins directement des paramètres fournis. Le constructeur ne donne pas de type de retour (on sait qu'il sert précisément à initialiser un objet de la classe) et est généralement précédé du mot-clé **`public`**.

```
  // construction d'un point avec les coordonnées fournies
  public Point(double x, double y) {
    this.x = x;
    this.y = y;
  }
```

Notez, dans une instruction telle que **`this`**`.x = x;`, la différence entre le paramètre `x` du constructeur et l'attribut `x` de l'objet construit. Le deuxième est désigné par **`this`**`.x`.

Dans un fragment de code Java, on peut créer un objet à l'aide du mot-clé **new** et du constructeur de la classe choisie.

```
new Point(1.0, 2.0)
```

Dans les cas simples, une telle construction est directement associée à la déclaration d'une variable, dont le type est le nom de la classe.

```
Point p = new Point(1.0, 2.0);
```

L'objet ainsi construit est matérialisé physiquement par une petite structure en mémoire, contenant notamment les valeurs `1.0` et `2.0` associées à ses attributs `x` et `y`. La variable `p` contient alors un pointeur vers cette structure.

Affichage Tout objet `o` peut être affiché par l'instruction `System.out.println(o);`. Par défaut, l'information affichée est l'adresse de la structure correspondante en mémoire, ce qui n'est pas toujours le plus informatif :

```
Point@659e0bfd
```

Pour obtenir un affichage plus centré sur le contenu d'un objet, il faut définir dans la classe une méthode `toString` renvoyant la chaîne de caractères que l'on souhaite afficher pour un objet donné.

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

Note : pour que cette méthode d'affichage soit prise en compte par `System.out.println`, il faut scrupuleusement respecter la signature de la méthode : mot-clé **public**, type de retour `String`, pas de paramètres.

Égalité L'opérateur == en Java permet de tester l'égalité de deux valeurs. Appliqué à deux objets `o1` et `o2`, cet opérateur vérifie que `o1` et `o2` sont *physiquement* identiques, c'est-à-dire que les structures sous-jacentes occupent les mêmes adresses en mémoire.

Pour comparer deux objets sur la base de leur contenu, on peut définir une méthode dédiée, qui compare l'objet courant avec un autre objet donné en paramètre.

```
public boolean egale(Point p) {
  return this.x == p.x && this.y == p.y;
}
```

On peut alors remplacer la comparaison *physique* `o1 == o2` par la comparaison *logique* `o1.egale(o2)`.

Par convention, la méthode que l'on définit pour cette comparaison a une signature légèrement différente, qui accepte en paramètre un objet de n'importe quel type (et porte le nom anglais `equals`).

```
public boolean equals(Object o) {
  if (o instanceof Point) {
    Point p = (Point)o;  // p pointe vers le même objet que o, mais a le bon type
    return this.egale(p);
  } else {
    return false;
  }
}
```

Cette version a une forme très codifiée : le paramètre est du type `Object` qui représente n'importe quel objet de n'importe quelle classe, puis un test avec le mot-clé **instanceof** sépare les cas où cet objet est effectivement de la classe voulue ou non. Si l'objet appartient à la bonne classe, on le *convertit* pour indiquer au vérificateur de types comment considérer l'objet (ici, comme un `Point`) puis on peut utiliser la méthode de comparaison précédente. Enfin, dans le cas où l'objet n'appartient pas à la bonne classe on peut de toute façon renvoyer **false**.

Encapsulation Selon les principes de la programmation objet, on interagit avec un objet en utilisant ses méthodes, et non en accédant directement à son contenu (c'est-à-dire à ses attributs). Il s'agit de séparer

1. la manière dont un objet fonctionne, et
2. la manière dont on l'utilise.

Un programme utilisant un objet donné n'a besoin de connaître que le *mode d'emploi* de cet objet. Le contenu exact et le fonctionnement interne sont des « détails d'implémentation » que, en tant qu'utilisateur, on n'a pas à connaître. Cette séparation a (au moins) trois avantages :

— cela fait moins de choses à connaître/comprendre pour l'utilisateur,

— cela permet d'apporter des modification au fonctionnement interne d'un objet (amélioration de l'efficacité, correction de bugs, adaptation à un nouvel environnement) sans que l'utilisateur ait besoin de s'adapter, tant que les modifications préservent le mode d'emploi,
— si le fonctionnement interne d'une classe repose sur une certaine cohérence entre les différents attributs d'un objet, les restrictions d'accès évitent qu'un utilisateur mal informé ne vienne invalider cette cohérence et perturber le fonctionnement de l'ensemble.

Ainsi, le mot-clé **`private`** utilisé dans les déclarations d'attributs indique que ces attributs font partie du fonctionnement interne de la classe et ne doivent pas être utilisés directement à l'extérieur de cette classe. À l'inverse, le mot-clé **`public`** du constructeur et des méthodes indique que tout le monde peut compter sur ces éléments, et les utiliser librement. Le compilateur Java vérifie que ces restrictions d'accès sont bien respectées.

En dehors de **`public`** et **`private`**, il existe deux autres niveaux d'accessibilité que nous verrons plus tard. Nous verrons également des situations dans lesquelles des attributs peuvent être publics, et des méthodes privées.

Invariants L'un des intérêts de l'encapsulation est de protéger les *invariants* d'un objet, c'est-à-dire les propriétés assurant la cohérence de son état interne. Pour une classe représentant des paires $\{x, y\}$ d'entiers, on peut par exemple proposer deux attributs pour les deux valeurs de la paire :

```
public class Paire {
  private int a, b;
}
```

Cependant, remarquez que les deux structures | Paire | a : 1 | b : 2 | et | Paire | a : 2 | b : 1 | représentent la même paire $\{1, 2\}$! On peut choisir une règle qui fixe une représentation unique pour ces deux versions, en décidant par exemple que le plus petit élément de $\{x, y\}$ sera stocké dans l'attribut `a`, et l'autre dans l'attribut `b`. On peut le forcer avec un constructeur écrit de la manière suivante.

```
  public Paire(int x, int y) {
    if (x < y) {
      this.a = x;
      this.b = y;
    } else {
      this.a = y;
      this.b = x;
    }
  }
```

Ensuite, les restrictions d'accès aux attributs `a` et `b` assure qu'aucune personne extérieure ne viendra directement manipuler ces attributs. Autrement dit, l'accès aux attributs est restreint à la personne qui développe cette classe elle-même, qui sait quelle contrainte d'ordre doit être respectée.

Un avantage d'un tel choix est que certaines méthodes vont devenir plus simples et plus efficaces : pour tester l'égalité entre deux paires, si on sait que $a < b$ pour tout objet, on peut se contenter de la méthode

```
public boolean egale(Paire p) {
  return this.a == p.a && this.b == p.b;
}
```

alors que sans cette hypothèse il aurait fallu tenir compte de plus de possibilités.

Méthodes statiques Une définition de classe Java peut également contenir des méthodes qui *n'ont pas accès* à un objet **`this`**. De telles méthodes sont qualifiées de « statiques » (on verra la raison de ce choix de vocabulaire plus tard) et correspondent au concept habituel de *fonction*.

Une telle méthode est déclarée avec le mot-clé **`static`**, et déclare explicitement tous ses paramètres.

```
public static double compareNorme(Point p1, Point p2) {
  return p1.norme() - p2.norme();
}
```

Cette méthode n'a pas accès à un objet **`this`**, et donc pas non plus accès aux attributs de la classe `Point`. Elle peut en revanche faire appel aux méthodes des objets passés en paramètres.

On appelle une méthode statique avec la notation pointée habituelle, en mettant à gauche du point le nom de la classe (plutôt qu'un objet comme dans une méthode habituelle).

```
double d = Point.compareNorme(p1, p2);
```

Note : le **`static`** apparaissant dans la déclaration de la fonction `main` est bien celui que l'on vient de décrire !

Fiche 4 : définition de classes, bis

IPO – Introduction à la programmation objet

Thibaut Balabonski @ Université Paris-Saclay
`https://www.lri.fr/~blsk/IPO/`

V1, automne 2022

Environnement de développement Un *environnement de développement* (IDE) regroupe un ensemble d'outils utiles pour écrire, compiler, tester, débugger (et encore d'autres choses...) un projet logiciel. Pour ce cours, on suggère *IntelliJ*. Depuis les salles du bâtiment 336, vous pouvez le lancer en exécutant la commande suivante dans un terminal.

```
idea.sh &
```

Pour commencer à programmer :

1. cliquer sur « New Project »,
2. donner un nom au projet, puis valider avec les valeurs par défaut,
3. créer (au moins) un fichier source avec un clic droit sur le répertoire `src`, (New > Java Class),
4. c'est prêt !

L'outil réagit en direct à tout ce qui est écrit. En particulier :

- lorsque l'on commence à écrire un mot, il montre les différentes manières possibles de le compléter, en se basant sur les noms de classes qui existent, ou sur les noms de méthodes et d'attributs qui peuvent être utilisés à cet endroit (les flèches permettent de sélectionner l'une des possibilités, et la touche entrée de valider),
- il signale en rouge les erreurs de compilation,
- il peut émettre des suggestions sur la manière de corriger certaines erreurs ou d'améliorer certaines lignes de code (à utiliser avec précaution !)

Une fois que votre programme possède une méthode `main`, vous pouvez cliquer sur le triangle vert pour le compiler et l'exécuter. S'affiche alors en bas de la fenêtre un terminal dans lequel on pourra observer le bon déroulement du programme, ou les erreurs survenant à l'exécution.

Exemple de code : associations On définit une classe pour représenter des paires « clé, valeur », formées par une chaîne de caractères (la *clé*) et une valeur entière.

```
public class Association {
  private String key;
  private int value;
```

Rappel : un constructeur doit initialiser les attributs. Il possède donc, en gros, une ligne par attribut.

```
  public Association(String k, int v) {
    this.key = k;
    this.value = v;
  }
```

Les attributs sont privés, ce qui garantit qu'ils ne seront pas modifiés par un utilisateur maladroit. Pour donner accès à un attribut *en lecture*, on peut créer un *getter*, c'est-à-dire une méthode publique dont le seul objectif est de transmettre la valeur d'un attribut.

```
  public String getKey() {
    return this.key;
  }
```

Note : cette méthode ne permet que de consulter l'attribut, pas de le modifier. Pour la modification, on introduit *parfois* une autre méthode appelée *setter*. *Par défaut, ne pas le faire !*

```
  public void setValue(int v) {
    this.value = v;
  }
```

Dans le cas où l'on définit un *setter*, attention : une telle méthode doit s'assurer que les modifications préservent les éventuels invariants de la classe.

Pour compléter notre classe, on peut donner une fonction définissant l'affichage d'une association.

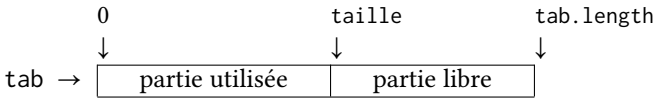
```
  public String toString() {
    return this.key + " -> " + this.value;
  }
}
```

Exemple de code : tableau redimensionnable On définit une classe pour représenter un tableau dans lequel il est possible d'ajouter dynamiquement des éléments. Notez que ceci n'est pas permis par les tableaux primitifs de Java, dont la taille est définitivement fixée lors de leur création.

On donne deux attributs à un tableau redimensionnable : un tableau primitif Java destiné à contenir les éléments, et un entier indiquant quelle partie du tableau concret est effectivement utilisée.

```
public class TableauRedimensionnable {
  private Association[] tab;
  private int taille;
```

On peut se figurer la situation ainsi :



À la création, le tableau sous-jacent `tab` a une longueur par défaut, définie arbitrairement, et la `taille` du tableau redimensionnable est fixée à zéro (il n'y a pas encore d'éléments).

```
  public TableauRedimensionnable() {
    this.taille = 0;
    this.tab = new Association[32];
  }
```

Pour accéder à un élément du tableau redimensionnable identifié par un indice `i`, on vérifie que l'indice désigne bien une position de la partie utilisée du tableau sous-jacent. Dans le cas contraire, la ligne commençant par **`throw`** déclenche la même erreur que celle produite par Java lors de l'accès à un indice invalide d'un tableau primitif.

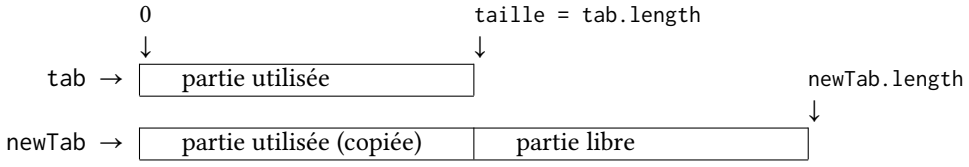
```
  public Association get(int i) {
    if (i < this.taille) {
      return this.tab[i];
    } else {
      throw new ArrayIndexOutOfBoundsException(i);
    }
  }

  public void set(int i, Association a) {
    if (i < this.taille) {
      this.tab[i] = a;
    } else {
      throw new ArrayIndexOutOfBoundsException(i);
    }
  }
```

Note : en cas d'indice négatif, c'est l'accès au tableau sous-jacent lui-même qui déclenche l'erreur.

Pour l'ajout d'un nouvel élément, on a deux cas à traiter. S'il reste effectivement de la place libre dans le tableau `tab` sous-jacent, il suffit d'utiliser la prochaine case, et d'incrémenter la `taille`. S'il ne reste plus de place libre en revanche, il faut au préalable *étendre* le tableau sous-jacent, ce qu'on délègue à une méthode `extend` détaillée ci-dessous.

```
  public void add(Association a) {
    if (this.taille >= this.tab.length) {
      this.extend();
    }
    this.tab[this.taille++] = a;
  }
```

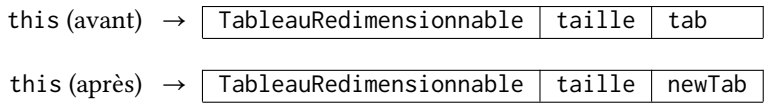
Comme il n'est pas possible de modifier la taille d'un tableau primitif de Java, on va créer un nouveau tableau plus grand, qui va remplacer le tableau d'origine. Pour ne rien perdre, on copie intégralement le contenu du tableau d'origine dans le début du nouveau tableau.



```
  private void extend() {
    Association[] newTab = new Association[2*this.taille];
    for (int i=0; i < this.taille; i++)
    newTab[i] = this.tab[i];
    this.tab = newTab;
  }
```

Note : ce code repose effectivement sur le fait que **this**.taille est la taille réelle tab.length du tableau sous-jacent.

Du point de vue de **this**, c'est-à-dire du tableau redimensionnable lui-même, la méthode extend ne fait rien d'autre que modifier un attribut.



Pour finir, on ajoute une méthode qui cherche dans un tel tableau redimensionnable une association portant une clé k donnée en paramètre. La méthode renvoie l'indice de l'association trouvée, ou -1 à défaut.

```
  public int findKey(String k) {
    for (int i=0; i < this.taille; i++) {
      String kk = this.tab[i].getKey();
      if (kk.equals(k)) {
        return i;
      }
    }
    return -1;
  }
}
```

Note : cette méthode fait appel à la méthode getKey() de la classe Association. En revanche, dans tout ce qui avait été fait avant cela, on ne parlait que du tableau et le type des éléments du tableau n'importait pas.

La classe TableauRedimensionnable définie ici reprend les principes de la classe ArrayList de Java, spécialisée pour contenir des associations (ce que l'on pourrait noter ArrayList<Association>).

Exemple de code : table d'association Pour finir, on définit une classe pour représenter un ensemble d'associations entre des clés (ici, des chaînes de caractères) et des valeurs (ici, des entiers). On veut pouvoir ajouter de nouvelles associations à volonté, et faire en sorte de ne pas avoir deux associations pour une même clé. Autrement dit, ajouter une nouvelle association pour une clé déjà présente doit remplacer l'association d'origine.

On donne pour seul attribut à une table associative un tableau redimensionnable contenant des associations. Ce sont les méthodes qui devront s'assurer de ne jamais créer deux associations pour une même clé.

```
public class TableAssociative {
  private TableauRedimensionnable tab;
  public TableAssociative() {
    this.tab = new TableauRedimensionnable();
  }
```

Pour ajouter une nouvelle association (k, v), on commence donc par chercher une éventuelle association déjà présente pour la même clé k. Si l'on en trouve une, il suffit de la modifier. Dans le cas contraire, on crée une nouvelle association et on l'ajoute au tableau redimensionnable.

```
  public void put(String k, int v) {
    int i = this.findKey(k);
    if (i >= 0) { // remplacer
      this.tab.get(i).setValue(v);
    } else { // ajouter à la fin
      this.tab.add(new Association(k, v));
    }
  }
}
```

Cette classe TableAssociative mime, du point de vue de l'utilisateur, le comportement de la classe HashMap de Java, en fixant les types des clés et des valeurs. Notre TableAssociative correspond donc à une HashMap<String, Integer>. De même, la classe Association correspond à Map.Entry<String, Integer> en Java. Il y a cependant une différence de poids à noter : notre version est inefficace (lente), alors que la vraie HashMap est basée sur un algorithme *très* efficace.

Fiche 5 : conception objet

IPO – Introduction à la programmation objet

Thibaut Balabonski @ Université Paris-Saclay
`https://www.lri.fr/~blsk/IPO/`

V1, automne 2022

Pointeur nul Rappel sur l'organisation des données en Java : chaque objet est matérialisé par une structure en mémoire, allouée dans le *tas*. On manipule un tel objet par l'intermédiaire d'un *pointeur* (autrement appelé une *référence*) vers la structure sous-jacente : lorsque l'on stocke un objet dans une variable ou dans un tableau, ou lorsque l'on passe un objet en paramètre à une méthode, on stocke ou on passe le pointeur, et non la structure entière. Lorsqu'une variable `a` désigne un objet, l'accès à un champ `a.x` consiste donc à aller chercher le champ `x` dans la structure désignée par le pointeur associé à la variable `a`.

Le pointeur **`null`** est un pointeur particulier, qui ne pointe vers *rien*. Il apparaît notamment lorsque l'on crée une structure destinée à contenir un objet, mais que l'objet en question n'a pas encore été fourni. Avec une définition comme

```
Personnage tab = new Personnage[5];
```

on a créé un tableau destiné à stocker cinq personnages, mais ces derniers n'existent pas encore. À défaut d'objets associés en mémoire, les cinq champs de `tab` contiennent pour l'instant le pointeur nul. On l'observe dès que l'on tente d'accéder à un attribut ou une méthode de l'un de ces objets, par exemple avec `tab[3].x`.

```
Exception in thread "main" java.lang.NullPointerException:
  Cannot read field "x" because "<local1>[3]" is null
```

Wrappers En Java, toutes les données manipulées sont des objets, exceptées les données de certains types de base comme **`int`** ou **`double`**. Du fait de l'omniprésence des objets et des classes, certains mécanismes ne fonctionnent qu'avec des classes, et pas avec les types de base. Par exemple : les classes des tableaux redimensionnables `ArrayList` ou des tables de hachage `HashMap` sont paramétrées par des classes, de sorte qu'on puisse écrire des déclarations comme :

```
ArrayList<Personnage> tab;
HashMap<String, ArrayList<Personnage>> map;
```

mais *pas* `ArrayList<int>`.

Pour corriger cela, Java propose des classes appelées *wrappers* associées à chacun des types de base, qui permettent d'utiliser ces types de base comme des classes ordinaires (et les valeurs comme des objets). On pourra donc écrire la déclaration

```
ArrayList<Integer> tab;
```

pour un tableau redimensionnable contenant des entiers.

Un objet `obji` de la classe `Integer` peut être :

— manipulé comme un objet, notamment avec des appels de méthodes

```
obji.compareTo(obj)
```

— ou utilisé comme un entier ordinaire, auquel cas il est effectivement converti en **`int`**.

```
obji + 3
```

Note : la fonction `Integer.parseInt` déjà croisée est donc une fonction fournie par cette classe *wrapper* `Integer`. Elle renvoie une valeur de base de type **`int`**.

Conception La *conception* d'un programme consiste à définir la structure générale du code et des données manipulées, pour réaliser un projet donné. Dans un cadre de programmation objet, on cherche notamment à :

— identifier les classes à définir et leurs attributs,

— déterminer comment chaque action à réaliser va se décomposer en appels de méthodes sur des objets des différentes classes.

On considère l'exemple d'un jeu de plateforme en deux dimensions où évoluent plusieurs personnages. On suppose le terrain divisé en cases. Un personnage est tourné soit vers la gauche, soit vers la droite, et avance d'une case par tour dans cette direction, à moins qu'il ne se trouve au-dessus du vide. Dans ce dernier cas il tombe d'une case par tour, tout en continuant à regarder du même côté. Enfin, le personnage change de direction s'il ne peut pas avancer. On cherche à définir un ensemble de classes permettant de modéliser cette situation, avec des méthodes réalisant les déplacements.

Classes En première approximation, on peut introduire une classe pour chaque élément à manipuler. En l'occurrence, on peut citer : les personnages, le terrain où évoluent les personnages, et une classe principale pour le jeu lui-même. D'autres pourront être ajoutées en fonction des besoins, lorsque l'ensemble deviendra plus précis.

Pour chacune des classes, on énumère les différents éléments composant un objet.

— Pour la classe `Personnage`, on retient au minimum une position et une direction. On pourrait ajouter en fonction des besoins : un identifiant, un état de santé, un équipement...
— Pour la classe `Terrain`, il nous faut au minimum une carte du terrain (zones libres et obstacles) et l'ensemble des personnages présents.
— Pour la classe principale `Jeu`, on demande au minimum un terrain et l'ensemble des personnages actifs.

Ces éléments vont être matérialisés par des attributs, dont le type sera à choisir en fonction des manipulations envisagées.

Cases libres Pour déplacer un personnage, on a besoin de pouvoir déterminer si une case est libre. Cette question est relative au terrain : on peut imaginer la traiter avec une méthode définie dans la classe `Terrain`, prenant en paramètres les coordonnées d'une case (numéro de ligne, numéro de colonne) et renvoyant un booléen.

```
public boolean estLibre(int lig, int col) { ... }
```

Cette méthode doit déterminer si la case ciblée est libre, c'est-à-dire si d'une part elle n'est pas un obstacle du terrain, et d'autre part elle ne contient pas déjà un personnage. Pour le faire efficacement, on propose que les attributs de la classe `Terrain` représentant le terrain lui-même et les personnages présents soient tous deux représentés par des tableaux à deux dimensions.

```
public class Terrain {
  private int hauteur, largeur;
  private boolean[][] carte;
  private Personnage[][] personnages;
```

Dans cette version, on propose de représenter le terrain par un tableau de booléens, indiquant par exemple **`true`** dans le cas d'une case libre, et **`false`** dans le cas d'un obstacle. On pourrait imaginer une version plus élaborée dans laquelle chaque case du terrain est représentée par un objet pouvant avoir des caractéristiques variées.

```
  public boolean estLibre(int lig, int col) {
    return this.carte[lig][col] && this.personnages[lig][col] == null;
  }
```

Déplacements Le déplacement d'un personnage concernant, justement, un personnage, on pourrait vouloir le réaliser par une méthode définie dans la classe `Personnage`. Cependant, l'application de l'algorithme de déplacement demande, entre autres choses, de consulter le statut libre ou non de certaines cases. Or, seul le terrain a cette connaissance, et la classe `Personnage` n'a, en l'état, pas d'accès au terrain. On peut donc plutôt définir une méthode au niveau du `Terrain`, ou du `Jeu` lui-même.

Le déplacement lui-même se manifeste à plusieurs endroits :

— il faut modifier les coordonnées du personnage `p` déplacé,
— il faut modifier le tableau `personnages` du terrain, pour retirer `p` de sa case d'origine et l'ajouter dans sa case de destination.

Ainsi, dans la classe `Jeu` on pourrait définir :

```
  public void joue(Personnage p) {
    int l = p.getLig();
    int c = p.getCol();
    if (this.terrain.estLibre(l+1, c)) {
      this.terrain.deplace(p, l+1, c);
      p.setPos(l+1, c);
    } else {
      ...
    }
  }
```

Cette méthode elle-même repose sur deux méthodes auxiliaires : `setPos` dans la classe `Personnage`, qui modifie les coordonnées d'un personnage,

```
  public void setPos(int lig, int col) {
    this.lig = lig;
    this.col = col;
  }
```

et `deplace` dans la classe `Terrain`, qui met à jour le tableau enregistrant les positions des personnages.

```
public void deplace(Personnage p, int lig, int col) {
  this.terrain[p.getLig()][p.getCol()] = null;
  this.terrain[lig][col] = p;
}
```

Pour capturer d'éventuels bugs, et assurer la robustesse de ce code, on peut ajouter une vérification du fait que les nouvelles coordonnées sont bien des coordonnées valides de notre terrain. Pour cela comme pour le reste, il faut se placer dans une classe disposant de suffisamment d'informations pour réaliser ce test. En l'occurrence, les personnages n'ont pas cette information, mais le terrain l'a. On pourrait donc ajouter la ligne

```
  assert (0 <= lig && lig < this.hauteur && 0 <= col && col < this.largeur);
```

au début de la méthode `deplace`. Ce faisant, on se rendra vite compte qu'il manque *quelque part* une précaution : aucune des méthodes écrites jusqu'ici ne teste si la case de coordonnées (`l+1`, `c`) vers laquelle le personnage est susceptible de chuter appartient bien aux limites du terrain. On a plusieurs manière de corriger cela. Par exemple :

- enrichir la méthode `estLibre` (classe `Terrain`) pour qu'elle ne renvoie **`true`** que si la case cible *existe* effectivement, ou encore
- introduire un test supplémentaire dans la méthode `joue` (class `Jeu`) pour traiter à part cette situation. Cela peut être pour l'interdire, ou au contraire pour déclencher un effet alternatif, comme retirer le personnage du jeu.

Bilan Dans la construction d'un programme, même d'ampleur modeste, on est amené à définir et à faire interagir plusieurs éléments. Le travail de conception demande d'identifier ces éléments, et de délimiter les contours *et les responsabilités* de chacun. Ce qui correspond, vu de haut, à une unique action, est généralement décomposé en plusieurs opérations élémentaires, s'appliquant à l'un ou l'autre des différents éléments présents. Pour faire cette décomposition, et répartir les tâches entre les différents acteurs, il faut observer ce à quoi chacun à accès.

Un problème d'origine étant fixé, il y a souvent de multiples manières légitimes de le résoudre, basées sur différentes organisations des données ou décompositions des actions. Autant que faire se peut, on favorise les solutions les plus *intelligibles*, dans lesquelles la structure générale et la répartition des responsabilités est claire (on affinera ces critères plus tard). Une fois ce découpage fixé, on cherche à l'intérieur de chaque classe les structures de données et algorithmes qui donneront à l'ensemble une efficacité raisonnable.

Fiche 6 : interfaces

IPO – Introduction à la programmation objet

Thibaut Balabonski @ Université Paris-Saclay
`https://www.lri.fr/~blsk/IPO/`

V1, automne 2022

Retour sur l'encapsulation Dans la construction d'un programme on cherche, pour chaque élément, à distinguer la manière dont on l'utilise (point de vue extérieur) et la manière dont il fonctionne (point de vue interne). À un utilisateur extérieur, on présente un mode d'emploi qui dit comment utiliser l'élément, et quels résultats il faut attendre. Ce mode d'emploi extérieur est appelé une *interface*, ou une *abstraction*. Ce point de vue en revanche ne dit pas exactement la manière dont l'élément est organisé en interne.

En java, ceci se manifeste notamment avec la notion d'encapsulation et avec l'habitude de déclarer les attributs « privés », c'est-à-dire inaccessibles à un utilisateur extérieur, et les méthodes « publiques », c'est-à-dire utilisables par un utilisateur extérieur.

Cette séparation a plusieurs vertus.

— Un utilisateur qui n'a pas directement accès aux attributs d'une structure ne pourra pas, par mégarde, les modifier d'une manière qui ne respecterait pas les conventions de cette structure.

— Si l'utilisateur n'a besoin de connaître que le mode d'emploi, et pas tous les détails internes des structures qu'il utilise, alors il a moins de choses à connaître ou à maîtriser : il peut se concentrer sur sa partie.

— En limitant la manière dont les utilisateurs peuvent utiliser une structure, on évite que chaque modification du fonctionnement de cette structure doive être suivie d'une adaptation du code de l'utilisateur. Tant que l'on préserve l'interface, on peut corriger le code de la structure, lui ajouter une nouvelle fonctionnalité ou modifier ses caractéristiques internes sans que cela invalide les programmes utilisant cette structure.

Interface Une *interface* en Java résume les caractéristiques « extérieures » d'une structure. Une interface est essentiellement constituée de déclarations de méthodes, placées dans un bloc ressemblant à une définition de classe. On peut ainsi placer la définition

```
public interface Pile {
  void push(int n);  // ajouter un élément au sommet
  int  pop();        // retirer (et renvoyer) l'élémént du sommet
  int  peek();       // renvoyer l'élément du sommet
  boolean isEmpty(); // la pile est-elle vide ?
}
```

dans un fichier `Pile.java` pour définir l'*interface* d'une structure de données de *pile* (*LIFO : Last In, First Out*).

Chaque méthode est ici résumée à une déclaration : on fournit le nom de la méthode, les types de ses paramètres et le type de son éventuel résultat, mais on ne donne pas de code. Remarquez aussi que, par essence, une interface est une liste d'éléments *publics*. On se passe donc du mot-clé **`public`**, qui serait superflu. Dans l'interface, on ne donne en revanche pas d'attributs (on pourrait, mais cela aurait un sens particulier que nous verrons plus tard), ni de constructeurs (l'interface n'est qu'une vue abstraite, et ne permet pas à elle seule de créer un objet).

L'interface donne les éléments permettant d'utiliser une certaine structure. Notez que l'on peut donc écrire un programme utilisant une pile, en s'appuyant sur cette interface, avant même d'avoir des définitions concrètes pour les différents éléments de la pile. Voici par exemple un programme utilisant une pile `p` et ses méthodes `push` et `pop` pour déterminer la parenthèse ouvrante associée à chaque parenthèse fermante dans une chaîne de caractères. La seule partie laissée en suspens est la création de la pile, puisque nous n'avons à ce stade pas de constructeur.

```
public static int[] matching(String s) {
  int[] m = new int[s.length()];
  Pile p = new ???;
  for (int i=0; i<s.length(); i++) {
    if (s.charAt(i) == '(') { p.push(i); }
    else if (s.charAt(i) == ')') { m[i] = p.pop(); }
  }
  return m;
}
```

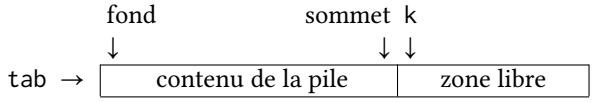
De la même manière que pour les classes, une interface définit un *type*. Ici, l'occurrence de `Pile` dans la déclaration `Pile p` désigne le type associé à la variable `p`.

Concrétisation d'une interface Une interface `I` étant fixée, on peut déclarer qu'une classe concrète `C` *réalise* cette interface en ajoutant une mention **`implements`** `I` dans la définition de `C`.

```
public class TabPile implements Pile {
  ...
}
```

Pour que cela soit admis, il *faut* que la classe concrète `TabPile` fournisse des définitions pour chacune des méthodes déclarées dans l'interface que l'on prétend réaliser. La classe concrète doit également contenir comme d'habitude des attributs, qui définissent sa structure, et au moins un constructeur, qui permet de créer des objets concrets.

On propose ici de représenter une pile d'entiers à l'aide d'un tableau primitif `tab`, de taille arbitraire mais fixe. Un attribut supplémentaire `k` donne l'indice de la première case libre du tableau, où sera ajouté le prochain élément introduit par la méthode `push`.



On peut alors simplement définir une telle classe `TabPile`, en déclarant ces deux attributs, en fournissant un constructeur qui initialise ces deux attributs, et en donnant une définition pour chacune des quatre méthodes mentionnées dans l'interface.

```
public class TabPile implements Pile {
  private int[] tab; // tableau sous-jacent
  private int k;     // première case libre

  public TabPile(int n) {
    this.tab = new int[n];
    this.k = 0;
  }

  public void    push(int n) { tab[k++] = n;    }
  public int     pop()       { return tab[--k]; }
  public int     peek()      { return tab[k-1]; }
  public boolean isEmpty()   { return k == 0;   }
}
```

En l'occurrence ici, les méthodes sont très simples et chacune tient en une ligne, mais on peut bien sûr avoir un code de complexité arbitraire. Note à propos de ce code : dans les méthodes, on accède aux attributs de l'objet courant sans utiliser le mot-clé **`this`**. Celui-ci est en effet facultatif lorsqu'il n'y a pas d'ambiguïté sur l'élément `tab` ou `k` désigné.

Utilisation conjointe d'une interface et d'une classe concrète associée Une interface ne fait que définir un *type*, et les méthodes auxquelles peut faire appel un utilisateur de ce type. Une classe concrète indique en revanche comment construire des objets concrets réalisant le type précédent. On peut donc compléter la ligne incomplète de notre fonction `matching` de la manière suivante.

```
Pile p = new TabPile();
```

On crée donc un objet de la classe (concrète) `TabPile`, que l'on associe à une variable `p` dont le type est donné par l'interface `Pile`. C'est possible car, du fait de la mention **`implements`** `Pile` dans la définition de la classe `TabPile`, tout objet de la classe `TabPile` est *à la fois* du type `TabPile` et du type `Pile`. En effet, tout objet `p` de la classe `TabPile` possède bien des méthodes `push`, `pop`, `peek` et `isEmpty`, et peut donc être utilisé à tout endroit où l'on attendait un élément de type `Pile`.

Notez qu'en revanche, la classe concrète `TabPile` peut définir des choses *en plus* de ce qui est déclaré dans l'interface. On pourrait ainsi par exemple ajouter une méthode publique

```
public boolean isFull() { return k == tab.length; }
```

dans la classe `TabPile`, qui fait référence à une caractéristique spécifique des piles concrètes `TabPile` mais n'existe pas dans l'interface générale des piles.

Cette possibilité de spécialisation de la classe concrète a deux conséquences.

1. Tout objet de type `Pile` ne peut pas nécessairement être considéré comme étant également du type `TabPile`. En effet, une pile concrète quelconque ne définit pas nécessairement la méthode `isFull` qu'un utilisateur de `TabPile` pourrait attendre.
2. Une même classe peut concrétiser simultanément *plusieurs* interfaces. On note par exemple

   ```
   public class TabPile implements Pile, Iterable<Integer> { ... }
   ```

   pour déclarer que la classe `TabPile` concrétise également l'interface `Iterable<Integer>` des structures permettant d'énumérer un ensemble d'entiers (voir `java.util.Iterable`).

Digression : surcharge statique et constructeurs multiples Dans une classe Java, on peut définir plusieurs méthodes portant le même nom, à conditions que ces méthodes puissent être distinguées par les types de leurs paramètres. On parle de *surcharge statique* de ces méthodes. Lorsque l'on utilise une telle méthode dans le code, le compilateur Java analyse les types des arguments fournis et choisit, parmi les différentes méthodes du même nom, celle dont les types des arguments attendus correspondent.

On utilise couramment ce système pour affecter des valeurs par défaut à certains paramètres qui ne seraient pas fournis, et cela s'applique aussi bien aux méthodes qu'aux constructeurs. On peut ainsi fournir deux constructeurs à la classe `TabPile` : celui déjà vu prenant en paramètre la taille à donner au tableau sous-jacent, et un autre ne prenant pas de paramètre et utilisant à la place une taille par défaut.

```
  public TabPile(int n) {
    this.tab = new int[n];
    this.k = 0;
  }
  public TabPile() {
    this.tab = new int[42];
    this.k = 0;
  }
```

Une création `new TabPile(5)` utilise alors le premier constructeur, et une création `new TabPile()` le deuxième. Remarque supplémentaire ici : la deuxième version correspond précisément à appeler la première version avec le paramètre par défaut `42`. Plutôt que de recopier le code, on peut donc explicitement faire appel à l'*autre* constructeur en lui fournissant cette valeur. On utilise le mot-clé **`this`** pour faire ainsi référence à un constructeur de la classe courante.

```
  public TabPile() { this(42); }
```

Note : utilisé ainsi, **`this`** fait référence à *un* constructeur de `TabPile`. Le choix parmi les différents constructeurs de cette classe est fait en regardant le type des paramètres fournis. On ne peut utiliser cette notation que dans un constructeur, et uniquement comme première instruction.

Concrétisations alternatives On peut proposer de multiples manières de concrétiser une interface donnée, les différents classes obtenues pouvant avoir des caractéristiques très différentes. Voici une concrétisation de l'interface `Pile` bâtie sur une structure de liste de chaînée, dotée d'une méthode publique supplémentaire `size`.

```
  class Cellule {
    private int k;
    private Cellule next;
    public Cellule(int k, Cellule n) {
      this.k = k;
      this.next = n;
    }
    public int hd() { return k; }
    public Cellule tl() { return next; }
  }

  public class ListePile implements Pile {
    private Cellule liste;
    private int size;
    public ListePile() {
      this.liste = null;
      this.size = 0;
    }
    public void push(int n) {
      this.liste = new Cellule(n, this.liste);
      size++;
    }
    public int pop() {
      size--;
      int e = liste.hd();
      liste = liste.tl();
      return e;
    }
    public int peek() { return liste.hd(); }
    public boolean isEmpty() { return this.liste == null; }
    public int size() { return this.size; }
}
```

Fiche 7 : extension d'une classe

IPO – Introduction à la programmation objet

Thibaut Balabonski @ Université Paris-Saclay
https://www.lri.fr/~blsk/IPO/

V1, automne 2022

Extension d'une classe Le mécanisme d'*extension* permet de définir une nouvelle classe en réutilisant le contenu d'une classe pré-existante. Ceci permet d'éviter des redondances dans le code, en nous dispensant de dupliquer le code de méthodes qui devraient sinon se trouver dans plusieurs classes.

Ainsi, si l'on doit manipuler deux classes `Point` et `PointCouleur` représentant respectivement un point dans le plan, et un point du plan doté d'une couleur, définies par les deux codes suivants,

```
class Point {
  private double x, y;
  public double norme() { return Math.sqrt(x*x + y*y); }
}

class PointCouleur {
  private double x, y;
  private String couleur;
  public double norme()      { return Math.sqrt(x*x + y*y); }
  public String getCouleur() { return this.couleur; }
}
```

on peut s'éviter la redondance de la déclaration des attributs `x` et `y` et du code de calcul de la norme en déclarant que la classe `PointCouleur` est une extension de la classe `Point`. On indique ceci à l'aide du mot-clé **`extends`** dans la ligne de déclaration de la classe. On n'écrit alors plus dans la définition de `PointCouleur` que les choses qui viennent *en plus* de ce qui est déjà présent dans `Point`.

```
class PointCouleur extends Point {
  private String couleur;
  public String getCouleur() { return this.couleur; }
}
```

Avec cette définition, on indique que toute instance de la classe `PointCouleur` possède, en plus de l'attribut `couleur` et de la méthode `getCouleur` explicitement déclarés, les champs définis par la classe `Point`. C'est-à-dire : les deux attributs `x` et `y` et la méthode `norme`. La classe `PointCouleur` est appelée une *sous-classe*, ou *classe fille*, et la classe `Point` une *super-classe*, ou *classe mère*.

On peut ajouter à la classe `Point` un constructeur de la manière habituelle, mais la situation est un peu différente pour le constructeur de `PointCouleur`.

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

```
public PointCouleur(double x, double y, String couleur) {
  this.x = x;     // incorrect
  this.y = y;     // incorrect
  this.couleur = couleur;
}
```

En effet, ces deux classes sont des classes *différentes*. Les attributs `x` et `y` de `Point`, qui sont privés, ne sont donc pas accessibles depuis la classe `PointCouleur`. À la place, on peut appeler le constructeur de la classe `Point` dans la définition du constructeur de `PointCouleur`, pour lui déléguer l'initialisation de ces deux champs. On le fait avec le mot-clé **`super`**, qui désigne la super-classe.

```
public PointCouleur(double x, double y, String couleur) {
  super(x, y);    // correct
  this.couleur = couleur;
}
```

Notez que l'on évite ainsi une redondance de plus dans le code !

Alternativement, on aurait pu donner aux attributs `x` et `y` une visibilité intermédiaire entre **`private`** et **`public`**, nommée **`protected`**, qui rend un champ accessible aux éventuelles sous-classes de la classe courante.

Sous-typage La classe `PointCouleur` possédant implicitement toutes les méthodes de la classe `Point`, on peut calculer la norme d'un objet `PointCouleur` :

```
PointCouleur pc = new PointCouleur(1., 2., "vert");
System.out.println(pc.norme());
System.out.println(pc.getCouleur());
```

En outre, toute instance de la classe `PointCouleur` peut également être vue comme une instance de la classe `Point`. Ainsi, on peut stocker un objet `PointCouleur` dans une variable variable destinée à contenir un `Point`.

```
Point p = new PointCouleur(1., 2., "vert");
System.out.println(p.norme());
```

De même, on peut donner un `PointCouleur` en argument à toute méthode qui attendrait un `Point`. Si la classe `Point` possède une méthode

```
public double distance(Point other) {
  Point v = new Point(this.x-other.x, this.y-other.y);
  return v.norme();
}
```

telle que `p1.distance(p2)` renvoie la distance entre le point `p1` et le point `p2`, on peut lui fournir en paramètre des `PointCouleur` aussi bien que des `Point`. On dit que le type `PointCouleur` est un *sous-type* du type `Point`, c'est-à-dire un *cas particulier* de `Point`.

À noter : cette notion de « cas particulier » ne fonctionne que dans un sens ! On ne peut pas prendre un `Point` quelconque et lui appliquer un traitement spécifique aux `PointCouleur`.

```
Point p = new Point(1., 2.);
String s = p.getCouleur();   // incorrect
```

Et pour cause, le point `p` de cet exemple appartient à la classe `Point`. Il ne possède donc pas d'attribut `couleur` dont on pourrait renvoyer le contenu. L'appel `p.getCouleur()` est rejeté par le compilateur, car le type `Point` déclaré pour `p` ne propose pas de méthode `getCouleur`.

Transtypage (*cast*) Les vérifications du compilateur sont faites sur la base des types déclarés pour les différents éléments. Ainsi, l'appel `p.getCouleur()` est rejeté même dans la situation suivante, où l'objet associé à `p` est effectivement un `PointCouleur`.

```
Point p = new PointCouleur(1., 2., "vert");
String s = p.getCouleur();   // rejeté à la vérification des types
```

Si l'on a de bonnes raisons de savoir qu'une variable `p` de type `Point` donnée contient plus précisément un `PointCouleur`, on peut cependant l'indiquer, en faisant figurer ce type entre parenthèses devant `p`. On parle de *transtypage*. Lors de la vérification des types, l'élément `(PointCouleur)p` sera donc considéré comme un `PointCouleur`, dont on peut bien appeler la méthode `getCouleur`.

```
String s = ((PointCouleur)p).getCouleur();   // accepté, mais peut échouer
```

Dans ce cas, Java vérifie lors de l'exécution que l'instance réelle associée à `p` est bien du type `PointCouleur`. Si ce n'était pas le cas, l'exécution du programme serait interrompue et indiquerait une `ClassCastException`.

À noter : appliqué à des classes, ce transtypage n'est qu'une *indication* donnée au compilateur, et ne peut être utilisé qu'entre des classes qui sont effectivement parentes. Il n'implique aucun autre effet à l'exécution que la vérification que l'on vient de décrire, et en particulier il ne modifie pas l'objet lui-même. Les choses sont différentes lorsque l'on applique ce même opérateur à des types de base, où on peut cette fois avoir réellement une *conversion* des données sous-jacentes au bon format.

```
int x = (int)1.41;           // changement de représentation concrète
Integer x = (Integer)1.41;   // incorrect, car Double et Integer incompatibles
```

Le mot-clé **`instanceof`** permet, à l'exécution, de tester l'appartenance d'un objet à une classe donnée, pour ne faire de transtypage que dans les cas légitimes.

```
String s;
if (p instanceof PointCouleur) {
  s = ((PointCouleur)p).getCouleur();   // accepté, et ne peut pas échouer
}
```

On a déjà utilisé ce mécanisme dans la définition de méthodes `equals(Object other)`, pour tester l'égalité entre un objet courant d'une classe que l'on est en train de définir, et un autre objet a priori quelconque.

Hiérarchie de classes Il est possible d'enchaîner les extensions, c'est-à-dire définir une classe étendant une autre classe qui aurait elle-même été obtenue par une extension.

```
class PointClignotant extends PointCouleur {
  private boolean jour;
  public void inverse() { this.jour = !this.jour; }
}
```

Cette nouvelle classe possède alors, en plus de ce qu'elle définit elle-même explicitement, tous les éléments définissant `PointCouleur`, dont certains sont eux-mêmes hérités de `Point`.

Au sommet de cette hiérarchie, on trouve en Java une classe prédéfinie `Object`, dont toutes les autres classes sont une extension (directe ou non). Ainsi, lorsque l'on écrit

```
class Point { ... }
```

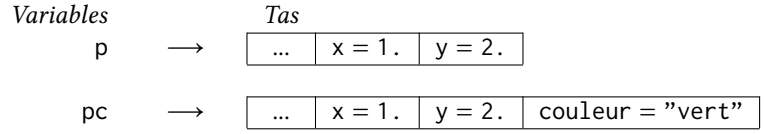
la classe `Point` est en réalité définie comme une extension de la classe `Object`.

```
class Point extends Object /* implicite */ { ... }
```

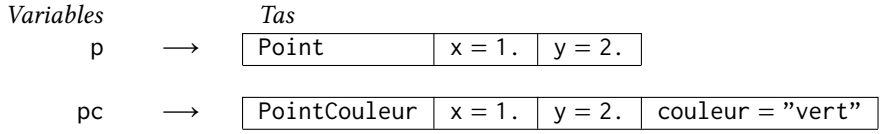
Et lorsque l'on définit `PointCouleur` ou `PointClignotant` comme des extensions (directes ou non) de `Point`, ces nouvelles classes deviennent égalements des extensions (indirectes) de `Object`.

C'est par ce mécanisme que le type `Object` désigne l'ensemble des objets que l'on peut construire en Java : toute instance d'une classe `C` quelconque, par sous-typage, est encore une instance des classes que `C` étend, et donc en particulier d'`Object`.

Modèle mémoire On a déjà évoqué que tout objet manipulé en Java est matérialisé par une structure de données en mémoire, contenant notamment les valeurs de chacun des attributs. Ces structures sont placées dans le *tas* mémoire, et on manipule ensuite concrètement chaque objet par l'intermédiaire d'un pointeur désignant l'adresse correspondante du tas.



En plus des valeurs des différents attributs, la structure de données matérialisant un objet contient une indication de la classe « réelle » de l'objet, c'est-à-dire de la classe du constructeur qui a été utilisé au moment de sa création.

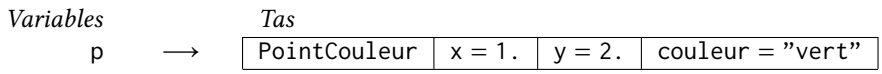


Durant la vie d'un objet, les seules choses modifiées sont les valeurs des attributs. La structure générale, et l'indication de la classe réelle, restent indéfiniment telles qu'elles ont été fixées à la création de l'objet.

Lorsque l'on utilise un objet `PointCouleur` comme un objet de type `Point`, par exemple après une déclaration

```
Point p = new PointCouleur(1., 2., "vert");
```

la représentation concrète de l'objet désigné par `p` est donc bien toujours la représentation d'un `PointCouleur`.



C'est ceci qui permet de réaliser des tests de type **`instanceof`**, ou encore de transtyper la variable `p` vers le type `PointCouleur`, lorsque la structure sous-jacente est effectivement une structure concrète de `PointCouleur`.

Redéfinition En guise de tremplin pour la suite : la classe `Object` contient elle-même des définitions de méthodes, que possèdent donc toutes les classes étendant `Object`, c'est-à-dire toutes les classes. Parmi ces méthodes, on a :

```
public String toString() { ... }
public boolean equals(Object other) { ... }
```

et on peut facilement observer l'effet de `toString` : cette méthode renvoie une chaîne de caractères indiquant la classe de l'objet ainsi que son adresse mémoire, et est utilisée par les fonctions comme `System.out.println`.

```
PointCouleur@5ca881b5
```

En revanche, lorsque l'on définit une méthode `toString` dans une nouvelle classe, cette nouvelle méthode vient prendre la place (pour cette classe seulement) de la méthode `toString` héritée de `Object`. On parle ici de *redéfinition* de méthode (*override*).

Fiche 8 : classes abstraites, et retour sur la visibilité

IPO – Introduction à la programmation objet

Thibaut Balabonski @ Université Paris-Saclay
`https://www.lri.fr/~blsk/IPO/`

V1, automne 2022

Rappel de deux éléments des fiches précédentes.

- Le mécanisme d'*extension* permet de définir une nouvelle classe héritant du contenu d'une classe pré-existante. Avantage : on évite d'écrire plusieurs fois le même code.
- Une *interface* définit un type pouvant être réalisé par plusieurs classes concrètes. Avantage : on peut grouper des éléments de ces différentes classes concrètes dans une même structure, ou les faire traiter par une même méthode.

Les *classes abstraites* combinent les deux aspects précédents.

Classe abstraite Une classe abstraite est une classe dans laquelle certaines méthodes sont seulement *déclarées*, et pas définies. Une telle classe contient, comme une classe ordinaire : des déclarations d'attributs et des définitions de constructeurs et de méthodes. Elle peut contenir, en plus, des *méthodes abstraites*, c'est-à-dire de simples déclarations de méthodes similaires à celles présentes dans les interfaces. Une déclaration de méthode indique le type de retour, les types des arguments, et termine par un point-virgule sans code ni accolades. Chaque méthode ou classe asbtraite est accompagnée du mot-clé **`abstract`**.

```
abstract class Forme { // classe générale pour des formes géométriques
  private double cx, cy;  // coordonnées du centre de la forme
  public Forme(double cx, double cy) { this.cx = cx; this.cy = cy; }
  public void translation(double dx, double dy) { cx += dx; cy += dy; }

  public abstract void agrandissement(double f);  // agrandissement d'un facteur f
                                                  // le code dépend de la forme
}
```

Note : contrairement aux méthodes déclarées dans les interfaces, les méthodes abstraites ne sont pas systématiquement publiques. Il faut donc indiquer leur visibilité explicitement, comme pour les attributs et les méthodes concrètes.

Même si une classe abstraite définit un constructeur, il est impossible d'en construire une instance avec **`new`**. Si l'on tente de le faire, le compilateur Java déclenche une erreur avant même que l'on puisse tenter d'exécuter le programme.

```
Forme f = new Forme(1., 2.);
              ^^^^^
java: Forme is abstract; cannot be instantiated
```

En effet, l'existence d'un tel objet `f` serait problématique, puisqu'il n'aurait aucun code à exécuter au moment d'un appel `f.agrandissement(2.);`.

Concrétisation On concrétise une classe abstraite `A` en définissant une nouvelle classe `C` qui :

- étend la classe abstraite `A`,
- fournit des définitions pour les méthodes abstraites de `A`.

Il s'agit d'une extension comme les autres : tous les attributs et toutes les méthodes définies par `A` sont bien conservés. On ajoute seulement :

- les déclarations des attributs supplémentaires,
- les définitions des constructeurs,
- les définitions des méthodes abstraites,
- (et éventuellement les définitions de nouvelles méthodes non mentionnées dans `A`).

Les constructeurs de la classe concrète `C` peuvent faire appel aux constructeurs de la classe abstraite `A` via **`super`**.

```
class Cercle extends Forme { // un cas concret de forme
  private double r;  // rayon
  public Cercle(double cx, double cy, double r) {
    super(cx, cy);  // ne pas oublier d'initialiser les coordonnées du centre
    this.r = r;
  }
  public void agrandissement(double f) { this.r *= f; }  // ici, on sait quoi faire
}
```

Comme avec les interfaces, on peut ensuite créer un objet ayant le type de la classe abstraite à l'aide du constructeur de la classe concrète.

```
Forme f = new Cercle(1., 2., 1.);
```

Avantage par rapport aux interfaces : le code de la classe abstraite est écrit une fois pour toutes, et partagé par toutes les classes concrètes. On peut ainsi multiplier les formes, en n'écrivant que la partie du code qui est spécifique à chacune.

```
class Rectangle extends Forme { // un autre cas concret
  private double h, l;  // hauteur, largeur
  public Rectangle(double cx, double cy, double h, double l) {
    super(cx, cy); this.h = h; this.l = l;
  }
  public void agrandissement(double f) { this.h *= f; this.l *= f; }
}
```

Les instances de différentes classes concrètes peuvent ainsi être regroupées au sein d'un même ensemble de formes.

```
ArrayList<Forme> arr = new ArrayList<>();
arr.add(new Cercle(1., 2., 1.));
arr.add(new Rectangle(0., 1., 2., 2.));
```

Toute classe possédant une méthode abstraite doit elle-même être déclarée abstraite, sans quoi le compilateur Java produit une erreur.

```
class Forme {
^
Class 'Forme' must either be declared abstract or implement abstract method
'agrandissement' in 'Forme'
```

Cela vaut y compris pour une classe `C` étendant une classe abstraite `A`, mais sans fournir de définitions pour certaines des méthodes abstraites de `A`.

```
abstract class Inclinable extends Forme {
  private double angle;
  public Inclinable(double cx, double cy, double a) {
    super(cx, cy); this.angle = a;
  }
  public void rotation(double a) { this.angle += a; }
  // pas de définition pour agrandissement
}
class Rectangle extends Inclinable { /* code déjà vu + constructeur à adapter */ }
```

Techniquement, rien n'interdit de déclarer **`abstract`** une classe dont toutes les méthodes seraient définies. En revanche, cela empêcherait d'utiliser la construction d'une instance avec **`new`**.

Retour sur les visibilités Les différents membres d'une classe (attributs, constructeurs, méthodes) sont toujours associés à une *visibilité*, qui spécifie *qui* peut accéder à ce membre ou l'utiliser. On a déjà vu :

- **`public`**, souvent utilisé pour les méthodes et constructeurs, qui ne donne aucune restriction d'accès,
- **`private`**, souvent utilisé pour les attributs, qui ne permet l'accès à un membre donné que dans le code de la classe elle-même.

Il existe deux niveaux de visibilité supplémentaires, intermédiaires entre **`public`** et **`private`**.

- Sans indication de visibilité on se trouve dans le mode par défaut, nommé *package-private*, qui rend le membre visible dans l'ensemble du *package* où est définie la classe (en gros : le répertoire contenant le fichier, on reviendra sur cette notion).
- Avec l'indication **`protected`**, le membre est visible dans l'ensemble du *package*, ainsi que dans toutes les sous-classes, directes ou indirectes.

Ces niveaux intermédiaires sont légitimement utilisables lorsque l'on développe plusieurs classes en forte interaction.

```
abstract class Forme {
  protected double cx, cy;  // les formes concrètes peuvent avoir besoin
                            // de connaître les coordonnées du centre
  ...
}

class Cercle extends Forme {
  ...
  public boolean contains(x, y) {
    return (x-cx)*(x-cx) + (y-cy)*(y-cy) <= r*r;  // en effet
  }
}
```

Constantes Lors de l'introduction d'une variable en Java, on peut déclarer que la valeur de celle-ci n'est pas modifiable avec le mot-clé **`final`**. Avec une telle déclaration, on peut réaliser une unique affectation à la variable (bonne pratique : la coupler à la déclaration). Les éventuelles affectations suivantes sont rejetées par le compilateur.

```
final int x = 1;
x = 2;
^
Variable 'x' might already have been assigned to
```

Cette qualification s'applique de même aux attributs des objets. Dans ce cas, l'affectation initiale peut être soit couplée à la déclaration, soit réalisée par le constructeur.

```
class Point {
  public final double x, y;
  public Point(double x, double y) {
    this.x = x;
    this.y = y;
  }
}
```

Toute tentative d'affectation ailleurs que dans le constructeur sera rejetée, car toute méthode autre que le constructeur est susceptible d'être appelée plusieurs fois dans la vie d'un objet.

```
  public void setX(double x) { this.x = x; }
                               ^^^^^^
  Cannot assign a value to final variable 'x'
```

Le compilateur fait des vérifications assez précises : un attribut **`final`** doit être initialisé, soit à sa déclaration, soit *dans chaque constructeur*, mais pas à ces deux endroits à la fois.

Note : un attribut **`final`** étant immuable, on élimine une des raisons poussant à déclarer les attributs **`private`** (personne ne pourra briser par inadvertance les invariants d'une classe en modifiant cet attribut, puisque la modification est impossible). Sauf à vouloir empêcher les utilisateurs de tenir compte de la valeur d'un tel attribut, on peut donc le rendre **`public`**, comme dans l'exemple ci-dessus. Fonctionnellement, cela équivaut à avoir des *getters* mais pas de *setters*, mais sans besoin d'alourdir le code avec des appels de méthode pour accéder à ces valeurs.

```
Point p = new Point(1., 2.);
double n2 = p.x * p.x + p.y * p.y;
```

Dans le cas d'un attribut **`static`**, qui est lié à la classe elle-même et pas à une instance produite par le constructeur, la seule possibilité est d'initialiser à la déclaration.

```
public static final int REPONSE = 42;
```

Ce dernier exemple est une bonne manière d'introduire une constante globale dans un programme.

Le mot-clé **`final`** s'applique aussi à des méthodes et à des classes, pour contrôler les extensions et redéfinitions :

— une méthode **`final`** ne peut pas être redéfinie dans une sous-classe,
— une classe **`final`** ne peut pas être étendue.

Fiche 9 : affichage et interaction

IPO – Introduction à la programmation objet

Thibaut Balabonski @ Université Paris-Saclay
https://www.lri.fr/~blsk/IPO/

V1, automne 2022

Les bibliothèques Swing et AWT permettent de bâtir en Java une interface graphique interactive. Premier tour d'horizon.

Fenêtre graphique Une application graphique est matérialisée une *fenêtre*, s'ouvrant dans l'interface graphique de votre système d'exploitation, généralement constituée d'une barre de titre avec quelques boutons (fermeture, agrandissement, réduction) et d'un cadre. La classe `JFrame` matérialise une telle fenêtre en Java. L'aspect extérieur de la fenêtre (forme du cadre, bouton, barre de titre) est lié au système d'exploitation. On ne s'intéressera ici qu'au contenu.

Le constructeur d'une `JFrame` prend en paramètre la chaîne à afficher dans la barre de titre.

```
public class Bulles {
  public static void main(String[] args) {
    JFrame frame = new JFrame("Bulles");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    ...
```

Le contenu principal de la fenêtre est regroupé dans un *conteneur* que l'on peut récupérer avec la méthode `getContentPane()`. Ce contenu est consituté de différents éléments graphiques : boutons, zones de texte, affichage...

Un élément graphique multi-fonctions : JPanel La classe `JPanel` définit l'un des éléments pouvant être affiché dans une fenêtre, qui est pratique à la fois pour dessiner directement et pour regrouper et organiser d'autres éléments. On peut inclure un `JPanel` dans une fenêtre en l'ajoutant à son `contentPane`.

```
    JPanel panel = ...
    frame.getContentPane().add(panel);
```

On termine ensuite la mise en place de la fenêtre en la dimensionnant autour de son contenu (`pack()`), et en la déclarant visible.

```
    frame.pack();
    frame.setVisible(true);
```

Il n'y a plus qu'à inclure du contenu dans notre `JPanel` pour l'afficher dans la fenêtre.

Une technique simple pour réaliser un affichage arbitraire consiste à définir une nouvelle classe étendant `JPanel`, en y incluant les éléments qui nous intéressent.

```
class Aquarium extends JPanel {
  private ArrayList<Bulle> bulles;
  ...
```

Parmi les premiers éléments que l'on peut configurer dès la construction d'une zone d'affichage : la taille de la zone, et sa couleur de fond.

```
  public Aquarium(int l, int h) {
    this.bulles = new ArrayList<>();
    setPreferredSize(new Dimension(l, h));
    setBackground(new Color(161, 202, 241));
  }
```

Les éléments graphique comme `JPanel` possèdent une méthode `paintComponent` définissant leur rendu à l'écran. Cette méthode prend en paramètre un objet `g` de la classe `Graphics`, que l'on peut considérer comme un « pinceau » :

— ses attributs mémorisent la configuration de dessin actuelle (notamment : la couleur sélectionnée),
— ses méthodes dessinent des traits ou des formes.

Pour définir l'affichage d'un panneau personnalisé, on redéfinit cette méthode. On peut alors se servir des différentes méthodes de la class `Graphics` pour réaliser des tracés arbitraites.

```
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(new Color(255, 255, 255, 127));
    for (Bulle b : bulles) {
      g.fillOval(b.x - b.r, b.y - b.r, 2*b.r, 2*b.r);
    }
  }
```

Note : pour ne pas perdre le comportement général d'affichage de la classe mère `JPanel`, on commence par faire un appel à sa méthode d'affichage d'origine avec **`super`**`.paintComponent`. Ceci va notamment mettre en place la couleur de fond, au-dessus de laquelle seront tracés les éléments suivants.

Dans le code principal, on peut alors utiliser cette nouvelle classe pour incarner le `JPanel` à afficher.

```
    JPanel panel = new Aquarium(800, 600);
```

Interaction avec la souris Les interactions en temps réel en Java sont basées sur un système d'*événements* et de *notifications*. Certains objets sont des *sources* d'événements, et d'autres sont des *récepteurs*, qui vont réagir aux événements des sources. Chaque objet récepteur s'inscrit auprès de la source dont il doit suivre les événements. En retour, une source diffuse auprès des récepteurs inscrits sur sa liste les informations de chaque nouvel événement.

— Un chronomètre (classe `Timer`) génère un tic d'horloge à un intervalle de temps périodique. Un récepteur inscrit auprès d'un chronomètre recevra donc une notification et pourra agir à chacun de ces tics périodiques.
— Une fenêtre graphique (classe `JFrame`) génère un événement à chaque frappe d'une touche ou action de la souris. Un récepteur inscrit auprès de la fenêtre pourra donc réagir à de tels événements.

Plusieurs interfaces décrivent les récepteurs capables de réagir à différentes sortes d'événements. Notamment :

— `MouseListener` pour les récepteurs réagissant aux actions de la souris,
— `KeyListener` pour les récepteurs réagissant à l'utilisation du clavier,
— `ActionListener` pour les récepteurs réagissant à un événement générique, comme l'utilisation d'un bouton d'une interface ou un tic d'horloge.

. La classe d'un récepteur doit donc implémenter l'interface correspondante

```
class Aquarium extends JPanel implements MouseListener {
  ...
```

et il faut, à un moment du code principal ou de la construction du système, inscrire chaque objet récepteur auprès de sa ou ses sources, à l'aide de méthodes dédiées de ces dernières.

```
    /* fin de Bulles.main */
    frame.addMouseListener(panel);
  }
```

À chaque nouvel événement, la source enverra des notifications à chaque récepteur ainsi inscrit.

Les « notifications » envoyées par une source à ses récepteurs sont matérialisées par l'appel de méthodes du récepteur. Plus précisément : par les méthodes déclarées dans l'interface du récepteur.

```
/* extrait de la bibliothèque java.awt */
interface MouseListener {
  /* Réaction aux événements de la source : */
  void mouseClicked(MouseEvent e);  /* clic */
  void mouseEntered(MouseEvent e);  /* arrivée du curseur dans la zone */
  void mouseExited(MouseEvent e);   /* sortie du curseur de la zone */
  void mousePressed(MouseEvent e);  /* bouton appuyé */
  void mouseReleased(MouseEvent e); /* bouton relâché */
}
```

Dans la classe d'un récepteur, on définit les méthodes correspondant aux différents événement auxquels on souhaite réagir. Ces méthodes de réaction prennent systématiquement un unique paramètre, de type adapté, décrivant précisément l'événement (par exemple : quelle touche du clavier a été utilisée, ou la position précisé du curseur de la souris). *Note :* si la réaction à un événement implique une modification des éléments à afficher, on peut rafraîchir l'affichage avec la méthode `repaint` de `JPanel`.

```
  public void mouseClicked(MouseEvent e) {
    bulles.add(new Bulle(e.getX(), e.getY(), 40));
    this.repaint();
  }
```

Rappel : pour concrétiser une interface, il faut fournir des définitions à toutes les méthodes de l'interface. Si l'on ne souhaite pas réagir à certains des événements prévus par l'interface, il suffit de définir la méthode correspondante avec un code vide.

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

Programme complet Tous les éléments précédents peuvent être regroupés dans un unique fichier `Bulles.java` (ou séparés en autant de fichiers que de classes).

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.MouseListener;
import java.awt.event.MouseEvent;
import java.util.ArrayList;

public class Bulles {
  public static void main(String[] args) {
    JFrame frame = new JFrame("Bulles");
    Aquarium panel = new Aquarium(800, 600);
    frame.getContentPane().add(panel);
    frame.addMouseListener(panel);
    frame.pack();
    frame.setVisible(true);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }
}

class Aquarium extends JPanel implements MouseListener {
  private ArrayList<Bulle> bulles;
  public Aquarium(int l, int h) {
    this.bulles = new ArrayList<>();
    setBackground(new Color(161, 202, 241));
    setPreferredSize(new Dimension(l, h));
  }

  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(new Color(255, 255, 255, 127));
    for (Bulle b : bulles) {
      g.fillOval(b.x-b.r, b.y-b.r, 2*b.r, 2*b.r);
    }
  }

  public void mouseClicked(MouseEvent e) {
    bulles.add(new Bulle(e.getX(), e.getY(), 40));
    this.repaint();
  }

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

class Bulle {
  public final int x, y, r;
  public Bulle(int x, int y, int r) {
    this.x = x;
    this.y = y;
    this.r = r;
  }
}
```

Fiche 10 : affichage et interaction, bis

IPO – Introduction à la programmation objet

Thibaut Balabonski @ Université Paris-Saclay
`https://www.lri.fr/~blsk/IPO/`

V1, automne 2022

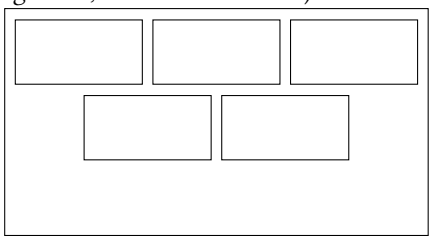
Rappel de la structure principale Une fenêtre d'une application graphique Java est matérialisée par un objet `JFrame`. Une telle fenêtre contient plusieurs éléments, dont notamment :

- un conteneur principal, avec les éléments à afficher (`contentPane`),
- une barre de menu, optionnelle,
- une « vitre » (`glassPane`), invisible par défaut mais recouvrant toute la fenêtre et pouvant servir à bloquer ce qu'elle recouvre.

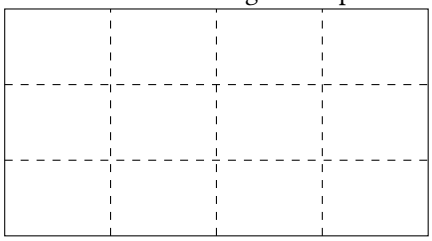
On ajoute au `contentPane` les éléments à afficher, par exemple des instances de `JPanel`.

Disposition des éléments Les conteneurs recevant des éléments peuvent être configurés pour disposer leur contenu d'une manière précise. Ceci vaut pour le `contentPane` d'une fenêtre aussi bien que pour chaque `JPanel`. Rappel : dans un cas comme dans l'autre, chaque nouvel élément doit être ajouté avec la méthode `add` du conteneur. La description d'une disposition est un objet de la classe `LayoutManager`. Java en propose plusieurs sous-classes concrètes prédéfinies. En voici trois exemples courants.

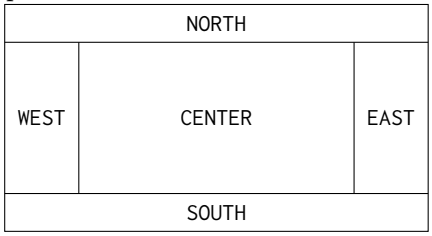
- `FlowLayout` dispose les éléments l'un à la suite de l'autre, dans l'ordre de leur ajout, en passant à la ligne lorsque la ligne courante est pleine, similairement à la manière dont on organiserait les mots d'un paragraphe. Les lignes sont rééquilibrées lorsque l'on modifie la taille du conteneur. On peut spécifier l'espace entre les éléments ainsi que leur alignement (par exemple : à gauche, à droite ou centrés).



- `GridLayout` dispose les éléments en une grille régulière. La taille des « cases » contenant les éléments est calculée en fonction de la taille globale du conteneur (et modifiée avec elle). Comme avec `FlowLayout`, les éléments sont placés dans les cases dans l'ordre des appels à `add`. On peut spécifier l'espace entre les cases, ainsi que le nombre de lignes *ou* le nombre de colonnes (on indique `0` pour ne pas spécifier l'un de ces paramètres, et si on donne des valeurs non nulles pour les deux, alors seul le nombre de lignes est pris en compte).



- `BorderLayout` dispose les éléments en fonction de cinq points de repère : le centre du conteneur et ses quatre bords. Les éléments sont dimensionnés en fonction de la position choisie et de la présence ou non d'éléments dans les zones voisines. On peut spécifier l'espace entre les éléments.



  Pour insérer un élément dans un tel conteneur, on donne en deuxième paramètre à la méthode `add` l'une des cinq constantes `CENTER`, `NORTH`, `WEST`, `SOUTH`, `EAST`. L'absence d'indication équivaut à `CENTER`.

Par défaut, le `contentPane` est configuré avec `BorderLayout`, et les `JPanel` avec `FlowLayout`. Pour changer la configuration, on utilise la méthode `setLayout`, qui prend en paramètre un objet de l'une des classes étendant `LayoutManager`.

```
panel.setLayout(new GridLayout(0, 4)); // 4 colonnes, nb de lignes selon nb d'éléments
```

Éléments pour l'affichage On a déjà mentionné la classe `JPanel`, qui est avant tout un conteneur pouvant regrouper d'autres éléments graphiques. Comme vu au cours précédent, on peut également étendre cette classe et spécialiser sa méthode d'affichage `paintComponent` pour y inclure des dessins arbitraires.

Pour afficher un court texte et/ou une image, on peut utiliser la classe `JLabel`. Le texte initial peut être donné en paramètre au constructeur, et mis à jour avec `setText`.

```
JLabel info = new JLabel("info");
...
info.setText(message);
```

Pour afficher un texte de plusieurs lignes, on a `JTextArea`. Par défaut, un texte de cette dernière forme est éditable par l'utilisateur, mais on peut désactiver cette possibilité.

```
JTextArea painVieuxFourneaux =
  new JTextArea("Sarmentine (farine de meule)\nFleurimeuline du Papé\nCâlinette");
painVieuxFourneaux.setEditable(false);
```

Boutons et actions Un élément interactif élémentaire est le bouton cliquable, donné par la classe `JButton`. Il peut afficher un court texte et/ou une image, à la manière d'un `JLabel`, et est supposé déclencher une action lorsqu'il reçoit un clic de l'utilisateur. Le déclenchement des actions d'un bouton suit le même principe que la réaction aux événements du clavier ou de la souris.

- Le bouton est une source d'événements, ici de type `ActionEvent`.
- D'autres objets peuvent s'inscrire comme récepteurs auprès du bouton pour recevoir une notification à chaque clic. On utilise pour cela la méthode **`void`** `addActionListener(ActionListener r)` du bouton.
- Les récepteurs doivent implémenter l'interface `ActionListener`, qui demande d'implémenter une unique méthode **`void`** `actionPerformed(ActionEvent e)`, qui sera appelée à chaque clic en guise de notification.

```
class Compteur implements ActionListener {
  private int compteur;

  public Compteur() {
    ...
    JButton b = new JButton("+1");
    panel.add(b);
    b.addActionListener(this);
    ...
  }

  public void actionPerformed(ActionEvent e) {
    this.compteur++;
  }
}
```

Ordinairement, une application contiendra plusieurs boutons utilisés pour déclencher des actions différentes. On peut imaginer regrouper ces différentes actions dans une seule méthode `actionPerformed` : il faudrait pour cela, lors de la réception d'une notification, consulter la source (`e.getSource()`) puis chercher le bouton correspondant.

```
class Compteur implements ActionListener {
  private int compteur;

  public Compteur() {
    ...
    JButton b1 = new JButton("+1");
    JButton b2 = new JButton("0");
    b1.addActionListener(this);
    b2.addActionListener(this);
    ...
  }

  public void actionPerformed(ActionEvent e) {
    if (e.getSource() == b1) { this.compteur++; }
    if (e.getSource() == b2) { this.compteur = 0; }
    ...
  }
}
```

Cette stratégie nécessite vite d'écrire une multitude de tests et multiplie les opportunités d'erreur. Une meilleure approche consiste à définir une « micro-classe » implémentant l'interface `ActionListener` pour chacun des boutons à créer, et à associer à chaque bouton une instance de la bonne classe (note : « micro-classe » n'est pas un terme technique officiel).

```
class Bouton1 implements ActionListener {
  public Bouton1(Compteur c) { ... }
  public void actionPerformed(ActionEvent e) {
    c.compteur++;
  }
}
class Bouton2 implements ActionListener {
  public Bouton2(Compteur c) { ... }
  public void actionPerformed(ActionEvent e) {
    c.compteur = 0;
  }
}

class Compteur {
  int compteur;
  public Compteur() {
    ...
    JButton b1 = new JButton("+1");
    JButton b2 = new JButton("0");
    b1.addActionListener(new Bouton1(this));
    b2.addActionListener(new Bouton2(this));
    ...
  }
}
```

Java propose une écriture (très) allégée pour ce processus, qui permet d'éviter de définir explicitement ces micro-classes à usage unique : on donne en paramètre à `addActionListener` une *lambda-expression*, similaire à une fonction Caml, qui décrit la méthode `actionPerformed` correspondante. On peut ainsi écrire « `e -> { ... }` » pour désigner « une instance d'une classe implémentant `ActionListener` avec la méthode **`void`** `actionPerformed(ActionEvent e) { ... }` » (et on ne donne pas de nom à la classe implémentant `ActionListener`, qui ne servira nulle part ailleurs).

```
class Compteur {
  private int compteur;

  public Compteur() {
    ...
    JButton b1 = new JButton("+1");
    JButton b2 = new JButton("0");
    b1.addActionListener(e -> { compteur++; });
    b2.addActionListener(e -> { compteur = 0; });
    ...
  }
}
```

Éléments interactifs Au-delà de `JButton`, la bibliothèque Swing propose une variété d'éléments interactifs : case à cocher (`JCheckBox`), champ de saisie de texte (`JTextField`), sélection dans une liste (`JComboBox`), qui sont encore des sources de `ActionEvent` et peuvent être traités de manière similaire (avec des méthodes spécifiques pour obtenir les informations sur ce qui a été entré ou sélectionné).

Un programme complet Combinons tous les éléments précédents pour écrire un programme interactif complet. On utilisera un certain nombre d'éléments des bibliothèques Java.

```
import javax.swing.*; // JFrame, JPanel, JLabel, JButton, JCombobox, JTextField
import java.awt.*;    // BorderLayout, FlowLayout, Color, Graphics
import java.awt.event.*; // MouseEvent, MouseListener, MouseMotionListener
import java.util.ArrayList;
import java.util.List;
```

Note : l'interface `ActionListener` et la classe `ActionEvent` n'apparaissent pas, car l'écriture allégée des actions ne les fera pas apparaître explicitement.

La fenêtre principale est composée de trois zones :

— une feuille centrale sur laquelle on peut cliquer pour dessiner,
— une barre d'outils en haut, permettant de configurer le pinceau,
— une barre d'information en bas.

La barre d'outils est connectée à la feuille pour en mettre à jour les paramètres, et la feuille et la barre d'information sont toutes deux à l'écoute des actions de la souris (du moins, tant que la souris survole la feuille).

```
public class Gribouille {
  public static void main(String[] args) {
    // Fabrication de la fenêtre elle-même
    JFrame frame = new JFrame("Gribouille");
    frame.getContentPane().setLayout(new BorderLayout()); // redondant avec défaut
    frame.setSize(800, 600); // définition explicite de la taille (largeur, hauteur)
    frame.setVisible(true);
    frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);

    // Création et placement des éléments (définitions ci-dessous)
    Feuille feuille = new Feuille();
    Info info = new Info();
    Outils outils  = new Outils(feuille);
    frame.add(feuille, BorderLayout.CENTER);
    frame.add(outils, BorderLayout.NORTH);
    frame.add(info, BorderLayout.SOUTH);

    // Préparation à la réception des événements
    feuille.addMouseListener(feuille);
    feuille.addMouseMotionListener(info);
  }
}
```

La feuille enregistre un ensemble de bulles à afficher, ainsi que la couleur et le rayon à donner à la prochaine bulle créée. Note : comme on écoute les événements de souris émis par la feuille, les coordonnées apportées par le `MouseEvent` sont relatives à la feuille et non à la fenêtre entière.

```
class Feuille extends JPanel implements MouseListener {
  private List<Bulle> bulles;
  Color couleur = Color.BLACK; // couleur par défaut
  int rayon = 10;              // rayon par défaut

  public Feuille() {
    this.bulles = new ArrayList<>();
    setBackground(Color.WHITE);                      // la feuille est blanche
    setCursor(new Cursor(Cursor.CROSSHAIR_CURSOR)); // curseur croix : plus précis
  }

  // Réaction aux clics de souris sur la feuille
  public void mouseClicked(MouseEvent e) {
    bulles.add(new Bulle(e.getX(), e.getY(), rayon, couleur));
    repaint();
  }
  public void mousePressed(MouseEvent e) {}  // Rien de particulier dans les autres cas
  public void mouseReleased(MouseEvent e) {}
  public void mouseEntered(MouseEvent e) {}
  public void mouseExited(MouseEvent e) {}

  // Servira plus bas : méthode pour tout gommer
  public void effacer() { bulles.clear(); repaint(); }

  // Affichage du dessin
  public void paintComponent(Graphics g) {
    super.paintComponent(g); // rappel : on commence en général par cela
    for (Bulle b : bulles) { // affichage du dessin proprement dit
      g.setColor(b.c);
      g.fillOval(b.x-b.r, b.y-b.r, 2*b.r, 2*b.r);
    }
  }
}
```

La barre d'outils contient trois éléments principaux :

- Un sélecteur de couleurs basé sur un menu déroulant (`JComboBox` contenant une liste de couleurs).
- Un champ de texte (`JTextField`) pour définir le rayon des bulles à créer.
- Un bouton pour tout effacer (`JButton`).

Les deux premiers éléments sont précédés d'une étiquette (`JLabel`) explicitant à l'utilisateur leur signification. Chacun des trois éléments produit des `ActionEvent`, et on spécifie l'action correspondante à l'aide d'une lambda-expression.

```
class Outils extends JPanel {
  private Feuille feuille; // référence vers la feuille pour la configurer
  public Outils(Feuille f) {
    this.feuille = f;
    setLayout(new FlowLayout()); // redondant avec défaut

    // Menu déroulant des couleurs
    this.add(new JLabel("Couleur"));
    Color[] listeCouleurs = { Color.BLACK, Color.BLUE, Color.GREEN,
                              Color.GRAY, Color.PINK, Color.RED };
    JComboBox<Color> couleurs = new JComboBox<>(listeCouleurs);
    couleurs.addActionListener(e -> { feuille.couleur =
                                      (Color)couleurs.getSelectedItem(); }
                              );
    this.add(couleurs);

    // Champ texte pour la sélection du rayon
    this.add(new JLabel("Rayon"));
    JTextField rayon = new JTextField("10");
    rayon.addActionListener(e -> { feuille.rayon = Integer.parseInt(rayon.getText()); });
    this.add(rayon);

    // Tout effacer
    JButton efface = new JButton("Effacer");
    efface.addActionListener(e -> { feuille.effacer(); });
    this.add(efface);
  }
}
```

Bande d'information : position du curseur à l'intérieur de la feuille, mis à jour en temps réel.

```
class Info extends JPanel implements MouseMotionListener {
  private JLabel infoPosition;
  public Info() {
    infoPosition = new JLabel("no info");
    this.add(infoPosition);
  }

  public void mouseMoved(MouseEvent e) {
    infoPosition.setText("x: " + e.getX() + ", y: " + e.getY());
  }
  public void mouseDragged(MouseEvent e) {}
}
```

Bulles.

```
class Bulle {
  final int x, y, r;
  final Color c;
  public Bulle(int x, int y, int r, Color c) {
    this.x = x; this.y = y; this.r = r;  this.c = c;
  }
}
```

Post-mortem du bug du 28 novembre Lors de la séance, j'ai voulu gagner du temps en reproduisant directement avec `contentPane` ce que je faisais habituellement dans un `JPanel` : insérer plusieurs éléments, et observer le comportement de `FlowLayout`. Ce faisant, j'avais oublié que `contentPane` avait un comportement par défaut différent, à savoir BorderLayout. Sans indication de placement, tous les éléments ont été placés en `BorderLayout.CENTER`, chacun masquant le précédent. À la fin, n'était donc visible que le dernier. Cela tient finalement à peu de choses. Toutes mes excuses pour ce cafouillage.

IPO – TP-projet : des moutons dans le donjon

http://www.lri.fr/~blsk/IPO/

Un berger et son troupeau se sont abrités de la pluie dans un vieux château. Se rendant compte que le donjon abritait des loups, le berger cherche à faire ressortir ses moutons. Le voilà fort affairé : il doit tout à la fois guider les moutons vers la sortie et s'interposer entre eux et les loups.

Objectif du TP Ce TP-projet est à réaliser en binôme. Vous y reprenez l'ensemble des éléments réalisés lors du TP 9 (classes abstraites), et les intégrez dans un jeu à un joueur en temps réel. L'ensemble formé par ces deux TP sera à compléter pour le 4 décembre (23h59) et donnera lieu à une soutenance la semaine du 5 septembre. Le tout formera votre note de contrôle continu. L'ensemble de votre projet doit être versionné avec `Git`, et vous devez donner à votre enseignant l'accès à votre dépôt.

Vous trouverez sur la page du cours une archive `.jar` contenant une version de démonstration minimale de ce jeu. Pour lancer cette démonstration, vous devez également télécharger le fichier `laby2.txt` et le placer dans le même répertoire que le `.jar`. Puis double-clic sur `moutons.jar`, ou dans un terminal entrez la ligne `java -jar moutons.jar &`.

Description Le donjon est matérialisé par la classe `Terrain` et ses différentes sortes de `Cases`. Les moutons sont représentés par la classe `Personnage` et les loups par la classe `Monstre`. Tous ces éléments peuvent être repris intégralement, sans modification par rapport à ce qui était demandé au TP9.

Le berger est représenté par un nouveau sous-type d'`Entite`. Comme les autres entités, il peut se trouver sur n'importe quelle case traversable, et dispose d'une résistance. Comme les autres entités, il est un obstacle au déplacement des moutons et des loups :

— un mouton se trouvant face au berger changera de direction,

— un loup se trouvant face au berger l'attaquera, faisant décroître sa résistance.

Le jeu présente au joueur une vue de l'ensemble du donjon, mise à jour après chaque `tour` (10 fois par seconde). Le joueur y déplace le berger en temps réel à l'aide des flèches du clavier.

Travail à réaliser Vous trouverez sur la page du cours un fichier principal `Donjon.java` (complet), un squelette d'interface graphique `FenetreGraphique.java` (à compléter), et un nouvel exemple de description de terrain (`laby2.txt`), à combiner avec vos fichiers du TP9. Voici une liste des tâches à réaliser, dans l'ordre.

1. Créer un dépôt `Git` par binôme pour gérer votre projet, et placez-y tous les fichiers (instructions détaillées page suivante).
2. Compléter dans la classe `FenetreGraphique` les méthodes d'affichage, de sorte à pouvoir observer le comportement de votre jeu, sans interaction.
3. Créer une classe `Joueur` étendant `Entite`, pour représenter le berger. Dans le constructeur de la classe `Terrain`, étendre le décodage du fichier d'entrée pour qu'il interprète un `H` comme la position de départ du joueur (vous pouvez maintenant utiliser le fichier `laby2.txt`).
4. Ajouter à la définition de la classe `FenetreGraphique` l'annotation **`implements`** `KeyListener`, et étendre cette classe avec les méthodes nécessaires, de sorte que les flèches du clavier permettent de déplacer le joueur. *Note :* l'interface `KeyListener` a un fonctionnement similaire à `MouseListener` (mais c'est à vous de trouver le détail des méthodes à implémenter !)
5. Ajouter au berger la possibilité de sortir lui aussi, en appuyant sur la touche « espace » lorsqu'il se trouve sur une `Sortie`.
6. Ajouter à la classe `Jeu` une méthode testant si la partie est finie :
   — le jeu s'arrête lorsque le berger sort, avec un score égal au nombre de moutons sauvés,
   — le jeu s'arrête également lorsque le berger est mangé, avec un score divisé par deux.
7. Ajouter, à volonté, de nouveaux éléments pour enrichir le jeu.

Instructions pour l'initialisation de `Git`

Création du projet lui-même, à faire par l'un des membres du binôme.

1. Allez sur l'interface web du `gitlab` de l'université :
   `https://gitlab.dsi.universite-paris-saclay.fr/`
   et connectez-vous (avec vos identifiants habituels de l'université).
2. Cliquez en haut à droite : *new project*, puis sélectionnez *create blank project*.
3. Donnez un nom à votre projet. À la ligne *project URL*, assurez-vous que votre nom apparaît, ou allez le sélectionner en cliquant sur *pick a group or namespace*.
4. Assurez-vous que le niveau de visibilité *private* est sélectionné, puis confirmez la création.

Sélection des autres participants, à réaliser dans l'interface web du projet.

1. Dans le menu, sélectionnez *project information*, puis *members*.
2. Cliquez sur *invite member* et intégrez le deuxième membre du binôme, avec le rôle *maintainer*.
3. Recommencez et intégrez votre encadrant(e) de TP, avec un rôle *reporter* (ou supérieur), puis de même avec `thibaut.balabonski`.

Note : vous ne pouvez inviter que quelqu'un qui s'est déjà connecté à ce `gitlab` un jour. Si vous ne trouvez pas la personne dans la liste, demandez-lui de se connecter.

Instructions pour connecter `Git` et `IntelliJ IDEA`

Import du projet dans `IntelliJ`, à faire par chaque membre du groupe et sur chaque ordinateur utilisé.

1. Dans `IntelliJ`, allez sélectionner dans le menu *file*, *new*, puis *project from version control*.
2. Vérifiez que `Git` est sélectionné dans la ligne *version control*, puis indiquez l'URL du projet (l'adresse de la page principale de votre projet sous `gitlab`, que vous pouvez aller copier depuis la barre d'adresse). Vous pouvez aussi depuis ce menu sélectionner le dossier dans lequel se trouvera le projet.
3. Entrez vos identifiants pour autoriser `IntelliJ` à se connecter à votre compte `gitlab`, puis validez.

Initialisation des fichiers, à faire par l'un des membres du groupe.

1. Dans le projet `IntelliJ`, faites un clic droit *new*, *directory*, et nommez `src` le dossier créé.
2. Faites un clic droit sur `src`, puis sélectionnez *mark directory as*, et *sources root*.
3. Ajoutez dans le dossier `src` vos fichiers `.java`. Si l'interface vous propose des les ajouter à `Git`, acceptez.
4. Poussez les nouveaux fichiers sur le dépôt `Git` : dans le menu *git*, allez chercher les actions *commit* et *push* (l'action *commit* ouvrira dans l'interface une fenêtre où vous devez écrire un message décrivant le contenu de la modification que vous enregistrez).

Instructions pour travailler une fois `Git` en place

À réaliser par chaque personne travaillant sur le projet.

1. Au début de chaque session de travail, récupérez les fichiers du dépôt partagé pour mettre à jour vos fichiers locaux avec `pull`. Pour cela : cliquez dans le menu *git*, puis *pull*. Recommandation : c'est généralement une bonne idée, dans *modify options*, d'allez sélectionner `--rebase`.
2. Après chaque modification significative, enregistrez (localement) votre travail avec `commit`. Dans le menu *git*, sélectionner *commit*, et entrer un message décrivant la modification.
3. À la fin de chaque séance (voire plus souvent), publiez vos *commits* sur le dépôt partagé avec `push`. Dans le menu *git*, sélectionner *push*, et valider.

Pour éviter les interférences entre les modifications faites par plusieurs personnes travaillant en parallèle sur le même projet, il vaut mieux souvent publier ses modifications (`commit`/`push`) et récupérer les modifications des partenaires (`pull`).

Note : l'historique des *commit*s est daté, et permettra à votre encadrant d'apprécier la régularité de votre travail.

IPO – TP 1 : syntaxe et rappels impératifs

http://www.lri.fr/~blsk/IPO/

Mise en route

1. Récupérer le fichier `HelloWorld.java` sur la page du cours,
2. compiler le fichier en utilisant : `javac HelloWorld.java` dans un terminal, en vous plaçant dans le répertoire où vous avez mis le fichier,
3. exécuter le programme avec : `java HelloWorld` dans un terminal,
4. ouvrir le fichier avec un éditeur pour modifier l'affichage (lui faire afficher votre nom par exemple), puis compiler et exécuter.

Vous trouverez également sur la page un squelette de programme utilisable pour les exercices suivants (pensez à changer le nom du fichier et le nom du programme), ainsi qu'un fichier vous donnant des exemples de saisies clavier.

5. Consulter le fichier `ScannerDemo.java` sur la page du cours,
6. modifier le programme `HelloWorld` pour qu'il demande son nom à l'utilisateur avant de le saluer.

Exercice 1 – Expressions

1. En utilisant la formule $C = \frac{5}{9}(F-32)$, écrire un programme `Degre` qui lit une température exprimée en degrés Fahrenheit et affiche sa valeur en degrés Celsius.
   Exemples d'exécution du programme

```
$java Degres
Donnez une température en Fahrenheit : 0.0
Cette température équivaut à -17.8 degrés Celsius
$java Degres
Donnez une température en Fahrenheit : 60.0
Cette température équivaut a 15.6 degrés Celsius
```

2. Écrire un programme `Hjms` qui, pour un nombre de secondes donné, calcule et affiche son équivalent en nombre de jours, d'heures, de minutes et de secondes.
   Exemples d'exécution du programme

```
$java Hjms
Donnez une durée en secondes : 3621
Cette durée équivaut à 0 jour 1 heure 0 minute 21 secondes
$java Hjms
Donnez une durée en secondes : 180181
Cette durée équivaut à 2 jours 2 heures 3 minutes 1 seconde
```

Exercice 2 – Instructions conditionnelles

1. Écrire un programme `Couronne` qui, pour un point P du plan de coordonnées x et y demandées à l'utilisateur, détermine si ce point se trouve ou non à l'intérieur de la couronne de centre $(0,0)$ et définie par un rayon extérieur r_{ext} et un rayon intérieur r_{int}.
   Exemple d'exécution du programme

```
$java Couronne
Rayon extérieur : 14
Rayon intérieur : 10
Donnez un point x : 12 y : 0
Ce point est dans la couronne.
```

2. Écrire un programme `TroisNombres` qui lit trois nombres au clavier, les classe dans l'ordre croissant et les affiche du plus petit au plus grand.
Exemple d'exécution du programme

```
$java TroisNombres
1er nombre : 14
2e nombre : 10
3e nombre : 17
Les nombres dans l'ordre croissant : 10 14 17
```

Exercice 3 – Boucles

1. Écrire un programme `Triangle` qui affiche un motif triangulaire dont la taille est fixée par une valeur donnée au clavier.
Exemple d'exécution du programme

```
$java Triangle
Donnez la taille du motif : 7
*
**
***
****
*****
******
*******
```

   (a) Écrire une première version en n'utilisant que des boucles **`for`**,

   (b) puis une deuxième version en n'utilisant que des boucles **`while`**.

2. Écrire un programme `Pyramide` qui affiche un motif pyramidal dont la taille est fixée par une valeur donnée au clavier. Utilisez les boucles qui semblent les mieux adaptées.
Exemple d'exécution du programme

```
$java Pyramide
Donnez la taille du motif : 7
      *
     ***
    *****
   *******
  *********
 ***********
*************
```

Exercice 4 – Suite de Fibonacci La suite de Fibonacci est définie par la formule de récurrence suivante :

$$\begin{cases} u_0 &= 0 \\ u_1 &= 1 \\ u_n &= u_{n-1} + u_{n-2} \qquad \text{si } n \geq 2 \end{cases}$$

1. Écrire un programme `Fibo1` qui permet de calculer le n^{e} terme de la suite de Fibonacci, n étant fixé par l'utilisateur.
2. Écrire un programme `Fibo2` qui permet d'obtenir la valeur et le rang du premier terme de cette suite supérieur à une valeur limite donnée par l'utilisateur.

IPO – TP 2 : manipulation de classes

https://www.lri.fr/~blsk/IPO/

1. Vérification d'adresse IP. Une adresse IP est définie par quatre nombres compris entre 0 et 255, séparés par des points. Par exemple : 212.85.150.134. L'objectif de cet exercice est d'écrire une méthode

```
public static boolean checkIP(String ip) { ... }
```

qui vérifie si la chaîne de caractères donnée en argument définit bien une adresse IP. Pour cela il faut vérifier que :

- elle comporte 3 points ;
- elle commence et se termine par un nombre (.123. n'est pas une adresse IP) ;
- les caractères situés entre les points sont des chiffres compris entre 0 et 255 ;
- le premier nombre ne peut pas être 0.

Récupérer le fichier `IP.java` sur la page du cours. Il contient une méthode **`void`** `testCheckIP()` testant différents cas de figure. Votre fonction `checkIP` doit valider tous ces tests.

On pourra utiliser les méthodes suivantes :

- **`int`** `Integer.parseInt(String s)` pour convertir une chaîne de caractères en nombre (si et seulement si la chaîne de caractères correspond bien à un nombre) ;
- **`boolean`** `isInteger(String s)` pour vérifier si une chaîne de caractères représente un nombre (ce code sera expliqué lors du cours sur les exceptions) :

```
public static boolean isInteger(String s) {
    try {
      Integer.parseInt(s);
      return true;
    } catch (NumberFormatException nfe) {
      return false;
    }
}
```

- `String[] split(String sep)` (de la classe `String`) pour découper une chaîne de caractères en fonction d'un séparateur `sep`. Voci un exemple d'utilisation de cette méthode.

```
public class ExempleSplit {
  public static void main(String[] a) {
    String s = "Bonjour.les.amis";
    // Attention aux anti-slashs devant le point !
    String separateur = "\\.";
    String[] mot = s.split(separateur);
    for (String m : mot) {
      System.out.println(m);
    }
  }
}
```

Ce programme affiche :

```
Bonjour
les
amis
```

2. Filtre par langue. Dans cet exercice, nous allons trier des textes dans différentes langues, en utilisant la classe `java.util.Locale`. Les objets de cette classe servent à désigner une région ou une langue, dans ses différentes variantes possibles. Nous nous intéressons ici aux caractéristiques *language* (la langue, par exemple l'anglais, critère principal utilisé), *country* (le pays, par exemple US), et *variant* (la variante régionale).

On veut dans cet exercice traiter des textes formatés de la manière suivante :

— Ils sont sous forme de chaîne de caractères (`String`).
— La première ligne désigne la langue du texte sous la forme suivante :
  — la langue, sous forme ISO 639 (2 ou 3 caractères alphanumériques) suivie d'une espace
  — puis, éventuellement, le pays sous forme ISO 3166 (2 lettres ou 3 chiffres) suivi d'une espace
  — puis, éventuellement et uniquement dans le cas où le pays est indiqué, la variante sous une forme compatible avec la propriété correspondante de `Locale` suivie d'un espace
  — et finit par un saut de ligne (`'\n'`), toujours présent.
— La suite de la chaîne de caractère est le contenu du texte.

Vous pouvez trouver de (brefs) exemples de tels textes dans la fonction `main` du fichier `TextFilterByLanguage.java` à récupérer sur la page du cours. Dans toute la section, on supposera les textes bien formés (pas de gestion d'erreur).

Le but, à travers les question de cette section, est de compléter la classe `TextFilterByLanguage` dont le squelette est donné sur le site. Vous pouvez (et êtes encouragés à) consulter les javadoc des classes utilisées.

1. Pour commencer, allez ouvrir les pages de documentations des trois classes `Locale`, `ArrayList` et `HashMap`. Gardez les onglets à portée de main, vous en aurez besoin au cours de cet exercice.
2. Écrire la méthode **`public static`** `Locale readLang(String text)` qui prend un texte dont la première ligne indique sa langue comme indiqué en introduction et renvoie la locale correspondant à son language (on ignore le pays et la variante éventuellement présents).
3. Écrire la méthode **`public static`** `Locale readLocale(String text)` qui prend un texte dont la première ligne indique sa langue, et éventuellement un pays et une variante, comme indiqué en introduction et renvoie la locale correspondante.
4. Écrire la méthode `displayLanguage(String text)` qui prend un texte et affiche sur la sortie standard la partie language de sa langue, dans la langue du système de l'utilisateur du programme.
   Indice : On utilise le terme *display* pour indiquer un affichage adapté ; et par défaut les méthodes correspondantes de `Locale` ne prenant pas d'argument utilisent la langue du système de l'utilisateur.
5. Écrire la méthode
   `public static` `ArrayList<String> filterByLang(ArrayList<String> textCollection, String lang)`
   qui prend une collection de textes, sous la forme indiquée en introduction, et une langue (dans la langue du système l'utilisateur du programme, telle qu'elle serait affichée par défaut, par exemple "anglais") et renvoie les textes de la collection qui sont dans la langue indiquée, en-tête comprise.
6. Écrire la méthode
   `public static` `HashMap<String, ArrayList<String>> groupByLang(ArrayList<String> textCollection)`
   qui prend un ensemble de textes, et crée un tableau associatif (`HashMap`), associant une langue (dans la langue du système l'utilisateur du programme, telle qu'elle serait affichée par défaut, par exemple "anglais") et l'ensemble des textes en cette langue, en-tête comprise.

3. Questions variées sur les chaînes.

1. Écrire une fonction `removeSpaces` qui supprime tous les espaces d'une chaine de caractères. Si l'entrée est `"a b c"`, la fonction renverra `"abc"`.
2. Écrire une fonction inverse qui renvoie la chaine de caractères passée en paramètre « inversée » : si l'entrée est `"abca"`, la fonction renverra `acba`.
3. Écrire une fonction `String expandNumber(String input)` qui prend en entrée une chaine de caractères composée de lettres et de nombres, et qui renvoie une chaine de caractères dans laquelle chaque nombre `N` a été remplacé par `N - 1` fois le caractère précédent. Voici quelques exemples de sorties attendues de la fonction :

entrée	sortie
abc	abc
abc2	abcc
ab2c	abbc
3abc	abc
a12bc	aaaaaaaaaaaabc

(le 3 suit une chaine vide)

Extraction non supervisée de lexique bilingue

Alice Jacquot (d'après un cours de Guillaume Wisniewski)
https://www.lri.fr/~jacquot/ipo/

septembre 2019

1 Extraction de lexique

L'objectif de ce TP est d'extraire automatiquement un lexique bilingue d'un ensemble de textes. Un lexique bilingue est un dictionnaire [1] qui associe à un mot d'une langue source la liste de ses traductions possibles dans une langue cible. La Table 1 donne un extrait d'un lexique français–anglais.

cuisine	cooks, cook, cooking, kitchen, food...
comprendre	understand, understood, understanding, realize, ...
être	human, being, be, ...
économie	free-market, Economy, economics, market, economy, ...

TABLE 1 – Extrait d'un lexique français–anglais

L'extraction sera *non supervisée* : la méthode utilisée prendra en entrée uniquement un ensemble de paire de phrases (une phrase en français et sa traduction en anglais). On appelle « corpus parallèle » un tel corpus et on qualifie de « parallèles » des phrases qui sont traductions l'une de l'autre. La Table 2 donne un exemple de corpus parallèle.

L'approche proposée est fondée sur une théorie linguistique, l'*hypothèse distributionnelle* : le sens d'un mot peut se déduire directement du contexte dans lequel celui-ci apparait. La généralisation au cas bilingue de cette hypothèse peut se formuler de la manière suivante : un mot français et un mot anglais qui *co-occurent* (apparaissent dans une paire de phrases parallèles) fréquemment ont une grande chance d'être traduction l'un de l'autre. Ainsi sur l'exemple du corpus de la Table 2, il est naturel de supposer que *cuisine* se traduit soit par *cooking*, soit par *kitchen* puisque ce sont les seuls mots qui apparaissent dans toutes les traductions.

1. En considérant le corpus français–grec, décrit Table 3 pouvez-vous donner (en les justifiants) les traductions en grec des mots `Elli`, `est` et `maison`.

1. aussi bien au sens informatique qu'au sens « général »

Living on my own, I really miss my Mom's **cooking**.
Vivant seul, la **cuisine** de ma mère me manque.
She left the **kitchen** with the kettle boiling.
Elle quitta la **cuisine** avec la bouilloire.
Is there any coffee in the **kitchen**?
Y a-t-il encore du café dans la **cuisine** ?
Cooking runs in my family.
La **cuisine** c'est de famille.
Both boys and girls should take **cooking** class in school.
Garçons et filles devraient suivre des cours de **cuisine** à l'école.

TABLE 2 – Exemple d'un corpus parallèle français–anglais

Πάω στο σπίτι μας.
Je vais chez nous (*lit.* dans notre maison).
Το σπίτι μου είναι μεγάλο.
Ma maison est grande
Το σπίτι της Έλλης είναι κοντά στην παραλία.
La maison d'Elli est à côté de la plage.
Με λένε Έλλη.
Je m'appelle Elli.
Ένα σπίτι του χωριού κάηκε
Une maison du village a brûlé.
Αγαπώ την Έλλη.
J'aime Elli.

TABLE 3 – Exemple d'un corpus parallèle grec–français

2 Approche naïve

Vous trouverez sur le site du cours deux fichiers contenant le roman *Voyage au centre de la Terre* en anglais et en français. Les documents ont été pré-traités pour :

— être alignés au niveau des phrases : la i^e ligne du fichier en français est la traduction de la i^e ligne du fichier anglais.

— segmentés en mots : cette segmentation consiste à séparer certains signes (par exemple « `à_l'école.` » est ré-écrit en « `à_l'_école_.` ») et à en regrouper d'autres (« 100 000 » est récrit en « 100000 »).

L'extraction d'un lexique à partir de ce corpus parallèle repose sur la construction d'un table de co-occurences. Cette table peut-être modélisée par une variable de type `HashMap<String, HashMap<String, Integer>` > qui associe à un mot français (la 1re clef), une `HashMap` dont les clefs sont l'ensemble des mots anglais co-occurant avec le mot français et les valeurs le nombre de paires de phrases dans lesquelles le mot français et le mot anglais apparaissent tous les deux. Par exemple, si le corpus est constitué de deux phrases :

doc n° 1	doc n° 2
la vache et le veau	the cow and the calf
le chien et le chat	the dog and the cat

la table extraite sera :

```
{'chat': {'and': 1, 'the': 1, 'dog': 1, 'cat': 1},
 'chien': {'and': 1, 'the': 1, 'dog': 1, 'cat': 1},
 'et': {'and': 2, 'calf': 1, 'cat': 1, 'cow': 1, 'dog': 1, 'the': 2},
 'la': {'and': 1, 'the': 1, 'cow': 1, 'calf': 1},
 'le': {'and': 2, 'calf': 1, 'cat': 1, 'cow': 1, 'dog': 1, 'the': 2},
 'vache': {'and': 1, 'calf': 1, 'cow': 1, 'the': 1},
 'veau': {'and': 1, 'the': 1, 'cow': 1, 'calf': 1}}
```

2. Quel est l'intérêt de la segmentation en mot ?
3. Écrivez une méthode `countSentencesWithWord` dont l'entrée est un nom de fichier et qui renvoie une `HashMap` associant à chaque mot du fichier le nombre de phrases dans lequel il apparait. **Attention** : on cherche à déterminer le nombre de phrases dans lequel un mot apparait, et même si un mot apparait plusieurs fois dans une même phrase, il ne faut le compter qu'une fois. Il est possible de supprimer les doublons d'une liste en utilisant l'instruction :

   ```
   HashSet<String> uniqWord = new HashSet<String>(Arrays.asList(tab));
   ```

   qui à partir d'un tableau de `String` (par exemple `["a", "b", "a"]`) construit une *liste* sans doublons (si on reprend l'exemple précédent `["b", "a"]`).
4. Écrivez une méthode `buildCoocTable` qui prend en argument deux noms de fichiers décrivant un corpus parallèle et retourne la table de co-occurence de ce corpus. À nouveau, il faut penser à supprimer les doublons à l'intérieur d'une phrase avant de réaliser les comptages.

5. À l'aide de la méthode `printSortedCoocTable`, afficher la table de co-occurences par ordre de fréquence décroissant.
6. Interprétez le résultat obtenu : est-ce que cette méthode permet de construire un « bon » lexique ?

Remarque : pensez à regarder le squelette qui vous est fourni sur le site du cours.

3 Tests statistiques

L'approche de la section précédente n'est utilisable que si l'on dispose d'un moyen de distinguer les associations « porteuses de sens » (celles que l'on observe dans un corpus parce que les mots sont traduction l'un de l'autre) des associations dues au hasard (par exemple, celles que l'on observe que parce que notre corpus est trop petit ou à cause des mots fréquents).

Les *tests de significativité* sont des outils statistiques qui répondent à cet objectif. Dans la suite du TP nous proposons d'utiliser un de ces tests, le *likelihood ratio*, pour filtrer la table des co-occurences construite dans la partie précédente et ne garder que les associations les *significatives*. Ce test évalue l'intérêt d'une association entre le mot a et le mot b à partir de sa table de contingence. Cette table décrit :

— $n(a, b)$ le nombre de paires de phrases dans lesquelles a et b co-occurent ;
— $n(\neg a, b)$ le nombre de paires de phrases dans lesquelles b apparait, mais pas a ;
— $n(a, \neg b)$ le nombre de paires de phrases dans lesquelles a apparait, mais pas b ;
— $n(\neg a, \neg b)$ le nombre de paires de phrases dans lesquelles ni a, ni b n'apparaissent.

La table 4 résume ces notations.

	nbre phrase avec a	nbre phrase sans a	
nbre phrase avec b	$n(a, b)$	$n(\neg a, b)$	$n(b)$
nbre phrase sans b	$n(a, \neg b)$	$n(\neg a, \neg b)$	$n(\neg b)$
	$n(a)$	$n(\neg a)$	N

Table 4 – Table de contingence

Le *likelihood ratio* est défini par :

$$G^2 = 2 \cdot N \left[\sum_{a? \in \{a, \neg a\}} \sum_{b? \in \{b, \neg b\}} p(a? \text{ et } b?) \cdot \log \frac{p(a? \text{ et } b?)}{p(a?) \cdot p(b?)} \right]$$

où $p(x?)$ est la fréquence de l'événement $x?$ (le rapport entre le nombre de fois où $x?$ est vrai et le nombre de fois où $x?$ est vrai *ou* faux). Par exemple, $p(a) = \frac{n(a)}{N}$ et $p(a, \neg b) = \frac{n(a, \neg b)}{N}$.

7. On note $n(a)$ (resp. $n(b)$) le nombre de phrases dans lesquelles a (resp. b) apparaît et N le nombre total de phrases. Exprimez $n(\neg a, \neg b)$, $n(\neg a, b)$ et $n(a, \neg b)$ en fonction de $n(a)$, $n(b)$, $n(a, b)$ et N.
8. Écrivez une méthode qui calcule le *likelihood ratio* d'une paire mot français / mot anglais. Cette méthode prendra en paramètre :

— le nombre de phrases du corpus ;
— le nombre de phrases dans lesquelles le mot apparait ;
— le nombre de phrases dans lesquelles le mot anglais apparait ;
— le nombre de phrases parallèles dans lesquelles la paire de mot apparait.

9. Écrivez une méthode qui calcule le likelihood ratio de l'ensemble des paires de mots du corpus. Cette méthode renverra une instance de `HashMap<String, Double>` qui associe à une paire de mots (obtenu, par exemple, en concaténant le mot français et le mot anglais) son score. Seules les paires dont le likelihood ratio n'est pas infini devront être ajoutées au résultat. Pour savoir si un `Double` est infini, on peut utiliser l'instruction

   ```
   lr.equals(Double.NaN)
   ```

10. Afficher les associations par ordre décroissant de *likelihood ratio* croissant. Que peut-on en conclure ?

IPO – TP 4 : premières classes

http://www.lri.fr/~blsk/IPO/

Tableaux décalés Dans certains langages de programmation, on définit pour chaque tableau son indice de début et son indice de fin. Ainsi, on peut déclarer un tableau de longueur 4 comme allant de l'indice 2 (inclus) à l'indice 6 (exclu), ou même de l'indice -2 (inclus) à l'indice 2 (exclu).

On simule ceci à l'aide d'une classe `Tab` dont les attributs sont un tableau primitif Java de la taille voulue, et les indices de début (inclus) et de fin (exclu). On considère que notre tableau contient des chaînes de caractères. Définir une telle classe `Tab` pour qu'elle contienne :

1. les bonnes déclarations d'attributs,
2. un constructeur prenant en paramètres un indice de début (inclus) et un indice de fin (exclu),
3. une méthode `length()` donnant la longueur du tableau,
4. une méthode `get(int i)` renvoyant l'élément du tableau à l'indice `i` (en tenant compte du décalage : si le premier indice du tableau est 3, un accès `get(3)` renverra le premier élément, un accès `get(4)` le deuxième élément, etc.),
5. une méthode `set(int i, String s)` plaçant la chaîne `s` à l'indice `i` (en tenant compte du décalage),
6. une méthode `main` exécutant ces opérations sur quelques exemples.

Si, lors de la construction, l'utilisateur fournit les bornes dans le mauvais ordre, on créera un tableau de longueur zéro. Si les méthodes `get` et `set` reçoivent en paramètre un indice invalide, on laissera se déclencher l'erreur `ArrayIndexOutOfBoundsException` de Java.

Fractions Définir une classe `Fraction` dont chaque objet est une fraction représentée par son numérateur et son dénominateur. On demande que le dénominateur soit strictement positif. La classe devra comporter les méthodes suivantes :

1. une méthode `String toString()` qui renvoie une représentation de la fraction sous la forme `n / d`, où `n` est le numérateur et `d` le dénominateur ; dans le cas où le numérateur vaut `1`, on affichera simplement `n`,
2. deux méthodes `Fraction add(Fraction f)` et `Fraction mul(Fraction f)` qui renvoient chacune une *nouvelle* fraction, égale à la somme ou au produit de **`this`** et `f`,
3. une méthode **`boolean`** `egale(Fraction f)` qui renvoie **`true`** si la fraction `f` est (numériquement) égale à la fraction courante,
4. une méthode **`int`** `compareTo(Fraction f)` qui renvoie un entier qui est :
   - positif si **`this`** est strictement plus grande que `f`,
   - négatif si **`this`** est strictement plus petite que `f`,
   - zéro si **`this`** et `f` sont égales.
5. une méthode `main` qui teste toutes ces opérations sur quelques exemples.

Bonus : faire en sorte que les fractions soient toujours réduites, c'est-à-dire que le numérateur et le dénominateur n'aient pas de facteur non trivial commun.

Chronomètre On veut créer une classe `Chrono` représentant un temps, mesuré en heures, minutes et secondes. Définir une telle classe avec les éléments suivantes :

1. trois attributs entiers pour les heures, minutes et secondes, et un constructeur,
2. une méthode `String toString()` qui donne d'un temps un affichage agréable,
3. une méthode `int toSeconds()` qui convertit un temps en un nombre de secondes,
4. une méthode `void normalise()` qui modifie un objet de la classe `Chrono` de sorte à s'assurer que les secondes et les minutes appartiennent toujours à l'intervalle $[0, 59]$,
5. une méthode `boolean equals(Object o)` qui renvoie `true` si l'objet passé en paramètre représente le même temps que `this`,
6. une méthode `boolean avant(Chrono c)` qui renvoie `true` si le temps représenté par `this` est inférieur ou égal au temps `c`,
7. une méthode `void avance(int n)` qui fait avancer un temps de `n` secondes,
8. une méthode `Chrono diff(Chrono c)` qui renvoie un *nouvel* objet `Chrono` représentant l'écart de temps entre `c` et `this`,
9. une méthode `main` testant toutes ces opérations sur quelques exemples.

Intervalles On veut définir une classe `Intervalle` représentant des intervalles de temps. Un intervalle est défini par deux objets de la classe `Chrono`, représentant respectivement le début et la fin de l'intervalle. Définir une telle classe avec les éléments suivants :

1. des attributs représentant le temps de début et le temps de fin,
2. un constructeur prenant en paramètres deux temps, qui fera en sorte de créer un intervalle de durée zéro si les bornes ne sont pas fournies dans le bon ordre,
3. une méthode `String toString()` renvoyant une chaîne de caractères représentant l'intervalle,
4. une méthode `Chrono duree()` renvoyant un temps représentant la durée de l'intervalle,
5. une méthode `boolean avant(Intervalle i)` qui renvoie `true` si l'intervalle `this` est intégralement avant l'intervalle `i`.
6. une méthode `boolean conflit(Intervalle i)` qui renvoie `true` si `this` et `i` ont une intersection non vide,
7. une méthode `main` testant toutes ces opérations sur quelques exemples.

Agenda On appellera un *agenda* une collection d'intervalles deux à deux disjoints. On veut créer une classe `Agenda` avec un attribut de type `ArrayList<Intervalle>`, comportant des intervalles disjoints rangés par ordre chronologique. Définir une telle classe avec les éléments suivants :

1. le tableau attribut, et un constructeur créant un agenda vide,
2. une méthode `String toString()` affichant l'intégralité d'un agenda,
3. une méthode `boolean compatible(Intervalle i)` qui renvoie `true` si l'intervalle donné en paramètre est bien disjoint de tous les intervalles déjà présents dans l'agenda,
4. une méthode `boolean insertion(Intervalle i)` qui insère l'intervalle `i` dans l'agenda `this` si cela est possible (et dans ce cas renvoie `true`), ou qui renvoie `false` sinon.
5. une méthode `main` testant toutes ces opérations sur quelques exemples.

IPO – TP 5 : conception objet

https://www.lri.fr/~blsk/IPO/

Objectifs On souhaite écrire un petit programme permettant de créer et de répondre à des QCM, répondant au cahier des charges suivant. Un QCM est un ensemble de questions. Chaque question contient un énoncé et une liste de plusieurs réponses possibles. Les énoncés comme les réponses sont donnés sous la forme d'un texte. Chaque réponse peut être correcte ou incorrecte. Une question peut avoir plusieurs réponses correctes. Le créateur du QCM veillera à ce qu'au moins une réponse par question soit correcte.

Vous allez devoir créer votre programme et vos classes en partant de zéro. Nous vous fournissons un unique fichier `Utils.java`, contenant une méthode qui demande à l'utilisateur d'entrer une valeur entière entre 0 et un entier donné en paramètre et qui renvoie la valeur entrée par l'utilisateur.

Création d'un QCM Avant de se lancer dans le code, prenez une feuille et un crayon : il va falloir planifier. On commence par s'intéresser aux manières de représenter et d'initialiser un QCM. On aura besoin notamment de pouvoir créer un QCM, des questions, des réponses, et d'associer des réponse à une questions, et des question à un QCM.

1. Quelles sont les classes à créer pour répondre au cahier des charges ci-dessus ? Dans quelle classe mettriez-vous votre fonction `main` ?
2. Pour chaque classe, de quels attributs aurons-nous besoin ?
3. Quels paramètres donner au constructeur de chacune des classes ? Aurons-nous besoin de méthodes supplémentaires pour initialiser un QCM ?

Une fois ceci fixé, vous pouvez commencer à coder.

4. Définir les classes que vous avez décrites dans les questions précédentes, et y coder les constructeurs et les méthodes nécessaire à la création d'un QCM.
5. Dans le `main`, définir un QCM composé des deux questions suivantes.
   - À propos des nombres premiers :
     - Le nomre 1 est premier (faux)
     - Il en existe une infinité (vrai)
     - Un nombre pair peut être premier (vrai)
     - Le produit de deux nombres premiers est premier (faux)
   - Quel temps fait-il aujourd'hui ?
     - Nuageux (vrai)
     - Pluvieux (faux)
     - Orageux (faux)
6. Écrire, dans la classe `QCM`, une méthode `isValid` qui vérifie la validité d'un QCM, c'est-à-dire qui vérifie que chaque question a bien au moins une réponse juste.
   Indication : vous pourrez avoir besoin de définir plusieurs méthodes dans plusieurs classes pour cela.

Affichage Les méthodes de cette section permettront d'afficher un QCM. Vous pourrez également vous en servir pour tester le résultat des questions précédentes.

7. Écrire la méthode `toString` pour une réponse. Cette méthode doit renvoyer une chaîne de caractères donnant le texte associé à la réponse, mais ne doit pas indiquer si la réponse est juste ou fausse.
   Exemple :

```
"Le nombre 1 est premier"
```

8. Écrire la méthode `toString` pour une question. Cette méthode doit renvoyer une chaîne de caractères donnant l'énoncé de la question, suivi de chaque réponse possible. Chaque réponse doit être précédée de son numéro (on commencera la numérotation à 1) et deux réponses doivent être séparées par un saut de ligne.

Exemple :

```
"Quel temps fait-il aujourd'hui ?
   1- Nuageux
   2- Pluvieux
   3- Orageux
"
```

9. Écrire la méthode `toString` pour un QCM. Cette méthode doit renvoyer une chaîne de caractères donnant l'ensemble des questions et des réponses possibles.

Répondre à un QCM Vous allez maintenant étendre les classes précédentes, pour permettre à un utilisateur de répondre à un QCM. On considère pour l'instant que l'utilisateur ne donne qu'une seule réponse par question (et qu'il répond bien à toutes les questions). La réponse de l'utilisateur est considérée comme juste dès lors que la réponse choisie est l'une des réponses correctes.

On utilisera les mêmes classes pour l'énoncé d'un QCM et pour les QCM donnés à chaque utilisateur. On considère que chaque utilisateur répond à sa propre instance du QCM. Il vous faut donc ajouter à vos classes de quoi enregistrer les réponses données par *un* utilisateur.

10. On ajoute dans la classe représentant les questions un attribut indiquant le numéro de réponse sélectionnée. Quel est l'intérêt d'utiliser le type `Integer` plutôt que **`int`** dans ce contexte ?
11. Écrire une méthode dans la classe `Question` qui indique si l'utilisateur a correctement répondu, c'est-à-dire s'il existe une réponse et que cette réponse est effectivement correcte.

    Indication : vous pouvez ajouter une méthode dans une autre classe si nécessaire.
12. Écrire dans la classe `Question` une méthode permettant de poser la question : elle en affiche l'énoncé, attend que l'utilisateur saisisse un entier valide (pour pouvez utiliser la méthode fournie dans `Utils.java`) et sélectionne la réponse correspondante.
13. Écrire une méthode qui permet de répondre à un QCM et affiche le score de l'utilisateur à la fin du questionnaire.
14. Écrire une méthode qui, à partir d'un QCM d'origine, crée une nouvelle instance ave les mêmes questions (mais sans aucune réponse), de sorte qu'un nouvel utilisateur puisse répondre au QCM.

Raffinements Voici quelques idées pour enrichir le système.

15. Associer à chaque question un barème. On peut y compris prévoir des points positifs ou négatifs réponse par réponse. Ajuster le calcul du score et la méthode `isValid`.
16. Permettre à l'utilisateur de revenir à une question précédente pour annuler, modifier ou compléter sa réponse.
17. Permettre à l'utilisateur de sélectionner plusieurs réponses pour une question, et ne compter comme validées que les questions où l'utilisateur a sélectionné toutes les réponses justes et aucune réponse fausse. Pour saisir plusieurs réponses, vous pouvez vous baser sur le mécanisme précédent, ou en imaginer un autre.
18. Permettre à plusieurs utilisateurs de répondre à un questionnaire, mémoriser leurs résultats, puis donner des statistiques sur les performances du groupe.

IPO – TP 6 : interfaces

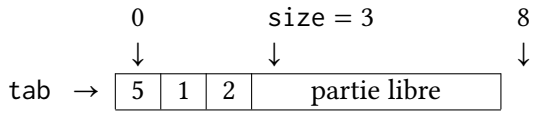
http://www.lri.fr/~blsk/IPO/

Ensembles On veut manipuler des ensembles d'entiers. L'objectif de cet exercice est de définir une interface pour une telle structure, et deux concrétisations différentes de cette interface.

1. Définir une interface `Set` pour une structure de données représentant un ensemble d'entiers (type `Integer`). On veut au minimum des méthodes correspondant aux descriptions suivantes :
   - une méthode `contains` indiquant si un élément donné appartient ou non à l'ensemble,
   - une méthode `cardinal` donnant le nombre d'éléments d'un ensemble,
   - une méthode `toString` qui renvoie une chaîne de caractères décrivant l'ensemble des éléments de l'ensemble (dans un ordre arbitraire),
   - une méthode `add` qui modifie un ensemble en lui ajoutant un élément, et une méthode `remove` qui retire un élément,
   - une méthode `clone` qui renvoie une copie d'un ensemble, de sorte que les modifications de l'une des copies n'ait pas d'influence sur les autres copies ni sur l'objet d'origine,
   - une méthode `union` renvoyant un nouvel ensemble obtenu par union de l'ensemble courant et d'un autre donné en paramètre, sans modifier aucun de ces ensembles d'origine, et une méthode `intersection` similaire.
2. Écrire dans une classe principale une méthode qui teste quelques opérations de cette structure (vous ne pourrez pas l'exécuter tout de suite, à défaut d'une concrétisation des ensembles).

On propose une première réalisation utilisant un tableau `tab` sous-jacent de taille fixe, et un attribut `size` donnant le nombre de cases de `tab` effectivement utilisées (en supposant, comme dans l'exemple de tableau associatif vu en cours, que les éléments utilisés sont dans les `size` premières cases).

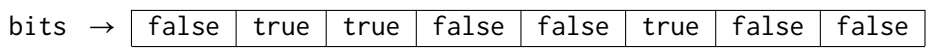
Une représentation possible de l'ensemble $\{1, 2, 5\}$ avec un tableau sous-jacent de taille 8 :



3. Définir une classe `TabSet` concrétisant l'interface `Set`, avec les caractéristiques suivantes :
   - utilisation d'un tableau primitif `tab` et d'un attribut `size`,
   - la taille (fixe) de `tab` est donnée en paramètre au constructeur,
   - un entier donné ne peut apparaître qu'une seule fois dans l'ensemble,
   - on s'autorise à échouer si un utilisateur tente d'ajouter des éléments à un `TabSet` dont le tableau sous-jacent est déjà plein.

   Indication : la méthode `union` doit avoir la signature `public Set union(Set s)`. Elle ne peut donc utiliser que les éléments de l'interface, et aucune connaissance spécifique à la classe `TabSet`.
4. Exécuter les tests définis précédemment sur l'interface `Set` en utilisant cette concrétisation.

On propose une deuxième réalisation utilisant un tableau `bits` de booléens, dans lequel `bits[k]` vaut **`true`** si et seulement l'ensemble représenté contient l'entier `k`. La représentation de l'ensemble $\{1, 2, 5\}$ par un tel tableau de taille 8 serait :



5. Définir une classe `BitSet` concrétisant l'interface `Set`, avec les caractéristiques suivantes :
   - utilisation d'un tableau primitif de booléens `bits`,
   - la taille (fixe) de `bits` est donnée en paramètre au constructeur.
6. Exécuter à nouveau les tests de `Set` avec cette nouvelle version.

Question supplémentaire, à garder pour quand vous aurez terminé la page suivante.

7. Définir les méthodes `equals` des classes `TabSet` et `BitSet`.

Itérateurs On s'intéresse maintenant aux deux interfaces `Iterable` et `Iterator` fournies par le langage Java (`java.util.Iterable` et `java.util.Iterator`), qui permettent d'itérer sur une collection à l'aide d'une boucle *for each*.

```
for (Integer k : set) {
  ...
}
```

Une telle boucle énumère tous les éléments d'un ensemble `set` d'entiers. On peut l'écrire en Java dès lors que l'objet `set` appartient à une classe `C` qui concrétise l'interface `Iterable<Integer>` des ensembles énumérables d'entiers.

Pour concrétiser l'interface `Iterable<Integer>`, une classe comme `TabSet` ou `BitSet` doit définir une méthode

```
public Iterator<Integer> iterator() { ... }
```

Cette méthode construit et renvoie un *itérateur*, c'est-à-dire un objet chargé d'énumérer tous les éléments de l'ensemble. L'itérateur est un objet d'une classe concrétisant l'interface `Iterator<Integer>`. Une telle classe doit fournir une méthode

```
public boolean hasNext() { ... }
```

qui renvoie **`true`** si et seulement il reste au moins un élément à énumérer, et une méthode

```
public Integer next() { ... }
```

qui renvoie le prochain entier de l'énumération. La méthode `next` est telle que chaque nouvel appel renvoie un nouvel élément de l'ensemble à énumérer. Elle fait donc évoluer l'état de l'itérateur. Ces méthodes, utilisées ensembles, permettraient d'écrire le code suivant, qui énumère tous les éléments d'un ensemble `set` d'entiers.

```
Iterator<Integer> it = set.iterator();
while (it.hasNext()) {
  Integer k = it.next();
  ...
}
```

La boucle *for each* est précisément une syntaxe simplifiée pour les lignes précédentes.

On propose d'énumérer les éléments d'un `TabSet` dans l'ordre donné par le tableau `tab` sous-jacent. Il suffit pour cela de se donner un indice `i` désignant le prochain élément à renvoyer, et d'incrémenter cet indice à chaque appel à `next`.

1. Compléter la ligne **`public class`** `TabSet` **`implements`** `Set` en

   ```
   public class TabSet implements Set, Iterable<Integer> {
   ```

   Cela permet de déclarer que la classe `TabSet` concrétise simultanément les deux interfaces `Set` et `Iterable<Integer>`. Ajouter à la classe `TabSet` la définition suivante,

   ```
   public Iterator<Integer> iterator() {
       return new TabSetIterator(tab, size);
   }
   ```

2. Définir la classe `TabSetIterator` concrétisant l'interface `Iterator<Integer>`. Il faut donc définir, en plus du constructeur, les méthodes `next` et `hasNext`.
3. Ajouter dans la fonction principale un test énumérant tous les éléments d'un `TabSet` pour les afficher, à l'aide d'une boucle *for each*.

On propose d'énumérer les éléments d'un `BitSet` dans l'ordre croissant.

4. Créer une classe `BitSetIterator`, qui concrétise l'interface `Iterator<Integer>`. À vous de préciser quels éléments devra contenir cette classe pour permettre l'itération.
5. Compléter la classe `BitSet` afin qu'elle concrétise l'interface `Iterable<Integer>`.

IPO – TP 8 : héritage

http://www.lri.fr/~blsk/IPO/

Médias et médiathèque On souhaite développer une application pour une médiathèque qui met à disposition de ses utilisateurs des médias variés (livres, musique, vidéos). Chaque type de média sera naturellement représenté par une classe. Chaque média est décrit par un titre et un auteur. Les utilisateurs peuvent donner leur avis sur la qualité des médias en leur attribuant des notes comprises entre 0 et 5.

On veut pouvoir réaliser les opérations générales suivantes pour tous les médias.

— Représenter le média par une chaîne de caractères (méthode `toString()`) dont le format suit le modèle :

```
"Titre" par Auteur
```

— Donner son avis sur le média à l'aide d'une méthode `vote(int note)`, où la `note` doit être comprise entre 0 et 5.

— Consulter la moyenne des notes reçues par une méthode `moyenneNotes()`. Si le média n'a reçu aucune note, cette méthode doit renvoyer 0.

Remarque : on ne distingue pas dans cette étude les médias physiques des médias numériques, téléchargeables ou consultables en ligne.

1. Écrire le code d'une classe `Media` réalisant ces différents éléments.

Les différents types de médias vont ensuite avoir des caractéristiques spécifiques.

— Un livre possède un nombre de pages.

— Un média audio ou vidéo a une durée.

— Un média audio peut être un album ou un titre isolé.

— Un média vidéo peut posséder des sous-titres.

Pour chaque type de média, on a besoin de constructeurs adaptés qui permettront par exemple d'écrire :

```
Media l  = new Livre("Les furtifs", "Alain Damasio", 704);
Media a1 = new Audio("Petit frère", "IAM", "single", 4);
Media a2 = new Audio("Origin of Symmetry", "Muse", 51);
Media v  = new Video("The Train Job", "Joss Whedon",
                     new String[]{"en", "de", "fr"}, 45);
```

2. Écrire la classe `Livre`.
3. Écrire la classe `Audio`. On représentera le type (album ou *single*) par une chaîne de caractères. À défaut de précision, l'objet construit devra être un album. On suppose que la durée est donnée en minutes, arrondie à un nombre entier.
4. Écrire la classe `Video`. On représentera l'ensemble des sous-titres disponibles par une collection de chaînes de caractères.
5. Avez-vous trouvé quelque part une redondance que vous pourriez éliminer, quitte à créer une classe supplémentaire ?

On veut également pouvoir modéliser les séries. Une série est un média contenant une liste de médias préalablement définis. Remarquez qu'il est donc possible de noter indépendamment une série elle-même et chacun de ses épisodes.

```
Media s = new Serie("Firefly", "Joss Whedon", new Media[]{v});
v.vote(5);
s.vote(5);
```

6. Écrire la classe `Serie`, de sorte à ce qu'on puisse facilement ajouter de nouveaux éléments à une série déjà définie.
7. Écrire une méthode `moyenneAgregee()` qui calcule une note moyenne pour une série en combinant la moyenne de ses notes globales et les moyennes de ses épisodes.

Gestion des recherches simples Dans une classe `Mediatheque`, la base de données de la médiathèque est une `ArrayList` de `Media`.

8. Écrire une classe `Mediatheque` possédant ceci, ainsi qu'une méthode `add(Media m)` qui permet d'ajouter un média à la collection.

On va maintenant réaliser des méthodes de recherche dans la médiathèque en fonction de critères variés. Pour les recherches basées sur des chaînes de caractères, on ne tiendra pas compte de la casse (majuscule/minuscule).

9. Trouvez dans la bibliothèque standard Java une méthode de comparaison de chaînes de caractères adaptée.
10. Écrire une méthode `searchByTitle(String titre)` qui renvoie un `ArrayList<Media>` contenant l'ensemble des médias ayant le titre passé en paramètre.
11. Écrire une méthode `searchByAuthor(String auteur)` qui renvoie un `ArrayList<Media>` contenant l'ensemble des médias ayant l'auteur passé en paramètre.
12. Écrire une méthode `filtre(String critere, String valeur)` qui renvoie un `ArrayList<Media>` contenant l'ensemble des médias vérifiant le critère passé en paramètre. Ce critère peut porter :
    - sur l'auteur ;
    - sur le titre ;
    - sur le type de média (on pourra utiliser le test `nomObjet` **`instanceof`** `nomClasse` qui renvoie **`true`** si l'objet `nomObjet` est une instance de la classe `nomClasse` ou de l'une de ses classes filles)

    Par exemple : `m.filtre("titre", "Firefly")` renverra l'ensemble des médias dont le titre est Firefly (avec ou sans majuscules) et `m.filtre("media", "audio")` renverra l'ensemble des musiques de la collection.

IPO – TP 9 : classes abstraites

http://www.lri.fr/~blsk/IPO/

On souhaite réaliser un jeu dans lequel différentes créatures se déplacent dans un labyrinthe : des personnages cherchant à s'échapper, et des monstres susceptibles de manger ces personnages. Le labyrinthe prendra la forme d'un ensemble de cases organisées dans un tableau à deux dimensions. Chaque case peut être soit une case intraversable (un mur), soit une case libre, soit une sortie. En plus des personnages et des monstres, les cases peuvent contenir des obstacles, que les créatures sont susceptibles de détruire.

Un objectif de ce TP est d'organiser le code des différentes sortes de cases et des différents occupants du labyrinthe en une hiérarchie de classes et de classes abstraites, de sorte à éviter au mieux les redondances dans le code. On fournit sur la page du cours deux squelettes de classes `Jeu` et `Terrain`, une énumération `Direction`, et le descriptif d'un terrain exemple (`laby1.txt`).

Cases Chaque case est caractérisée par ses coordonnées (numéro de ligne et numéro de colonne dans le tableau). Chaque case libre ou sortie peut contenir au plus une entité (obstacle ou créature). Une case intraversable ne contient rien.

1. Définir une classe abstraite `Case` avec deux attributs `lig` et `col` désignant les coordonnées de la case, et une méthode abstraite de signature `boolean estLibre()`. Note : une fois une case créée, les valeurs des deux attributs `lig` et `col` ne changent jamais.
2. Définir une classe abstraite `Entite`, avec un attribut entier `resistance` et une méthode abstraite de signature `String toString(String background)`.
3. Définir une classe `CaseTraversable` étant `Case` et possédant un attribut `contenu` (une entité), et les méthodes suivantes.
   - `Entite getContenu()` : renvoie le contenu,
   - `void vide()` : retire le contenu, et
   - `void entre(Entite e)` : définit le nouveau contenu, en supposant que la case ne contenait pas encore d'entité.
4. Définir trois classes `CaseIntraversable`, `CaseLibre` et `Sortie` respectant la description précédente. Chacune doit étendre `Case` (éventuellement via `CaseTraversable`), et concrétiser la méthode `estLibre()` de sorte à ce qu'elle renvoie **`true`** si et seulement si la case concernée est susceptible d'accueillir une entité *et* n'est pas déjà occupée.
5. Ajouter des méthodes `toString` aux classes des cases de sorte que chacune s'affiche sous la forme d'une chaîne de 3 caractères :
   - "###" pour les cases intraversables,
   - " " pour les cases libres,
   - "( )" pour les sorties.
6. Ajouter à la classe `Terrain` une méthode `void print()` qui affiche le terrain complet en utilisant les représentations précédentes.

Obstacles Les obstacles sont des entités, qui peuvent se trouver soit sur une case libre soit sur une sortie. Les obstacles ne peuvent pas se déplacer, mais sont détruits lorsque leur `resistance` atteint zéro.

7. Définir une classe `Obstacle` concrétisant `Entite`. On veut deux constructeurs : un constructeur par défaut fixant la résistance à 3, et un constructeur prenant une résistance en paramètre. La méthode `String toString(String background)` doit être concrétisée de la manière suivante.
   - On suppose que `background` est une chaîne de longueur 3 représentant la case sur laquelle l'obstacle est placé.
   - Si l'obstacle a une résistance supérieure ou égale à 3, la chaîne produite est "@@@".

— Si l'obstacle a une résistance de 1, on n'affiche qu'un symbole `'@'`, entouré du premier et du dernier caractères de `background`. Dans ce cas on doit donc obtenir `" @ "` avec l'appel `toString("  ")` et `"(@)"` avec l'appel `toString("( )")`.
— Similairement, si l'obstacle a une résistance de 2 on forme une chaîne avec deux symboles `'@'` et un symbole de `background`.

Entités mobiles Les personnages et les monstres sont deux cas particuliers d'entités mobiles. Chacun possède une direction (susceptible d'évoluer avec le temps).

8. Créer une classe abstraite `EntiteMobile`.
9. Créer les classes `Personnage` et `Monstre`. Chacune doit posséder une version concrète de la méthode `String toString(String background)` renvoyant une chaîne formée d'un symbole central entouré du premier et du dernier caractères de `background`. Le symbole central dépend du type de créature et de sa direction :

	`Personnage`	`Monstre`
nord	`'^'`	`'m'`
sud	`'v'`	`'w'`
est	`'>'`	`'»'`
ouest	`'<'`	`'«'`

   On doit donc renvoyer par exemple :
   — `" < "` pour un personnage tourné vers l'ouest sur un fond `"  "`, et
   — `"(w)"` pour un monstre tourné vers le sud sur un fond `"( )"`.
   Note : les affichages des personnages et des monstres étant similaires, il est possible (mais non obligatoire) de partager une partie du code à écrire dans la classe `EntiteMobile`.
10. Modifier les méthodes `toString` des classes des cases susceptibles d'accueillir une entité de sorte que, lorsqu'une case possède effectivement un contenu, on affiche la combinaison produite par les méthodes `String toString(String background)`.

Note : à ce stade, vous pouvez utiliser le constructeur fourni dans la classe `Terrain`, qui initialise un terrain à partir d'une description donnée par un fichier texte (voir `laby1.txt`).

Actions Un tour de jeu consiste à faire agir une créature tirée au hasard parmi les créatures présentes. Les créatures obéissent aux règles suivantes.

— Un `Personnage` sur une `Sortie` sort du jeu (et on incrémente un compteur des personnages sauvés).
— Un `Personnage` ou un `Monstre` face à une case libre avance.
— Un `Personnage` face à un `Obstacle`, décrémente la résistance de l'`Obstacle` de 1.
— Un `Monstre` face à un `Obstacle` ou un `Personnage` décrémente la résistance de sa cible de 1.
— Dans tous les autres cas : changement de `Direction` aléatoire.

Un `Personnage` ou un `Obstacle` dont la résistance tombe à zéro disparaît du jeu.

11. Faire que chaque `EntiteMobile` contienne une méthode **`void`** `action(Case courante, Case cible)`, qui fait agir la créature selon le descriptif ci-dessus. On suppose que `courante` est la case occupée par la créature, et `cible` la case qui se trouve devant elle. Note : cela nécessite également d'inclure un moyen de décrémenter la résistance d'une `Entite`, et de détecter les cas où une `Entite` doit être retirée du jeu.
12. Écrire dans la classe `Jeu` une méthode **`void`** `tour()` qui sélectionne de manière aléatoire une case contenant une créature, et fait agir cette créature. Ajouter une méthode `main` qui enchaîne les tours de jeu et affiche le terrain à chaque étape.

Extensions

13. Faire qu'à chaque tour, chaque créature joue exactement une fois.
14. Arrêter le jeu lorsque tous les personnages sont sauvés ou mangés.
15. Ajouter à volonté d'autres types de cases et d'autres types de créatures.

IPO – TP-projet : des moutons dans le donjon

http://www.lri.fr/~blsk/IPO/

Un berger et son troupeau se sont abrités de la pluie dans un vieux château. Se rendant compte que le donjon abritait des loups, le berger cherche à faire ressortir ses moutons. Le voilà fort affairé : il doit tout à la fois guider les moutons vers la sortie et s'interposer entre eux et les loups.

Objectif du TP Ce TP-projet est à réaliser en binôme. Vous y reprenez l'ensemble des éléments réalisés lors du TP 9 (classes abstraites), et les intégrez dans un jeu à un joueur en temps réel. L'ensemble formé par ces deux TP sera à compléter pour le 4 décembre (23h59) et donnera lieu à une soutenance la semaine du 5 septembre. Le tout formera votre note de contrôle continu. L'ensemble de votre projet doit être versionné avec `Git`, et vous devez donner à votre enseignant l'accès à votre dépôt.

Vous trouverez sur la page du cours une archive `.jar` contenant une version de démonstration minimale de ce jeu. Pour lancer cette démonstration, vous devez également télécharger le fichier `laby2.txt` et le placer dans le même répertoire que le `.jar`. Puis double-clic sur `moutons.jar`, ou dans un terminal entrez la ligne `java -jar moutons.jar &`.

Description Le donjon est matérialisé par la classe `Terrain` et ses différentes sortes de `Cases`. Les moutons sont représentés par la classe `Personnage` et les loups par la classe `Monstre`. Tous ces éléments peuvent être repris intégralement, sans modification par rapport à ce qui était demandé au TP9.

Le berger est représenté par un nouveau sous-type d'`Entite`. Comme les autres entités, il peut se trouver sur n'importe quelle case traversable, et dispose d'une résistance. Comme les autres entités, il est un obstacle au déplacement des moutons et des loups :

— un mouton se trouvant face au berger changera de direction,

— un loup se trouvant face au berger l'attaquera, faisant décroître sa résistance.

Le jeu présente au joueur une vue de l'ensemble du donjon, mise à jour après chaque `tour` (10 fois par seconde). Le joueur y déplace le berger en temps réel à l'aide des flèches du clavier.

Travail à réaliser Vous trouverez sur la page du cours un fichier principal `Donjon.java` (complet), un squelette d'interface graphique `FenetreGraphique.java` (à compléter), et un nouvel exemple de description de terrain (`laby2.txt`), à combiner avec vos fichiers du TP9. Voici une liste des tâches à réaliser, dans l'ordre.

1. Créer un dépôt `Git` par binôme pour gérer votre projet, et placez-y tous les fichiers (instructions détaillées page suivante).
2. Compléter dans la classe `FenetreGraphique` les méthodes d'affichage, de sorte à pouvoir observer le comportement de votre jeu, sans interaction.
3. Créer une classe `Joueur` étendant `Entite`, pour représenter le berger. Dans le constructeur de la classe `Terrain`, étendre le décodage du fichier d'entrée pour qu'il interprète un `H` comme la position de départ du joueur (vous pouvez maintenant utiliser le fichier `laby2.txt`).
4. Ajouter à la définition de la classe `FenetreGraphique` l'annotation **`implements`** `KeyListener`, et étendre cette classe avec les méthodes nécessaires, de sorte que les flèches du clavier permettent de déplacer le joueur. *Note :* l'interface `KeyListener` a un fonctionnement similaire à `MouseListener` (mais c'est à vous de trouver le détail des méthodes à implémenter !)
5. Ajouter au berger la possibilité de sortir lui aussi, en appuyant sur la touche « espace » lorsqu'il se trouve sur une `Sortie`.
6. Ajouter à la classe `Jeu` une méthode testant si la partie est finie :
   — le jeu s'arrête lorsque le berger sort, avec un score égal au nombre de moutons sauvés,
   — le jeu s'arrête également lorsque le berger est mangé, avec un score divisé par deux.
7. Ajouter, à volonté, de nouveaux éléments pour enrichir le jeu.

Instructions pour l'initialisation de `Git`

Création du projet lui-même, à faire par l'un des membres du binôme.

1. Allez sur l'interface web du `gitlab` de l'université :
   `https://gitlab.dsi.universite-paris-saclay.fr/`
   et connectez-vous (avec vos identifiants habituels de l'université).
2. Cliquez en haut à droite : *new project*, puis sélectionnez *create blank project*.
3. Donnez un nom à votre projet. À la ligne *project URL*, assurez-vous que votre nom apparaît, ou allez le sélectionner en cliquant sur *pick a group or namespace*.
4. Assurez-vous que le niveau de visibilité *private* est sélectionné, puis confirmez la création.

Sélection des autres participants, à réaliser dans l'interface web du projet.

1. Dans le menu, sélectionnez *project information*, puis *members*.
2. Cliquez sur *invite member* et intégrez le deuxième membre du binôme, avec le rôle *maintainer*.
3. Recommencez et intégrez votre encadrant(e) de TP, avec un rôle *reporter* (ou supérieur), puis de même avec `thibaut.balabonski`.

Note : vous ne pouvez inviter que quelqu'un qui s'est déjà connecté à ce `gitlab` un jour. Si vous ne trouvez pas la personne dans la liste, demandez-lui de se connecter.

Instructions pour connecter `Git` et `IntelliJ IDEA`

Import du projet dans `IntelliJ`, à faire par chaque membre du groupe et sur chaque ordinateur utilisé.

1. Dans `IntelliJ`, allez sélectionner dans le menu *file*, *new*, puis *project from version control*.
2. Vérifiez que `Git` est sélectionné dans la ligne *version control*, puis indiquez l'URL du projet (l'adresse de la page principale de votre projet sous `gitlab`, que vous pouvez aller copier depuis la barre d'adresse). Vous pouvez aussi depuis ce menu sélectionner le dossier dans lequel se trouvera le projet.
3. Entrez vos identifiants pour autoriser `IntelliJ` à se connecter à votre compte `gitlab`, puis validez.

Initialisation des fichiers, à faire par l'un des membres du groupe.

1. Dans le projet `IntelliJ`, faites un clic droit *new*, *directory*, et nommez `src` le dossier créé.
2. Faites un clic droit sur `src`, puis sélectionnez *mark directory as*, et *sources root*.
3. Ajoutez dans le dossier `src` vos fichiers `.java`. Si l'interface vous propose des les ajouter à `Git`, acceptez.
4. Poussez les nouveaux fichiers sur le dépôt `Git` : dans le menu *git*, allez chercher les actions *commit* et *push* (l'action *commit* ouvrira dans l'interface une fenêtre où vous devez écrire un message décrivant le contenu de la modification que vous enregistrez).

Instructions pour travailler une fois `Git` en place

À réaliser par chaque personne travaillant sur le projet.

1. Au début de chaque session de travail, récupérez les fichiers du dépôt partagé pour mettre à jour vos fichiers locaux avec `pull`. Pour cela : cliquez dans le menu *git*, puis *pull*. Recommandation : c'est généralement une bonne idée, dans *modify options*, d'allez sélectionner `--rebase`.
2. Après chaque modification significative, enregistrez (localement) votre travail avec `commit`. Dans le menu *git*, sélectionner *commit*, et entrer un message décrivant la modification.
3. À la fin de chaque séance (voire plus souvent), publiez vos *commits* sur le dépôt partagé avec `push`. Dans le menu *git*, sélectionner *push*, et valider.

Pour éviter les interférences entre les modifications faites par plusieurs personnes travaillant en parallèle sur le même projet, il vaut mieux souvent publier ses modifications (`commit`/`push`) et récupérer les modifications des partenaires (`pull`).

Note : l'historique des *commit*s est daté, et permettra à votre encadrant d'apprécier la régularité de votre travail.

IPO – TD 7 : préparation avant partiel

https://www.lri.fr/~blsk/IPO/

1 Questions de cours

Structure d'une classe Soit le code suivant :

```
1  public class User {
2    private int age ;
3    private String username ;
4    private String role ;
5
6    public User(int age, String username, String role) {
7      this.age = age ;
8      this.username = username ;
9      this.role = role ;
10   }
11
12   public User(int age, String username) {
13     this(age, username, "suscriber") ;
14   }
15
16   public int getAge() {
17     return age ;
18   }
19
20   public void setAge(int age) {
21     this.age = age ;
22   }
23
24   public String getUsername() {
25     return username ;
26   }
27
28   public boolean isAdult() {
29     return age >= 18 ;
30   }
31
32 }
```

1. Désigner par leurs numéros de lignes les parties du code correspondant aux déclarations d'attributs, aux constructeurs et aux méthodes.
2. Quel est le nom de la classe ? Le nom complet du fichier ?
3. Que signifie le mot-clé **`private`** à la ligne 2 ?
4. Écrire une instruction permettant de créer une instance de cette classe.
5. Écrire une instruction permettant d'en changer l'âge à 25.
6. Que signifie le mot-clé **`this`** à la ligne 7 ?
7. Que signifie le mot-clé **`this`** à la ligne 13 ?

Convention de style

8. On vous demande d'écrire un programme gérant un jeu se déroulant en plusieurs manches. Parmi ceux-ci, quel nom de variable choisir pour désigner le score d'un joueur pour une manche ? Pourquoi ?
   - `score_d_une_manche`
   - `nb`
   - `currentpoints`
   - `scorePlayerRound`
   - `MonScoreCetteManche`

Debug Lors du test d'un programme, le message suivant s'affiche :

```
Exception in thread "main" java.lang.NullPointerException
  at cours.Mail.send (Mail.java:7)
  at cours.Mail.main (Mail.java:14)
```

9. Ce message est-il susceptible de se produire lors de la compilation, ou lors de l'exécution du programme compilé ?
10. Quel est le souci ? Que faudrait-il aller regarder dans le code et où ? Comment le corriger ?

Fonctions Soit une classe `Equation` contenant la méthode

```
public static Double solver(String equation, ArrayList<Double> parameters)
```

11. Comment appeler cette méthode depuis une autre classe ?

2 Compréhension de code et programmation

Nous allons ici gérer l'historique des scores pour un jeu de style arcade. À chaque partie, un nom de joueur est entré. À la fin de chaque partie, un score est enregistré, associé au nom du joueur. On veut ensuite pouvoir retrouver le meilleur score de chaque joueur et le meilleur score général. Le contenu du jeu ne vous est pas demandé et n'a pas d'importance pour cet exercice, on a juste besoin de savoir que chaque partie aboutit à un score positif.

On se donne les trois squelettes de classes montrés en page annexe.

Questions

12. Ligne 5 de `Historique.java`, pourquoi le type `Integer` est-il utilisé dans la structure de `historiqueScores` plutôt que le type **`int`** ?
13. Que représente l'attribut `historiqueScores` défini ligne 5 de `Historique.java` ? Quelles données contiendra-t-il ?
14. Décrire dans quelles situations chacun des cas du **`if`** (ligne 13 et 17 de `Historique.java`) se produiront.
15. Ligne 22 de `Historique.java`, que signifie le mot-clé **`static`** ?
16. Ligne 22 de `Historique.java`, que signifie le mot-clé **`private`** ?
17. À quoi sert la méthode `max` de `Historique.java` (lignes 22 à 33) ?
18. Dans quel cas la méthode `max` de `Historique.java` pourrait-elle lancer une exception ? Cela peut-il se produire pour des données issues de notre historique de scores, si l'on suppose que l'on y ajoute des données uniquement par la méthode `ajoutScore` ?
19. Écrire une méthode **`public`** `Integer trouverHighScore(String joueur)`, dans la classe `Historique`, qui prend le nom d'un joueur et renvoie son meilleur score.
20. Écrire une méthode **`public`** `ArrayList<String> getListeJoueurs()` dans la classe `Historique`, qui renvoie la liste des noms de tous les joueurs ayant déjà enregistré un score.

    Indication : Vous pouvez utiliser la méthode `Set<K> keySet()` de `HashMap<K,V>`.
21. Écrire une méthode `HashMap<String, Integer> highScoreParJoueur()` dans la classe `Historique`, qui renvoie une `hashMap` associant à chaque joueur son meilleur score.
22. Écrire une méthode **`public`** `Integer trouverHighScoreGlobal()` dans la classe `Historique`, qui renvoie le meilleur score parmi tous les joueurs.
23. Dans la classe `Lanceur`, écrire un `main` contenant un boucle infinie qui, à chaque tour : demande d'entrer un nom, lance une partie, enregistre le score, et enfin affiche le meilleur score du joueur et le meilleur score global.
24. Proposer une solution pour ne pas avoir à recalculer le meilleur score global à chaque fois.

```java
import java.util.ArrayList;
import java.util.HashMap;

public class Historique {
  private HashMap<String, ArrayList<Integer>> historiqueScores;

  public Historique() {
    this.historiqueScores = new HashMap<>();
  }

  public void ajoutScore (String joueur,  int score) {
    ArrayList<Integer> scoresJoueur = historiqueScores.get(joueur);
    if (scoresJoueur == null) {
      scoresJoueur = new ArrayList<>();
      scoresJoueur.add(score);
      historiqueScores.put(joueur, scoresJoueur);
    } else {
      scoresJoueur.add(score);
    }
  }

  private static Integer max(ArrayList<Integer> tab) {
    if (tab == null || tab.isEmpty()) {
      return null;
    }
    Integer max = tab.get(0);
    for (Integer i : tab) {
      if (i > max) {
        max = i;
      }
    }
    return max;
  }
}
```

```java
import java.util.Scanner ;

public class Lanceur {

  public static String entrerNom (Scanner sc) {
    System.out.println ("Entrez votre nom");
    return sc.nextLine();
  }

  public static void main (String[] args) {
    Scanner sc = new Scanner (System.in);

    //TODO Question 23
  }

}
```

```java
public class Partie {

  public Partie(String joueur) { ... }

  public int jouer() { ... }

}
```

Correction

Exercice 1

1. Attributs : 2, 3, 4. Constructeurs : 6-10, 12-14. Méthodes : 16-30.
2. Classe `User` dans fichier `User.java`.
3. L'attribut age n'est pas accessible depuis l'extérieur de cette classe.
4. `User u = new User(99, "Toto", "plaisantin");`
5. `u.setAge(25);`
6. Précise que l'on parle de l'attribut age de l'objet courant.
7. Appel à un (autre) constructeur de la même classe.
8. `scorePlayerRound` : suite de mot décrivant bien le contenu, écrit en « camelCase ».
9. Il s'agit d'une exception. Celle arrive uniquement à l'exécution.
10. Le code de la méthode `send` (dans la classe `Mail`, fichier `Mail.java` du package `cours`) contient probablement un accès à un champ d'un objet inexistant. Aller vérifier la provenance de l'objet en question, et s'assurer qu'il est bien initialisé, ou au contraire ajouter à la méthode concernée un test pour modifier son comportement lorsque l'objet n'est pas défini.
11. `Equation.solver(..., ...)`

Exercice 2

12. `ArrayList` doit être paramétré par un nom de classe, et pas par un type de base, d'où emploi du *wrapper* `Integer` des entiers **`int`**.
13. Table d'association, qui à chaque nom de joueur associe la liste des scores réalisés (sous la forme d'un tableau redimensionnable).
14. Première branche si la liste de scores du joueur n'a pas encore été créée, ce qui arrive s'il n'y a pas encore d'entrée pour ce joueur, c'est-à-dire si l'on est en train d'enregistrer son premier score. Deuxième branche à partir du deuxième score, où le tableau du joueur existe déjà.
15. La méthode `max` est indépendante de toute instance de la classe `Historique`. Il s'agit d'une simple « fonction » définie dans cette classe.
16. La méthode `max` n'est pas accessible depuis l'extérieur de la classe `Historique`.
17. Méthode auxiliaire de recherche du maximum d'un tableau d'entiers. Renvoie **`null`** en cas de tableau vide ou non défini (ne peut arriver que pour un joueur qui n'a encore enregistré aucun score).
18. Peut déclencher une exception si l'un des scores du tableau est égal à **`null`**. La méthode `ajoutScore` ne peut pas ajouter un tel score indéfini (elle prend en paramètre un entier primitif, et non un objet `Integer`).
19.
```
1  public Integer trouverHighScore(String joueur) {
2    ArrayList<Integer> scores = historiqueScores.get(joueur);
3    return Historique.max(scores);
4  }
```

```
1  public ArrayList<String> getListeJoueurs() {
2    ArrayList<String> joueurs = new ArrayList<>();
3    for (String joueur : historiqueScores.keySet()) {
4      joueurs.add(joueur);
5    }
6    return joueurs;
7  }
```

20.
```
1  public HashMap<String, Integer> highScoreParJoueur() {
2    HashMap<String, Integer> hs = new HashMap<>();
3    for (String joueur : historiqueScores.keySet()) {
4      hs.put(joueur, trouverHighScore(joueur));
5    }
6    return hs;
7  }
```

```java
public Integer trouverHighScoreGlobal() {
  ArrayList<Integer> highScores = new ArrayList<>();
  for (String joueur : historiqueScores.keySet()) {
    highScores.add(trouverHighScore(joueur));
  }
  return Historique.max(highScores);
}
```

23.

```java
Historique historique = new Historique();

while (true) {
  String nom = Lanceur.entrerNom(sc);
  Partie partie = new Partie(nom);
  int score = partie.jouer();
  historique.ajouteScore(nom, score);
  System.out.println("Meilleur score du joueur : "
                     + historique.trouverHighScore(nom));
  System.out.println("Meilleur score global : "
                      + historique.trouverHighScoreGlobal());
}
```

24. Il suffit d'ajouter un attribut `Integer highScoreGlobal` à la classe `Historique`, *ET* de modifier la méthode `ajouteScore` pour qu'elle teste si le nouveau score bat le record global, et le mette à jour le cas échéant.

IPO, TD, annale examen 2021

1 Questions de cours

Question 1 En supposant les noms bien choisis, associer les quatre noms suivants à leur rôle (interface, classe, méthode ou variable) :
`Furniture`
`animal`
`Deletable`
`driveStraight`

Question 2 Soit la classe Vehicule :

```
1  public class Vehicule {
2     protected String immatriculation;
3     private int puissance;
4
5     public Vehicule(String immatriculation, int puissance) {
6        this.immatriculation = immatriculation;
7        this.puissance = puissance;
8     }
9
10    public String getImmatriculation() {
11       return immatriculation;
12    }
13
14    public void setImmatriculation(String immatriculation) {
15       this.immatriculation = immatriculation;
16    }
17
18    public int getPuissance() {
19       return puissance;
20    }
21 }
```

et la classe Voiture :

```
1  public class Voiture extends Vehicule {
2     private int nbPortes;
3
4     public Voiture(String immatriculation, int puissance, int nbPortes) {
5        super(immatriculation, puissance);
6        this.nbPortes = nbPortes;
7     }
8
9     public int getNbPortes() {
10       return nbPortes;
11    }
12 }
```

Après les déclarations suivantes (dans un `main`) :

```
1        Vehicule moto = new Vehicule("ZZZ", 4);
2        Voiture voiture = new Voiture("ABC", 6, 5);
3        Vehicule def = new Voiture("DEF", 6, 3);
```

que donneraient les instructions suivantes ? Expliquer brièvement pourquoi.

1. `System.out.println(voiture.getImmatriculation());`
2. `System.out.println(voiture.getNbPortes());`
3. `System.out.println(moto.getImmatriculation());`
4. `System.out.println(moto.getNbPortes());`
5. `System.out.println(def.getImmatriculation());`
6. `System.out.println(def.getNbPortes());`
7.
```
Vehicule[] mesVehicules = {moto, voiture, def};
for (int i =0; i<3; i++){
   System.out.println(mesVehicules[i].getPuissance());
}
```

Question 3 Soient les interfaces et classe suivantes :

```
1  public interface Identifiable {
2     String getIdentite();
3  }
```

```
1  public interface AppelableParTelephone {
2     String getTelephone();
3  }
```

```
1  public interface Localisable {
2     String getAdresse();
3  }
```

```
1  public abstract class IdentiteBancaire implements Identifiable {
2     protected final String rib;
3     protected String nom;
4     private double fonds;
5
6     public IdentiteBancaire(String rib, String nom) {
7        this.rib = rib;
8        this.nom = nom;
9        this.fonds = 0;
10    }
11
12    public abstract String procedureDeDemarchage();
13
14    public void crediter(double montant){
15       fonds += montant;
16    }
17
18    public boolean debiter(double montant){
19       if (montant > fonds){
20          return false;
21       }
22       fonds -= montant;
23       return true;
24    }
25
26 }
```

Écrire une classe `Societe` minimale héritant d'`IdentiteBancaire` et implémentant `AppelableParTelephone` et `Localisable`

Question 4 Quel(s) intérêt(s) voyez-vous à utiliser Git pour un projet que l'on réalise seul ?

2 Création et composition de classes : Gestion de stock (11 points)

On cherche ici à développer une ébauche de gestion de stock d'une application de vente pour un magasin. On crée pour cela des produits, qui vont être associés à des quantités pour apparaître dans l'inventaire du magasin ou dans les achats des clients. On distingue un type de produit particulier : les aliments, qui ont une date de péremption (la même pour tout le lot).

Le code de la classe `Produit`, le squelette de la classe `Achat` et un exemple de main de test vous sont fournis en annexe 2.

Un produit a toujours une désignation (le nom de l'article), un prix, et un code produit qui l'identifie sans ambiguïté. Il peut posséder une description.

Question 12 Écrire le constructeur `public Produit(String designation, String description, String codeProduit, String prixStr)` de la classe `Produit`. Ce constructeur doit appeler le constructeur principal en convertissant le prix donné sous forme de chaîne de caractère en double, via la méthode de le classe `Double` : `public static double parseDouble(String s)`.
Nous supposons la chaîne au bon format (c'est-à-dire qu'il n'est pas demandé de traiter les cas d'erreurs).

Question 13 Écrire une classe `ProduitEnQuantite` qui devra contenir :

- un attribut `produit`, donnant un référence vers le produit concerné
- un attribut entier `quantite`
- un constructeur, prenant un produit et un quantité en paramètre
- des getters pour produit et quantite

Question 14 Écrire, pour la classe `ProduitEnQuantite` les méthodes suivantes :

- `public void setQuantite(int quantite)` qui fixe la quantité du produit à celle indiquée, à condition que celle-ci soit positive. Sinon, la quantité est fixée à 0.
- `public void ajouterAuStock(int nb)` qui ajoute `nb` à la quantité disponible
- `public boolean vendre(int nb)` qui retire `nb` à la quantité disponible s'il y en a assez et renvoie vrai, sinon ne fait rien et renvoie faux.

Question 15 Quelle est l'utilité d'avoir à la fois un setter et des méthodes pour ajouter et diminuer la quantité ?

Question 16 Écrire un deuxième constructeur pour `ProduitEnQuantite`, prenant un unique paramètre de type `Produit` en en fixant la quantité à 1 par défaut. Ce constructeur doit appeler le constructeur précédemment défini.

Les questions qui suivent vous demandent de définir votre classe `Inventaire` avec vos propres choix d'implémentation et de structures. Vous serez notés sur la justesse de votre code mais également sur la pertinence de vos choix.
Conseil : l'énoncé des questions 17 à 19 peut vous aider à comprendre ce que l'on vous demande et donc à faire des choix d'implémentation pertinents.

Question 17 Écrire une classe inventaire, qui doit contenir un ensemble de produits en stock. Elle doit contenir un constructeur, une méthode pour y ajouter en produit en une quantité voulue et une méthode pour vendre (c'est à dire retirer du stock) un produit, dans une quantité indiquée.

Question 18 Écrire une méthode `purge` qui supprime de l'inventaire tous les produits présents avec une quantité de 0.

Question 19 Écrire la méthode `public String toString()` qui affiche l'état courant de l'inventaire, en listant tous les produits présents, suivis de leurs quantités.
Un rendu possible :

```
trombone, 0.38euros, 4 en stock
stylo, un stylo noir à bille, 0.56euros, 10 en stock
ardoise, 2.78euros, 5 en stock
```

Nous allons maintenant écrire une classe `Aliment` représentant un type de produit particulier ayant une date de péremption.

Question 20 Écrire une classe `Aliment` qui hérite de `Produit`. Elle y ajoute un champ textuel (de format non précisé) `dateDePeremption`. Y inclure un constructeur, ainsi que la méthode `public String toString()` qui ajoute la date de péremption à la suite des informations sur le produit.

Question 21 Avez-vous besoin d'ajouter autre chose à la classe Aliment ou d'adapter le reste de vote code ? Comment se comporte la méthode `toString` d'`Inventaire` lorsque le stock comprend à la fois des produits alimentaires et non-alimentaires ?

Nous nous intéressons maintenant à la création d'une classe `Achat`, qui permet de faire diminuer le stock en fonction de l'achat de plusieurs produits en une fois (lors de passages en caisse). Un squelette vous est donné en annexe 2.

Question 22 Écrire la méthode `public boolean ajouteProduit(Produit p, int quantite)` permettant d'ajouter un produit à l'achat en une certaine quantité. On ne vérifiera pas si le même produit est déjà présent dans le contenu de l'achat. En revanche, l'appel à cette méthode doit réduire de la quantité indiquée le stock du produit correspondant dans l'inventaire. La méthode ne fait rien et renvoie faux dans le cas où le stock serait insuffisant.

Question 23 Écrire la méthode `public double getPrix()` qui calcule le prix total de l'achat.

Question 24 Écrire la méthode `public void annule()` qui remet en stock tous les produits du contenu puis vide le contenu de l'achat.

Question 25 Avez-vous des remarques sur vos choix d'implémentation ? Quels points supplémentaires faudrait-il développer ?

3 Annexe 1 : squelettes pour la gestion de stock

Produit.java

```
1  package vente;
2
3  public class Produit {
4      private String designation;
5      private String description;
6      private final String codeProduit;
7      private double prix;
8
9      public Produit(String designation, String description,
10                    String codeProduit, double prix) {
11         this.designation = designation;
12         this.description = description;
```

```
13          this.codeProduit = codeProduit;
14          this.prix = prix;
15      }
16
17      public Produit(String designation, String description,
18                          String codeProduit, String prixStr) {
19          //TODO Question 12
20      }
21
22      public double getPrix() {
23          return prix;
24      }
25
26      public void setPrix(double prix) {
27          this.prix = prix;
28      }
29
30      public String getCodeProduit() {
31          return codeProduit;
32      }
33
34      public String getDesignation() {
35          return designation;
36      }
37
38      @Override
39      public String toString() {
40          //TODO Question
41          return designation + ", " +
42                  ((description == null)?"":(description + ", ")) +
43                  prix + "euros";
44      }
45  }
```

Achat.java

```
1   package vente;
2
3   import java.util.ArrayList;
4   import java.util.List;
5
6   public class Achat {
7       private List<ProduitEnQuantite> contenu;
8       private Inventaire inventaire;
9
10      public Achat(Inventaire inventaire){
11          contenu = new ArrayList<>();
12          inventaire = inventaire;
13      }
14
15      public boolean ajouteProduit(Produit p, int quantite){
16          //TODO Question 22
17      }
18
19      public double getPrix(){
```

```
20          //TODO Question 23
21      }
22
23      public void annule(){
24          //TODO Question 24
25      }
26  }
```

Un main, ici dans Inventaire.java :

```
1   package vente;
2
3   public class Inventaire {
4
5           //TODO Questions 17-18-19
6
7       public static void main(String[] args) {
8           Produit trombone = new Produit("trombone", null, "T438", 0.38);
9           Produit stylo = new Produit("stylo", "un stylo noir à bille", "S003", 0.56);
10          Produit ardoise = new Produit("ardoise", null, "A768", 2.78);
11
12          Inventaire stockDuMagasin = new Inventaire();
13
14          stockDuMagasin.ajouter(trombone, 4);
15          stockDuMagasin.ajouter(stylo, 10);
16          stockDuMagasin.ajouter(ardoise, 5);
17
18          Produit yaourt = new Aliment("yaourt", "4 pots de yaourt à la fraise",
19                  "2Y45", 0.73, "17-01");
20
21          stockDuMagasin.ajouter(yaourt, 30);
22          Achat unClient = new Achat(stockDuMagasin);
23          unClient.ajouteProduit(yaourt, 1);
24          unClient.ajouteProduit(stylo, 2);
25          unClient.ajouteProduit(trombone, 26);
26
27          System.out.println(stockDuMagasin);
28      }
29  }
```

La sortie de ce programme est :

```
yaourt, 4 pots de yaourt à la fraise, 0.7euros, date de péremption : 17-01, 29 en stock
trombone, 0.38euros, 4 en stock
stylo, un stylo noir à bille, 0.56euros, 8 en stock
ardoise, 2.78euros, 5 en stock
```