Programmatin Objet et Génie Logiciel, Cours de F. Gruau 2023—2024.

Source Ce cours reprends les notes de Thibaut Balabonsky et les slides de Delphine Longuet. Lorsque les commentaires des slides sont dispo sur utube, nous vous donnons le lien pour y accéder. Vous pouvez copier coller ce lien depuis le pdf.

Plan du cours Chaque cours traite en général :

1. Du Génie Logiciel, avec des diagrammes UML
2. Une technique objet et sa mise en oeuvre en Java.
3. Une structure de donnée Tableau 2D, Liste, Liste doublement chainée, table de hashage, graphes...

Chacun de ses trois aspects et ensuite repris dans le TD ou TP qui suit le cours. Le cours est synchronisé avec les TD/TP, on peut le voir comme une préparation au TD/TP. Dans ce poly, chaque section corresponds a l'une des ces 12 semaines :

1. TP Revision, utilisation d'une interface graphique
2. TP Réaliser une interface générique de 2 façons
3. TD Analyse et spécification via des diagrammes UML.
4. TP Conception architecturale via des diagrammes UML
5. TP Polymorphisme : objet ayant plusieurs type possible
6. TP Développement guidé par les tests
7. TD Conception/spécification d'opération
8. TD Démarrage projet. Donner/Expliquer le matériel
9. TP Les classes internes pour un code clair et compact.
10. TD Liaison dynamique et Transtypage
11. TP Bonne pratique pour gros programmes.
12. cours-TD sur les exception / Soutenance de Projet

1 Revisions, Exemple simple, Interface Graphique

1.1 Revision

1.1.1 Partie de java non-objet

Types et opérations de base Quelques types de base de Java :

Types	Contenu	Variantes
`boolean`	Vrai/Faux	
`int`	Entiers	`byte`, `short`, `long` (± grands)
`double`	Décimaux	`float` (moins précis)
`char`	Caractères	

Quelques opérations :

Famille	Applicabilité	Opérations
Opérations logiques	Booléens	`&&`, `||`, `!`
Arithmétique	Tous les nombres	`+`, `-`, `*`, `/`
Modulo	Entiers	`%`
Incréments	Nombres entiers	`++`, `--`
Égalité	Tous	`==`, `!=`
Comparaison	Nombres et caractères	`>`, `>=`, `<`, `<=`

Tableaux Le type des tableaux de booléens est `boolean[]`, celui des tableaux de tableaux de nombres entiers est `int[][]`. Opérations principales :

- `int[] tab = new int[5]` crée un tableau appelé `tab`, de taille 5, et destiné à contenir des nombres entiers. Les indices de ses cases vont de 0 à 4.
- `tab[2]` renvoie le contenu de la case d'indice 2 du tableau `tab` (c'est-à-dire la troisième case). Si le tableau était à double entrée on pourrait utiliser `tab[1][3]` pour accéder à la case de la deuxième ligne, quatrième colonne.
- `tab[2] = 4` met l'entier 4 dans la case d'indice 2 du tableau `tab`.

Structures de contrôle Branchement conditionnel, pour choisir entre deux comportements en fonction du résultat d'un test.

-
```
if  ( this.estPiegee ) {
    setBackground(Color.RED);
} else {
    int nb = ...
}
```
- `if  ( grille[x][y].estPiegee() )  { nb++; }`

Boucle ‘ pour répéter une opération un nombre déterminé de fois. Dans les exemples suivants, on considère un tableau `tab` contenant 10 chaînes de caractères (son type est `String[]`).

- Boucle “pour :”
```
for  (int i=0; i<10; i++)  {
    System.out.println( tab[i] );
}
```
- boucle “for each :”
```
for  (String s : tab)  {
    System.out.println( s );
}
```

1.1.2 Partie de java orienté objet.

Classes, attributs, méthodes En programmation orientée objet, la déclaration d'une classe regroupe des membres, méthodes et propriétés (attributs) communs à un ensemble d'objets. La classe déclare, d'une part, des attributs représentant l'état des objets et, d'autre part, des méthodes représentant leur comportement.

- Les *attributs* ce sont les données associées à un objet. Les attributs sont généralement déclarés comme privés (`private`) pour les rendre inaccessibles aux autres objets.
```
private  boolean   estPiegee
private  double       re, im
private  double[]  mesures
```

— Les *méthodes* : ce sont les actions que peut effectuer un objet. Les méthodes sont généralement déclarées comme publiques (`public`) pour permettre aux autres objets d'y faire appel.

```
public  void  clicDroit()            { ... }
public int compteMines(int i, int j) { ... }
public Complexe somme(Complexe c) { ... }
```

— Une méthode dont le type de retour est autre que `void` (par exemple, `compteVoisines` renvoie un résultat de type `int`) renvoie son résultat avec l'instruction `return`.

```
return nb;
```

— Des *constructeurs* : des méthodes particulières, sans type de retour, qui permettent de créer un objet.

```
public Complexe(double x, double y){ ... }
public Terrain(int h, int l, double s){ ... }
```

Un fichier de code Java peut contenir plusieurs classes, mais une seule d'entre elles est "publique", de nom correspondant, introduite par la mention public class C { ... } où C est le nom donné à la classe.

Objets Un nouvel objet est créé avec l'instruction `new`, suivie d'un constructeur de la classe associée.

```
— Complexe c=new Complexe(2,3);
— Terrain  terrain  =  new  Terrain(6, 8, 0.2);
— Case     c        =  new  Case(b);
```

On accède à un attribut d'un objet (par exemple l'objet courant `this`) en utilisant l'opérateur "point".

```
— this.x
```

On procède de même pour faire appel à une méthode d'un objet, et ces opérations peuvent être combinées.

```
— this.compteMines(i, j);
— this.changeTexte(Integer.toString(
        this.nombreMines));
— this.grille[x][y].estPiegee();
```

Allocation mémoire Lorsque un nouvel ojbet est crée, une zone est allouée en mémoire pour le stoquer. L'objet est utilisé via un pointeur vers cette zone. Par exemple pour un nombre complexe, cette zone contiendra quatre 32bit word, deux pour stoquer la partire réelle (resp. la partie imaginaire). Dans la vrai vie, les classes utilisent d'autres classes, et donc les objets eux-même, contiennent des pointeurs vers d'autres objets, nous verront comment on représente l'état de la mémoire avec un diagramme d'objets.

1.2 L' exemple simple des complexes

Ce premier exemple presente un exemple de la classe complexe le plus simple que l'on puisse imaginer, pour ceux qui n'ont jamais fait de java. Les attributs représentent la partie réelle et la partie imaginaire, et les méthodes sont les opération que l'on peut faire sur les complexes.

La classe Complexe contient la méthode main décrivant le programme principal. Sa signature public static void main(String[]) est fixée par convention, nous détaillerons sa signification au prochain cours.

```
public static  void main(String[]  args)  {
```

Le main est le plus souvent utilisé pour tester les méthodes, au fur et a mesure que on en rajoute.

Avant exécution le programme doit être compilé, c'est-à-dire traduit en langage machine. En l'occurrence, la traduction ne se fait pas vers le langage assembleur de votre machine physique, mais vers le *bytecode* de la machine virtuelle Java (JVM). Cette machine virtuelle simule un ordinateur. Cette technique facilite la diffusion des applications Java : les développeurs n'ont pas besoin de s'adapter qu'à une architecture cible (la JVM), sans se soucier de l'environnement réel des utilisateurs.

Un fichier de code Java porte l'extention .java. Dans un terminal, on demande la compilation avec la commande

```
javac Complexe.java
```

Au passage, cette commande vérifie certains éléments de cohérence du programme, et indique les éventuelles erreurs relevées. Le programme compilé prend la forme d'un nouveau fichier Complexe.class. Dans Intelij, ceci correspond à la commande *Build*.

Dans un terminal on exécute le programme, c'est-à-dire le code de la méthode main, par la commande

```
java Complexe
```

Attention, ceci ne peut se faire qu'en présence du fichier Complexe.class, c'est-à-dire après la compilation. Dans Intelij, ceci correspond à la commande *Run*.

Mutabilité Dans ce premier exemple, on ne modifie jamais un nombre complexe déja crée. On crée toujours un nouveau complexe, dans lequel on met la somme, le produit, le congugué, l'inverse. . . . l'état interne d'un objet complexe reste toujours le meme, une fois crée. On peut dire que la classe Complexe n'est pas mutable. C'est une bonne pratique de travailler avec de telle classe, modifier l'état interne par effet de bord est suceptible de provoquer des erreurs de programmation. Exercice : ecrire les méthodes conjugués et inverse.

```
/** Bibliothéque pour manipuler les  nombres complexes*/
class Complexe {
  //attribut d'instance  déclarés  en private
private double re;  double im;
 //getter
public double getRe(){return re;}
public double getIm(){return re;}

  //le main est utilisé   pour tester  les
  //méthodes, au fur et  a mesure que on en ajoute
public static  void main(String[]  arg)
{ Complexe c=new Complexe(2,3);
  System.out.println(c  + "␣"+c.somme(c)+ "␣"+
          c.produit(2)+ "␣"+c.produit(c)+"␣"+
          c.carreModule()+ "\n"+c.module());
    // +    c.conjugue()+   " "+c.inverse());
}

  /** Construit un Complexe
```

```
   * @param x la partie reelle
   * @param y la partie imaginaire */
  public Complexe(double x, double y) {
    re=x;im=y;
    }

  /** @return representation sous forme de chaine*/
  public String toString() {
    return "("+re+"+"+im+" i";
    }

  /**
   * @param c le Complexe a ajouter
   * @return somme de this avec c */
  public Complexe somme(Complexe c) {
    return (new Complexe(re+c.getRe(), im+c.getIm()));
    }

  /**
   * @param d le double par lequel multiplier
   * @return produit de this avec d. */
  public Complexe produit(double d ) {
    return (new Complexe( re*d, im*d));
    }

  /** @param c le Complexe par lequel multiplier
   * @ produit de this avec c */
  public Complexe produit(Complexe c) {
    return (new Complexe(re*c.getRe() - im*c.getIm(),
                         re*c.getIm() + im*c.getRe()));
    }

  /** @return le carre de la norme de ce Complexe */
  public double carreModule() {
    return re*re + im*im;
    }

  /** @return la norme de ce Complexe */
  public double module() {
    return Math.sqrt(this.carreModule());
    }
}
```

1.3 Utilisation d'une interface graphique.

L'objectif ici est de réviser les principaux éléments de Java que vous avez abordés au premier semestre, à travers l'utilisation d' une interface graphique, en cours puis en TP. On développe pour cela une mini-application JDemine, qui reprend le principe du jeu classique du démineur.

Notre application consiste en trois classes :

- JDemine est la classe principale de l'application,
- Terrain modélise la grille du jeu,
- Case modélise les cases de cette grille.

L'application utilise également un petit paquet IG pour "Interface Graphique" qui donne des éléments minimaux pour interagir avec la bibliothèque graphique "Swing" de Java.

1.3.1 Classe JDemine

Tous les fichiers java commencent par une série de déclarations "import" indiquant les classes extérieures dont ils dépendent (outre les classes définies dans le même *package* qui sont accessibles par défauts). Dans la suite on détaillera classe par classe quelles sont ces déclarations. La classe principale JDemine est définie dans un fichier JDemine.java commençant par déclarer l'utilisation de la classe Fenetre du *package* IG.

```
import IG.Fenetre;

public class JDemine {
```

La classe principale JDemine contient la méthode main décrivant le programme principal.

On définit une variable locale x de type T par la déclaration T x ; On peut en outre affecter une valeur à cette variable en étendant la ligne de la façon suivante T x = ... En l'occurrence, on crée un nouvel objet (on dit aussi une instance) avec le mot clé new, suivi du nom de la classe concernée appliqué à d'éventuels paramètres.

On obtient une fenêtre graphique f intitulée JDemine et un terrain t de hauteur 6 et de largeur 8.

```
    Fenetre f = new Fenetre("JDemine");
    Terrain t = new Terrain(8, 6);
```

On appelle une méthode m d'un objet obj avec une expression de la forme obj.m(...), en fournissant entre parenthèses les paramètres donnés à m.

Les méthodes ajouteElement et dessineFenetre sont définies dans la classe Fenetre. La première permet d'intégrer de nouveaux éléments (ici le terrain t) à l'affichage, et la deuxième provoque l'affichage de la fenêtre et de l'ensemble de ses éléments.

```
    f.ajouteElement(t);
    f.dessineFenetre();
  } // Fin de la methode main

} // Fin de la classe JDemine
```

1.3.2 Classe Terrain

La classe Terrain décrivant le terrain de jeu est définie dans un fichier Terrain.java, et étend la classe Grille du *package* IG.

Lors de la définition d'une nouvelle classe C, on peut déclarer que cette classe étend une classe pré-existante D avec la mention class C extends D { ... }. Dans ce cas, les fonctionnalités de la classe D sont intégrées à la classe C. Les implications de cette déclaration sont plus subtiles qu'il n'y paraît, on reviendra en détail plus tard dans le semestre sur ce que cela signifie exactement.

Ici, Terrain va hériter des fonctionnalités de la classe Grille, c'est-à-dire d'une capacité à intégrer plusieurs éléments graphique disposés selon une grille à deux dimensions.

```
import IG.Grille;

public class Terrain extends Grille {
```

La définition d'une classe contient trois types d'éléments :

- des attributs, qui sont des données attachées à chaque instance de la classe,

— des constructeurs, qui permettent de créer et d'initialiser des instances de la classe,
— des méthodes, qui permettent d'agir sur les instances.

La déclaration d'un attribut a de type T a la forme T a ;s Cette déclaration est généralement précédée d'une indication de visibilité private, indiquant que l'attribut n'est pas accessible depuis les autres classes. On peut déclarer plusieurs attributs du même type à la fois en les séparant par des virgules.

La convention selon laquelle les attributs sont en général privés est l'un des aspects du concept d'encapsulation, qui est central en programmation objet. De manière générale, on distingue dans chaque objet et chaque structure de données des aspects publics définissant la manière dont un utilisateur extérieur peut interagir avec l'objet (on parle pour cela d'interface, essentiellement constituée de méthodes dans le cas de la programmation objet) et des aspects privés liés à la réalisation de l'objet ou de la structure (dont souvent les attributs). L'objectif est d'obtenir une certaine modularité et adaptativité du code : on peut modifier des choses en profondeur à des fins d'amélioration ou de correction de bugs sans que cela n'affecte les utilisateurs tant que l'interface est respectée.

Note sur les conventions de nommage : le code n'étant pas uniquement écrit pour être exécuté, mais aussi pour être lu, on préférera utiliser des noms significatifs. Quand un nom d'attribut est composé de plusieurs mots, on met en majuscule la première lettre de chaque mot (sauf pour le premier mot, seuls les noms de classe commençant par une majuscule).

Ici on définit d'abord deux attributs entiers indiquant la hauteur et la largeur du terrain.

```
private int hauteur, largeur;
```

On note T[] le type des tableaux d'éléments de type T. Le type d'un tableau à deux dimensions est donc noté T[][], avec un parenthésage implicite (T[])[] : on parle d'un tableau dont les éléments sont eux mêmes des tableaux (d'élément de type T).

```
private Case[][] terrain;
```

Un constructeur porte systématiquement le nom de sa classe, a généralement une visibilité public, et peut prendre des paramètres dont il précise les types.

Ici le constructeur de la classe Terrain attend deux paramètres entiers h et l définissant la hauteur et la largeur du terrain à créer.

```
public Terrain(int hauteur, int largeur) {
```

Lorsqu'un classe C étend une classe pré-existante D, les constructeurs de C peuvent commencer par un appel aux constructeurs de D, qui s'écrit super(...) ; Ici, on transmet au constructeur de la classe Grille les dimensions de la grille.

```
    super(hauteur, largeur);
```

Le constructeur est ensuite en charge d'initialiser les attributs de l'objet en cours de création.

Un attribut a de l'objet courant est désigné par this.a. Le this est optionnel mais son usage est recommandé.

À savoir : un attribut non initialisé aura généralement par défaut une valeur équivalente à 0 ou null. On recommande de ne pas se reposer sur ce fait.

```
    this.hauteur  = hauteur;
    this.largeur  = largeur;
```

On crée un tableau de n éléments de type T avec l'expression new T[n]. Si le tableau a plusieurs dimensions, on enchaîne les crochets. Cette déclaration ne créé que le tableau lui-même, et n'initialise pas les éléments qu'il contient.

Ici, on initialise l'attribut terrain avec un tableau de hauteur hauteur et de largeur largeur, destiné à contenir des objets de type Case qui ne sont pas encore définis mais qu'on va ensuite pouvoir créer et ajouter.

```
    this.terrain  = new Case[hauteur][largeur];
```

La boucle inconditionnelle for permet la répétition d'un bloc de code. On y déclare une variable de boucle int i, sa valeur initiale =0, une condition de poursuite de boucle i<hauteur à vérifier avant chaque tour et une instruction de mise à jour de la variable i++ à appliquer à la fin de chaque tour.

Ici, on emboîte deux boucles, dont les indices i et j correspondent aux ordonnées et aux abscisses de notre grille à deux dimensions. On parcourt ainsi chaque ligne du tableau, et chaque élément de chaque ligne, pour initialiser l'ensemble du tableau.

Remarque : indépendamment de nos attributs hauteur et largeur, on peut connaître la longueur d'un tableau t avec l'expression t.length. Dans un tableau de tableaux, on obtient donc la longueur de la i-ème ligne avec t[i].length.

```
    for (int i=0; i<hauteur; i++) {
      for (int j=0; j<largeur; j++) {
```

Définition d'une variable booléenne mine à utiliser dans la construction d'une nouvelle case, indiquant si ladite case doit contenir une mine ou non.

La méthode Math.random() renvoie un nombre flottant de type double, compris entre 0 et 1.

```
        boolean mine = Math.random() < 0.25;
```

Création d'une nouvelle case, par appel au constructeur de la classe Case. On donne en paramètres : les coordonnées, le booléen indiquant si la case doit être piégée ou non, et une référence au terrain lui-même (this) qui permettra à une case de faire appel à des méthodes du terrain.

```
        Case c = new Case(i, j, mine, this);
```

On affecte la valeur d'une expression e à l'indice i d'un tableau t avec une instruction t[i] = e ; En cas de tableaux à plusieurs dimensions on enchaîne les indices entre crochets.

En l'occurrence, la prochaine instruction place la case c à l'indice j de la i-ème ligne de terrain.

```
        terrain[i][j]  = c;
```

On appelle une méthode m de l'objet courant avec l'expression this.m(...)

En l'occurrence, la méthode ajouteElement, qui ajoute la case passée en paramètre à la liste des objets à afficher, est

définie dans la classe Grille, et l'objet courant de classe Terrain y a accès car la classe Terrain étend la classe Grille.

```
            this.ajouteElement(c);
          }
        }

      } // Fin du constructeur de Terrain
```

Alternativement, on aurait pu inclure directement le calcul du booléen à l'appel au constructeur :
Case c = new Case(x, y, Math.random() < 0.25, this) ;
On a également une forme générale d'expression conditionnelle qui combine un test c et deux expressions e1 et e2 :
(c) ?e1 :e2
Si le test c vaut true, alors le résultat sera celui de l'expression e1, sinon le résultat sera celui de l'expression e2. L'expression booléenne Math.random() < 0.25 est donc équivalente à (Math.random()<0.25) ?true :false ou encore à (Math.random()>=0.25) ?false :true

Enfin, la variable mine aurait pu être d'abord simplement déclarée, puis initialisée lors d'une instruction conditionnelle :
boolean mine ;
if (Math.random() < 0.25) { mine = true ; }
else { mine = false ; }

La définition d'une méthode ressemble à celle d'un constructeur, mais permet d'utiliser un nom quelconque et précise un éventuel type de résultat (ou indique void quand aucun résultat n'est produit).

Ici, la méthode nombreVoisinesPiegees prend en paramètres deux entiers représentant les coordonnées d'une case, et renvoie une valeur entière indiquant le nombre de voisines piégées de cette case.

Cette méthode sera appelée par les cases lorsque celles-ci auront besoin d'afficher le nombre de leurs voisines piégées après un clic. Elle doit donc être visible par des objets extérieurs, avec une visibilité protected ou public.

```
    public int nombreVoisinesPiegees(int x, int y) {
      // Initialisation d'une variable a 0
      int nb = 0;
      // Pour chaque case dans le carre de 3x3 autour des
          coordonnees
      // considerees...
      for (int i=x-1; i<=x+1; i++) {
        for (int j=y-1; j<=y+1; j++) {
          // si ces coordonnres sont valides et correspondent a
              une
          // case piegee, alors on incremente le compteur.
          if (coordonneesLegales(i, j) &&
            this.terrain[i][j].estPiegee())        {
            nb++;
          }
        }
      }
      // A la fin, on renvoie le nombre obtenu
      return nb;
    }
```

Remarques : l'opérateur && est paresseux, et n'évaluera pas son deuxième argument si le premier vaut déjà false. Le fragment précédent
if (coordonneesLegales(i, j) && this.terrain[i][j].estPiegee()) {
nb++ ; } est donc équivalent à :
if (coordonneesLegales(i, j)) {
if (this.terrain[i][j].estPiegee()) {
nb++ ;
}
}
et on s'assure qu'on n'essaiera jamais de lire le tableau à des coordonnées inadaptées.

Les accès à des attributs, indices de tableaux ou méthodes peuvent s'enchaîner. Pour chaque paire valide de coordonnées i, j, l'instruction précédente accède à la case ayant ces coordonnées dans le tableau terrain de l'objet courant this, puis appelle la méthode estPiegee() de cette case.

Remarque : l'attribut estPiegee est déclaré dans la classe Case avec la visibilité private et est donc accessible seulement via la méthode *getter* estPiegee().

Remarque : contrairement à ce qui était prétendu au début, l'invocation nombreVoisinesPiegees(x, y) ne renvoie pas seulement le nombre de voisines piégées de la case de coordonnées x, y, mais compte aussi cette case. Question : pourquoi peut-on se permettre cette approximation ? On reparlera de ce genre de considérations à partir de mi-février.

Des coordonnées sont valides si elles sont convenablement encadrées par les dimensions du terrain. Rappel : les indices d'un tableau t vont de 0 à t.length - 1.

Cette méthode est destinée à un usage interne, on peut en restreindre la visibilité avec une déclaration private.

```
    private boolean coordonneesLegales(int i, int j) {
      return i >= 0 && i < this.hauteur && j >= 0
                    && j < this.largeur;
    }
```

La méthode main des classes annexes comme Terrain peut être utilisée pour effectuer de petits tests unitaires. Après compilation, on peut exécuter ce test en exécutant la classe actuelle, c'est-à-dire Terrain. Dans un terminal, cela se traduit par la ligne de commande

```
java Terrain
```

Consigne générale : incluez ce genre de tests dans tous vos développements.

```
    public static void main(String[] args) {
      Terrain t = new Terrain(3, 4);
      t.terrain[0][0]  = new Case(0, 0, true, t);
      t.terrain[0][1]  = new Case(0, 1, false, t);
      t.terrain[0][2]  = new Case(0, 2, false, t);
      t.terrain[1][0]  = new Case(1, 0, false, t);
      t.terrain[1][1]  = new Case(1, 1, true, t);
      t.terrain[1][2]  = new Case(1, 2, false, t);
      // Affichage d'un compte rendu du test.
      System.out.println("Voisines␣de␣(0,␣1):␣"
        + t.nombreVoisinesPiegees(0, 1)
        + "␣(attendu␣:␣2)");
    }

  } // Fin de la classe Terrain
```

Variante possible : utiliser une assertion pour interrompre le programme en cas d'échec.

assert t.compteVoisines(0, 1) == 2 : "Erreur compte voisines";
System.out.println("Test classe Terrain OK");
Dans ce cas, il faut exécuter le programme avec l'option `-ea` (*execute assertions*), c'est-à-dire avec la commande

```
java -ea Terrain
```

1.3.3 Classe Case

La classe Case étend la classe ZoneCliquable du *package* IG. Cela va permettre de rendre les cases réactives aux clics, mais demande de définir deux méthodes clicGauche et clicDroit. Au passage, l'extension de ZoneCliquable donne aussi accès à une méthode changeTexte permettant de choisir l'éventuel texte affiché dans la case. La classe Case utilise également la classe Color de la bibliothèque standard de Java.

```
import java.awt.Color; import IG.ZoneCliquable;
class Case extends ZoneCliquable {
```

À chaque case on associe ses coordonnées, un booléen indiquant la présence ou l'absence d'une mine, un booléen (finalement non utilisé en classe) indiquant si la case a déjà été cliquée, et une référence vers le terrain pour le calcul du nombre de voisines piégées.

```
  private int x, y;
  private boolean estPiegee;
  private boolean dejaCliquee;
  private Terrain terrain;
```

La méthode publique estPiegee permet d'accéder à la valeur de l'attribut piegee. L'attribut piegee étant déclaré privé, on ne peut y accéder librement que depuis l'intérieur de la classe Case, et les méthodes de la classe Terrain par exemple sont forcées d'utiliser cette méthode d'accès.

Ce genre de méthodes, qu'on appelle aussi *getter*, est à introduire en fonction des besoins.

```
  public boolean estPiegee() {
    return this.estPiegee;
  }
```

Le constructeur prend en paramètres les coordonnées de la case, un booléen indiquant la présence ou l'absence d'une mine, et une référence vers le terrain.

```
  public Case(int x, int y, boolean mine, Terrain terrain) {
```

Première étape : appeler le constructeur de ZoneCliquable, auquel on transmet le texte à afficher initialemenet (la chaîne vide) ainsi que les dimensions de la case (40x40 pixels).

```
    super("", 40, 40);
```

Deuxième étape : initialisation des attributs avec les valeurs fournies. Exception : l'attribut dejaCliquee est initialisé à la valeur false dans tous les cas.

```
    this.x = x;
    this.y = y;
    this.estPiegee = mine;
    this.dejaCliquee = false;
    this.terrain = terrain;
  } // Fin du constructeur
```

Alternativement, la valeur de l'attribut dejaCliquee aurait pu être fixée dès la déclaration de cet attribut en remplaçant la ligne
private boolean dejaCliquee;
par la ligne
private boolean dejaCliquee = false;
Dans ce cas il n'aurait plus été nécessaire d'initialiser cette valeur dans le constructeur : on aurait omis la ligne
this.dejaCliquee = false;

La méthode clicGauche colore la case en rouge si la case était piégée, et affiche sinon le nombre de voisines piégées.

```
  public void clicGauche() {
    if (this.estPiegee) {
/* La methode [setBackground] est recuperee de la bibliotheque
   standard de Java via [ZoneCliquable], et change la couleur
      de
   fond d'un element graphique.
 */
        setBackground(Color.RED);
        // Plus fin de partie a gerer
      } else {
        int nb = this.nombreVoisinesPiegees();
        this.changeTexte(Integer.toString(nb));
      }
  }
```

Remarque : la méthode changeTexte attend en paramètre une chaîne de caractères. Or, le nombre de voisines piégées est un nombre entier. On applique donc une fonction de conversion proposée par la bibliothèque Integer de Java.

Seul le terrain a accès à toutes les cases et est donc apte à calculer le nombre des voisines piégées. La méthode nombreVoisinesPiegees de la case fait donc appel à la méthode correspondante du terrain, en lui fournissant en paramètres ses coordonnées.

```
  public int nombreVoisinesPiegees() {
    return this.terrain.nombreVoisinesPiegees(x, y);
  }
```

La méthode clicDroit colore la case en bleu.

```
  public void clicDroit() {
    setBackground(Color.BLUE);
  }

} // Fin de la classe Case
```

2 Classe Génériques,Interface, Introduction au Génie Logiciel

2.1 Classes génériques<T>

Pour programmer, on utilise souvent des "container" https://en.wikipedia.org/wiki/Container_(abstract_data_typ
Ce sont des méta-types permettant de stoquer une collection de données d'un certain type. Il est pratique de pouvoir laisser ce type en paramètre, lorque on programme un container, pour pouvoir réutiliser le container pour différent

type. L'exemple emblématique est le container "tableau" ou array en anglais. On veut pouvoir définir des tableux de n'importe quoi : des tableaux d'entier des tableaux de cases d'échiquier, des tableaux de tableaux.... Ce sont là, à chaque fois, de nouveaux types utilisables ; Donc on peut voir "tableau" comme un constructeur de type. On peut aussi voir tableau comme un type générique paramétrés par le type du truc qui sera rangé dedans. Il existe bien d'autre containers dans la bibliothéque java, comme `ArrayList<T>, Set<T>, HashMap<K,T>`, ceux ci permettent de déclarer des tableaux extensibles, des ensembles, des tables de hashage, ou tableau associatifs.

On peut aussi soi-même déclarer nos propres container génériques, le mieux pour comprendre est un exemple simple. Je propose le type pile (stack en anglais). On peut empiler sur le haut de la pile, dépiler un élement, et tester si la pile est vide.

```
public class MyStack<T> {
    private T[] elements;
    private int capacity;
    private int size=0;
    MyStack(int c){
        capacity=c;
        elements=(T [])new String[c];
  //on ne peut pas écrire directement
  //elements=new T[c];on alloue l'espace
  //pour un tableau de pointeurs vers des Strings,
  // et aprés on cast vers un tableau de T
    }
    public void push (T elt){
        if(size<capacity) {
            elements[size] = elt;
            size++;
        }
        else
            System.out.println("stack␣overflow");
        //on pourrait aussi réallouer le tableau
    }
    public T pop(){
        if(size>0){
            T res=elements[size-1];
            size--; return res;}
        else{ System.out.println("stack␣underflow");
            return null; }
    }
    public boolean isEmpty(){
        return (size==0);
    }
    public String toString() {
        String res="";
        for (int i=0;i<size;i++)
            res+=elements[i]+":";
        return res;
    }
    public static void main(String args[]) {
        MyStack<String> stack = new MyStack(10);
        stack.push("tata");
        stack.push("tutu");
        stack.push("toto");
        System.out.println(stack);
        stack.pop();
        System.out.println(stack);
    }
}
```

Dans ce programme, on montre un exemple d'utilisation avec une pile de chaine de caractères. La variable de type s'appelle T, c'est la convention courante. Ce nom T est utilisée pour déclarer des variables de type T, ou pour spécifier le type retourné par la méthode `pop()`.

2.2 Interfaces

C'est quoi une interface. Au moment de définir une classe `A`, on peut spécifier que cette classe `A` *implémente une interface* `I`.

- `class A implements I { ... }`
- `class A extends B implements I { ... }`
- `class A extends B implements I, J, K { ... }`

Une interface, c'est un ensemble de variables et de *signatures* de fonctions (signature = nom + type de retour + types des arguments), regroupées dans un paquet qui est défini similairement à une classe abstraite dont toutes les méthodes seraient abstraites.

```
interface I {
  public    int  a();
  protected void b(String s);
}
```

Une classe `A` qui implémente une interface `I` doit définir des méthodes correspondant à chaque signature présente dans `I`. On peut voir l'interface comme une classe abstraite dont *toutes* les méthodes seraient abstraites, et dont on pourrait "hériter" à condition de redéfinir ces méthodes abstraites.

Différence avec l'héritage : on utilise le mot `implements`, et alors qu'une classe `A` qu'on ne peut hériter (`extends`) que d'*une seule* classe `B` (abstraite ou non), elle peut implémenter (`implements`) plusieurs interfaces `I`, `J`, `K`...

Utilité des interfaces En général on programme toujours en réutilisant des bibliothéque de classes déja faite. soit en regardant sur la bibliothéque java standard `docs.orcale.com` . On se fiche de savoir comment les méthodes sont implémentées, on regarde juste l'interface qui est déjà assez longue à lire. Une interface définit un contrat entre vous, le client/utilisateur d'une telle bibliothéque , et le programmeur qui à fournit cette bibliothéque, i.e qui a programmé toutes les primitives que vous souhaitez utiliser. De manière plus concréte, faire que les objets que vous manipulez implémentent certaines interfaces vous permettra d'utiliser quelques fonctions intéressantes de la bibliothèque standard de java. Par exemple, les tableaux "tout courts", (`Array`)extensibles (`ArrayList`) proposent une méthode de tri (`sort`), qui peut-être appliquée lorsque les éléments implémentent l'interface `Comparable`. Cette interface prescrit une méthode `public int compareTo(Object other)`, qui compare l'objet courant (`this`) à l'objet argument (`other`), et renvoie :

- -1 si `this` est plus petit que `other`.
- 1 si `this` est plus grand que `other`.
- 0 si `this` et `other` sont égaux.

```
interface Comparable {
  public int compareTo(Object other);
}
```

Donc si vous voulez trier un array ou un arrayList de “bidule”, vous n'avez que à definir la méthode compareTo dans la classe de “bidule”, et vous récupérer la possibilité de trier.

Réaliser une interface : l'exemple de comparable. Certaines classes de base implémentent déjà `Comparable`, par exemple la classe `String` (qui suit l'ordre du dictionnaire, dit *lexicographique*) ou la classe `Integer` (qui suit l'ordre usuel sur les entiers). Donc pour ces types la, vous n'avez même pas besoin de programmer compareTo. C'est en revanche à l'utilisateur de fournir la fonction de comparaison de toute nouvelle classe, comme par exemple SIPair.

```
class SIPair implements Comparable {
  private String s;
  private int    n;
  public String getS() { return this.s; }
  public int    getI() { return this.i; }
  public SIPair(String s, int i)
     { this.s = s; this.i = i; }

  public int compareTo(Object other) {
      return this.s.compareTo(
         ((SIPair) other).getS() );
  }
}
```

Probleme de comparable vis a vis de la vérification de type. L'interface que nous venons d'utiliser a un défaut : la méthode `compareTo(Object other)` prend comme argument un objet de la classe `Object`, qui doit ensuite être transtypé dans le type qui nous intéresse (`SIPair`). Cette classe particulière `Object` est la mère de toutes les classes que vous pouvez définir en Java : toutes les autres classes que vous pourrez écrire ou utiliser seront des héritères de `Object`. Autrement dit, pour le vérificateur de types, cette méthode accepte littéralement n'importe quel argument, et il n'y a de plus aucune restriction sur ce vers quoi ledit argument peut être transtypé. Un appel comme :

```
SIPair pair = new SIPair("un", 2);
pair.compareTo(0);
```

sera accepté par le compilateur, car l'entier `0` appartient bien par héritage à la classe `Object`, et car il est parfois légitime de transtyper un `Object` vers `SIPair`. En revanche, comparer une paire avec un entier n'a pas de sens, et la tentative de transtypage de `0` en `SIPair` va déclencher une erreur à l'exécution.

Ce mécanisme de transtypage passant par la Classe Mère `Object` permet d'écrire des fonctions qui s'appliquent à une grande variété d'arguments. Cela se fait en revanche au prix de possibles erreurs de transtypage qui ne sont détectées que à l'exécution. Le reste de ce chapitre présente un mécanisme de généricité plus précis, qui permet de vérifier dès la compilation le bon usage des fonctions génériques, et donc d'éviter les erreurs de transtypage à l'exécution.

2.3 Interface génériques

Quand on implémente la classe `Comparable`, il est possible de considérer la version générique qui paramétrise le type `T` des objets avec lesquels on permet la comparaison. L'interface aussi peut être générique : il suffit d'ajouter la mention `<T>`. La définition complète de l'interface générique `Comparable` est :

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

Dans le cas de notre classe `SIPair`, les objets doivent pouvoir être comparés à d'autres objets de la même classe, et donc implémenter `Comparable<SIPair>`.

```
class SIPair implements Comparable<SIPair> {
  ...
  public int compareTo(SIPair other) {
      return this.s.compareTo( other.getS() );
  }
}
```

Plusieurs variables de types Une classe ou une interface générique peut être paramétrée par un nombre quelconque de *variables de type* (ci-dessous : `T1` et `T2`), auxquelles on peut ensuite faire référence dans le corps de la définition de la classe.

```
class Pair<T1, T2> {
  private T1 first;
  private T2 second;
  public T1 getFirst()    { return this.first;  }
  public T2 getSecond()   { return this.second; }
  public Pair(T1 f, T2 s) {
    this.first = f; this.second = s; }
}
```

Contrainte sur les variables de type Il est possible d'imposer des contraintes aux variables de types pour s'assurer que les classes correspondantes possèdent certaines méthodes qui nous intéressent. On peut modifier notre exemple pour s'assurer que la classe instanciant `T1` implémente l'interface `Comparable`, ce qui nous permettra de comparer deux paires en fonction de leur premier élément.

```
class Pair<T1 extends Comparable<T1>, T2> {
  ...
  public int compareTo(Pair<T1, T2> other) {
    return this.first.compareTo( other.getFirst() );
  }
}
```

Faute de mention `T1 extends Comparable<T1>`, le code précédent eût été rejeté par le compilateur : la variable de type `T1` pouvant a priori représenter n'importe quelle classe, il n'eût pas été garanti que l'attribut `this.first` possédât une méthode `compareTo`.

Avec la restriction sur `T1` en revanche, ce code est légitime, et il est même possible d'étendre la première ligne

pour préciser que notre classe `Pair` implémente l'interface `Comparable<T1, T2>` :

```
class Pair<T1 extends Comparable<T1>, T2>
   implements Comparable<Pair<T1, T2>>
```

Cette restriction posée, il est possible de construire des objets de la classe `Pair<String, int>` (car `String` implémente `Comparable<String>`). En revanche, le compilateur rejettera l'utilisation d'un type `Pair<String[], int>`, car les tableaux ne sont pas comparables.

2.4 Introduction au Génie logiciel

L'introduction utilise des slides qui sont dans votre receuil de slides.

Le génie logiciel traite des concepts et des méthodes pour la construction d'un logiciel. Les objectifs visés sont en particulier la réponse à des besoins exprimés par un client ou des utilisateurs et l'obtention d'un logiciel qui fonctionne et qu'il sera possible de faire évoluer.

2.4.1 Cycle de vie du logiciel

On peut distinguer deux étapes essentielles dans la vie d'un logiciel :

1. La genèse et le développement : de la définition du projet jusqu'à la livraison de la première version de production. Le travail d'ingénierie logicielle a lieu essentiellement à cette étape, dont la durée peut être de l'ordre de quelques mois à un petit nombre d'années.
2. La vie : une fois le logiciel entré en usage. Il reste à cette étape un travail de maintenance, qui peut s'étendre sur un grand nombre d'années. Ce travail peut prendre trois formes :
   — la *correction* de problèmes,
   — l'*adaptation* à l'évolution des infrastructures matérielles et logicielles, et
   — l'*évolution* du logiciel lui-même, par exemple par ajout de nouvelles fonctionnalités.

La période de maintenance est bien plus longue que celle de développement, et c'est finalement là que se concentre souvent la plus grosse partie du coût (temps de travail, finances) d'un logiciel. Par conséquent, un des objectifs de l'ingénierie logicielle dans la phase initiale est de bâtir un logiciel facilitant ce travail ultérieur de maintenance.

2.4.2 Phases de développement

On peut organiser un développement logiciel en quatre temps :

1. Analyse : définir les besoins du logiciel. Parmi eux, on distingue
   — les besoins fonctionnels : les fonctionnalités qui doivent être fournies par le logiciel,
   — les besoins non fonctionnels : les contraintes additionnelles, liées par exemple à l'ergonomie ou aux performances.
   Cette phase détermine *ce qui* doit être fait.
2. Conception : définir *comment* réaliser le logiciel répondant aux besoins. Cela concerne en particulier, du plus large au plus pointu : l'architecture du système, ses principaux composants, les structures de données, les algorithmes, etc.
3. Code.
4. Validation : s'assurer que le logiciel produit correspond au plan et répond aux besoins. Cela couvre en particulier, du plus pointu au plus large : test unitaire (des fonctions isolées), test d'intégration (test de plusieurs composants assemblés), déploiement local , version béta, déploiement global.

Dans un développement complètement planifié on peut envisager de faire un seul passage par ces quatre phases. Il est souvent intéressant de procéder en plusieurs itérations, en faisant plusieurs cycles à travers ces quatre phases, pour faire grossir petit à petit le système à construire. Cela permet notamment de fournir plus tôt des premières versions à évaluer au client, et d'avoir des retours rapides sur les parties jugées les plus risquées (qu'il convient donc de traiter dans les premières itérations).

3 Phase analyse : Diagramme de cas d'utilisation, scenario

Ce cours sur UML utilise deux sets de slides présentés sur
`https://www.youtube.com/watch?v=GC5BdRve38A&list=PLSVDd2z0rl6PlOscYvneTFvU8qqPFzFkl`
`https://www.youtube.com/watch?v=1G0omjzh1OQ&list=PLSVDd2z0rl6PlOscYvneTFvU8qqPFzFkl`

3.1 Cas d'utilisation : diagramme, descrition détaillé

Les diagrammes de cas d'utilisation décrivent les principales fonctionnalités d'un système, et précisent les personnes ou entités extérieures impliquées dans chaque fonctionnalité. Ils sont généralement complétés par des descriptifs détaillés de ce qui est attendu de chaque fonctionnalité.

3.1.1 Éléments principaux

Un diagramme de cas d'utilisation comporte des *acteurs*, associés à des *cas d'utilisation*. Un cas d'utilisation représente une fonctionnalité du système, c'est-à-dire une tâche que celui-ci doit pouvoir accomplir. Chaque cas d'utilisation est associé aux acteurs impliqués dans la réalisation de la fonctionnalité qu'il représente. Dans le diagramme de cas, les cas d'utilisation sont placés dans des bulles, à l'intérieur d'un grand cadre représentant le système.

3.1.2 Les acteurs

Les acteurs ont déja étaient introduit dans les scénario. Dans le diagramme de cas, on les représente en général par un personnage-bâtons, notamment pour les acteurs humains.

Dans le cas d'un acteur non humain, comme par exemple un autre système informatique interagissant avec le système étudié, on peut aussi dessiner une simple boîte portant la mention `<<acteur>>`. Les acteurs sont représentés à l'extérieur du système dans le diagramme de cas. Deux acteurs peuvent être liés par une flèche de généralisation/spécialisation, dont la signification est proche de la notion de généralisation entre classes : l'acteur spécialisé "hérite" des associations aux cas d'utilisation auxquels l'acteur général est associé, et peut ajouter d'autres associations à de nouveaux cas.

3.1.3 Les cas d'utilisation

Un cas d'utilisation est une fonctionnalité fournie par le système, qui est utilisée ou déclenchée par l'un des acteurs, et qui correspond à une tâche que celui-ci veut ou doit accomplir. Ils correspondent grosso-modo aux messages echangés dans un scenario. Chaque cas d'utilisation est nécessairement associé à un acteur principal, qui est l'acteur qui utilise, déclenche, ou bénéficie de la fonctionnalité. Un cas d'utilisation peut de plus être associé à un ou plusieurs acteurs secondaires, qui sont impliqués dans le déroulement de l'action sans en être à l'origine ou sans en être les principaux bénéficiaires.

Les cas d'utilisation peuvent être mis en relation de trois manières :

— Par une relation de généralisation/spécialisation, qui a à nouveau une signification similaire à la généralisation de classes : le cas d'utilisation spécialisé (à l'origine de la flèche) est un cas particulier du cas d'utilisation général (à la pointe de la flèche). Le cas spécialisé "hérite" des associations (à des acteurs ou à d'autres cas) du cas généralisé et peut en ajouter de nouvelles.

— Une une relation d'inclusion (flèche pointillée étiquetée par `includes`), d'un cas X vers un cas Y, signifie que X comprend systématiquement Y. Le cas Y est vu un peu comme un appel a une procédure, comme par exemple payer, s'authentifier.

— Une relation d'extension (flèche pointillée étiquetée par `extends`), d'un cas X vers Y, signifie que le cas X peut être appelé au cours de l'exécution du cas Y, mais c'est optionnel.

3.1.4 Erreur fréquente

Ces relations ne doivent pas être utilisées pour préciser qu'un cas d'utilisation contient un certain nombre d'étapes si ces étapes ne sont pas de vraies fonctionnalités du système. Un diagramme de cas ne comporte pas plus que 7 ou 8 cas pour conserver la lisibilité. Critère : une "vraie" fonctionnalité est une chose qu'un acteur peut vouloir réaliser pour elle-même, indépendamment des éventuelles autres fonctionnalités la contenant.

3.1.5 Description détaillée des cas d'utilisation

Pour compléter la description des cas d'utilisation, on peut utiliser la table 1 : Les procédures normales ou alternatives peuvent être décrite ou bien par plusieurs petit diagramme de séquence, ou bien textuellement, en décrivant les évenements, et en les numérotant pour y faire référence.

3.2 Diagramme de séquences

3.2.1 Des outils pour la phase d'analyse

Le langage UML (*Unified Modelling Language*) permet de décrire les fonctionnalités et l'architecture d'un système en utilisant des diagrammes. On présente ici deux premiers types de diagrammes, permettant de représenter les fonctionnalités d'un système. Ces diagrammes sont un support pour la phase d'analyse.

3.2.2 Acteur

Un acteur représente une personne ou une autre entité extérieure qui interagit avec le système. On a deux critères principaux pour définir les acteurs :

1. ils ne font pas partie du système lui-même,
2. ils interagissent directement avec le système.

Les acteurs peuvent être humain ou non humain. Les acteurs, notamment humains, ne représentent pas à strictement parler des personnes physiques, mais des *rôles* qui peuvent être tenus par des personnes physiques. Ainsi, une même personne interagissant avec un système à la fois en tant qu'administrateur et en tant qu'utilisateur ordinaire, sera vue successivement comme acteur "administrateur" ou comme acteur "utilisateur".

3.2.3 Diagramme de séquence

Les diagrammes de séquences décrivent des enchaînements d'événements caractérisant l'utilisation du système, et en particulier les communications entre le système et les acteurs extérieurs. On peut les utiliser pour décrire le déroulement d'un cas d'utilisation.

Les diagrammes de séquence possèdent un axe des temps vertical implicite : l'ordre des événements est donné par une lecture de haut en bas. Cet axe ne donne que l'ordre : il n'a pas d'échelle et ne dit rien sur les durées.

Le système et les différents acteurs sont représentés côte à côte, par des lignes verticales appellées "ligne de vie" surmontées d'une boîte indiquant la nature et éventuellement l'identité de l'acteur ou du système.

Un message entre le système et un acteur est représenté par une flèche horizontale allant de la ligne de l'émetteur du message vers la ligne du récepteur. Cette flèche est annotée par la nature du message et des paramètres.

On peut inclure dans le diagramme des boîtes décrivant des séquences alternatives, ou appelant un autre cas d'utilisation.

3.3 Scenario

Les diagramme de séquence sont utilisés pour décrire des utilisation plus large du système dans lesquels on voit différents cas d'utilisation se succéder. Dans ce cas on instancie les messages et les acteurs avec des valeurs concrètes, et le diagramme de séquence s'appelle un "scénario". Dans la discussion avec le client, pour élaborer l'analyse, on commence

Nom	*nom du cas d'utilisation*
Description courte	*résumé de la fonctionnalité*
Précondition	*condition préalable nécessaire au bon déroulement*
Postcondition	*résultat, effet*
Situations d'erreur	*ce qui peut poser problème*
En cas d'erreur	*résultat, effet en cas de problème*
Acteurs	*acteur principal et acteurs secondaires*
Déclencheur	*ce qui fait que le cas est déclenché*
Procédure normale	*déroulement dans le cas normal (indiquer les différentes étapes)*
Procédures alternatives	*déroulements alternatifs correspondant aux cas d'erreur*

Table 1 – Description cas d'utilisation

par décrire des scénario pour se faire idée de ce que le système doit faire.

3.3.1 Erreur féquentes

Les messages circulent exclusivement entre le système et les acteurs, il ne peut y avoir de messages circulant entre deux acteurs, car cela se passerai à l'extérieur du système, et on ne modèlise que le système.

3.4 Champs d'instance, champs de classe

La situation usuelle pour les attributs et les méthodes est la suivante :

- La classe contient des déclarations (c'est-à-dire des descriptions, ou spécifications) d'attributs et de méthodes.
- Chaque objet de la classe possède sa propre version de chaque attribut déclaré, et les méthodes d'un objet agissent sur les attributs de cet objet.

Dans ce cadre, de plus, tout appel de méthode se fait par l'intermédiaire d'un objet, et en particulier aucune méthode ne peut être appelée sans qu'un objet de la classe correspondante ne soit créé au préalable. Cette situation est celle des champs d'instance (champ désigne un attribut ou une méthode, tandis qu'instance est synonyme d'objet ; on dit aussi qu'un objet instancie une classe).

Attributs de classe Cependant, il existe aussi des champs de classe, qui ne sont pas liés à un objet mais directement à une classe. Autrement dit, tous les objets de la classe partagent un attribut unique, au lieu d'en avoir chacun une version séparée comme avant. De tels attributs sont introduits par le mot-clé `static`.

```
class A {
public
int a;
public static int b=1;
public void print() {
   System.out.println(a + " " + b); }
}
A obj1 = new A();
A obj2 = new A();
obj1.a = 1;
obj2.a = 2;
obj1.b = 3;
```

appel	affichage	Erreur compilation
obj1.print()	1 3	
obj2.print()	2 3	
obj2.b = 4 ; obj1.print()	1 4	
System.out.println(A.b)	3	
System.out.println(A.a)		a n'est pas statique

Dans cet exemple, l'attribut (d'instance) a a une valeur différente dans l'objet obj1 et dans l'objet obj2. En revanche, les deux objets partagent une même valeur pour l'attribut (de classe) b, et dès que l'un des objets modifie la valeur de b, le changement est visible pour les deux objets. Trois conséquences en chaîne :

- On peut faire référence à l'attribut de classe b sans passer par un objet de notre classe A. Un appel direct A.b remplace obj1.b ou obj2.b.
- En particulier, on peut consulter l'attribut de classe via A.b même si aucun objet n'a été créé !
- Par conséquent, il est judicieux de donner une valeur par défaut à l'attribut de classe lors de sa déclaration, plutôt que d'attendre qu'il soit initialisé lors de la création d'un objet.

Une application : compter le nombre d'objets créés pour une classe.

```
class A {
private static int compteur = 0;
public A() { compteur++; }
}
```

Ici, un champ de classe compteur est associé à la classe A, et ce compteur est incrémenté à chaque appel du constructeur A(). Bilan : cet attribut est partagé entre tous les objets de la classe A et indique le nombre des objets qui ont été créés.

Méthodes de classe Le mot-clé static peut également être appliqué à une méthode. Dans ce cas, cette méthode ne peut utiliser des champs de la classe à laquelle elle appartient que si ceux-ci sont aussi qualifiés de static.

```
class A {
private static int compteur = 0;
private
int identite;
```

```
public static void printCompteur() {
System.out.println(compteur);
}
public void printId() {
System.out.println(identite + " sur " + compteur);
}
public A() {
compteur++;
this.identite = compteur;
}
}
A obj1 = new A();
A obj2 = new A();
A obj3 = new A();
```

appel	affichage	Erreur compilation
obj1.printCompteur()	3	
A.printCompteur()	3	
obj2.printId()	2 sur 3	
A.printId()		printId() pas statique

Comme pour les attributs de classe, les méthodes de classe n'ont pas besoin d'être appelées par un objet, et peuvent intervenir avant même qu'un objet ne soit créé. Et c'est ce qui se passe avec la méthode main ! public static void main(String[] args) ...

3.5 Final : Contrôler l'héritage et la rédéfinition

Le mot-clé final peut être utilisé lors de la déclaration d'un attribut, d'une méthode, ou d'une classe. Les trois cas ont des significations différentes, mais une idée commune : empêcher que quelque chose soit modifié.

Attributs immuables L'utilisation de final pour qualifier un attribut fait que la valeur de cet attribut, dès lors qu'il a été initialisé, ne peut plus être modifiée. On parle d'attribut immuable, par opposition aux attributs usuels mutables dont la valeur peut être modifiée. Le qualificatif final s'applique aussi bien aux attributs d'instance qu'aux attributs de classe.

```
class A {
private static
int compteur = 0;
private static final int maximum = 10;
private
final int identite;
public A() {
if (compteur < maximum) { compteur++; }
else
{ compteur=0; }
this.identite = compteur;
}
}
```

Dans cet exemple, les deux attributs compteur et maximum sont associés à la classe (qualificatif static), et partagés par tous les objets. L'attribut compteur est modifié à chaque création d'objet, tandis que l'attribut maximum est qualifié de final et ne peut jamais être modifié : sa valeur de 10 est définitive et est partagée par tous les objets. On a aussi un attribut d'instance identite, qui pourra être différent d'un objet à l'autre. Cet attribut est également qualifié de final, ce qui signifie que l'identite d'un objet ne sera jamais modifiée une fois l'objet créé. Quelques variantes qui n'auraient pas marché :

- Déclarer compteur comme final n'est pas possible, car sa valeur est modifiée par une méthode.
- Ne pas initialiser this.identite ou maximum génère une erreur : un attribut final doit être initialisé.
- Initialiser maximum dans le constructeur A() n'est pas possible : il faut que cet attribut de classe soit initialisé même si aucun objet n'est créé.

Méthodes finales L'utilisation de final pour qualifier une méthode d'une classe A empêche que cette méthode soit redéfinie dans une sous-classe B de A.

```
class A {
public final a() { System.out.println("Classe A"); }
}
class B extends A {
public a()
{ System.out.println("Classe B"); } // ERREUR
public a(int n) { System.out.println("Classe B" + n); }
}
```

Cet exemple provoque une erreur de compilation indiquant qu'on ne peut pas redéfinir la méthode a(). Il est toujours possible en revanche de surcharger une méthode finale, puisque la surcharge n'est pas une redéfinition. Ainsi la déclaration de a(int n) est valable.

Classes finales L'utilisation de final pour qualifier une classe A empêche la définition de sous-classes de A. Cela implique en particulier que toutes les méthodes de A sont aussi "finales" (on ne peut de toute façon pas redéfinir ces méthodes sans faire une sous-classe). En revanche, qualifier une classe A par final n'implique pas que tous les attributs de A soient immuables (final).

```
final class A {
public final int n=0;
public
int m=0;
public void an() { n++; } // ERREUR
public void am() { m++; } // OK !
}
class B extends A { } // ERREUR
```

Dans cet exemple, on a deux erreurs à la compilation : d'une part il est impossible de faire que B hérite de A, et d'autre part la méthode an() n'a pas le droit de modifier l'attribut immuable n. En revanche, la définition de la méthode am() est valide : bien que la classe A soit finale, l'attribut m n'est pas immuable.

Forcer la redéfinition Lorsque, dans une classe B héritant d'une classe A, on redéfinit une méthode a(), la définition peut être précédée de l'indication @Override. Ceci indique au compilateur que l'on a l'intention de redéfinir une méthode d'une super-classe. Dans ce cas, le compilateur vérifie que cette déclaration redéfinit une méthode de la classe mère, et génère une erreur si ce n'est pas le cas.

```
class A {
public void a() {
   System.out.println("Classe A"); }
}
class B extends A {
@Override
public void a() {
    System.out.println("Classe B"); }
@Override // ERREUR
public void a(int n) {
    System.out.println("Classe B"+n); }
}
```

Dans cet exemple, la classe B redéfinit la méthode a(), et la première indication @Override est donc satisfaite. En revanche, le deuxième @Override déclenche une erreur à la compila- tion, car la méthode a(int n) ne redéfinit rien (elle ne fait que surcharger une méthode). Remarque : les indications @Override ne sont là que pour demander au compilateur de faire des vérifications, et ne changent pas le comportement du programme. Si un programme avec @Override est accepté par le compilateur, alors on peut retirer toutes les indications @Override, et ces deux programmes auront exactement le même comportement

3.6 Visibilité des attributs

Un champ d'une classe Java, qu'il s'agisse d'un attribut, d'une méthode ou d'un constructeur, est systématiquement associé à un niveau de visibilité définissant quelles classes peuvent accéder à ce champ. Il existe quatre niveaux, donnés ici du plus restrictif au plus libre

1. niveau "restreint" ou "privé" (mot-clé [private]) : le champ n'est accessible que depuis la classe elle-même
2. niveau "restreint au paquet" (niveau par défaut, sans mot-clé associé) : le champ est accessible depuis la classe elle-même et depuis toutes les classes définies dans le même paquet (approximativement : définies dans le même dossier)
3. niveau "protégé" (mot-clé [protected]) : le champ est accessible depuis la classe elle-même, depuis ses classes filles, et encore depuis les autres classes définies dans le même paquet
4. niveau "ouvert" (mot-clé [public]) : le champ est accessible depuis n'importe quelle classe

4 Conception : Diagramme de classe, associations

La phase de conception intervient une fois complétée l'analyse du système à construire, et définit les structures de données et l'organisation du code. La phase d'analyse s'intéressait au "quoi", la phase de conception se concentre sur le "comment".

4.1 Diagramme de classes

Ce cours sutilise deux sets de slides sur comment représenter les diagrammes de classes, avec des associations. Ils sont commentés sur
https://www.youtube.com/watch?v=8VMMu-vcF60&list=PLSVDd2z0rl6PlOscYvneTFvU8qqPFzFkl&index=3
https://www.youtube.com/watch?v=nRqTXoiNUHk&list=PLSVDd2z0rl6PlOscYvneTFvU8qqPFzFkl&index=4

4.2 Réaliser des associations

Je souhaite avant tout qu'a l'issue du cours 4, la notion de diagramme de classe et d'association soit bien comprise. Il est donc possible que cette partie soit faite plus tard, voire pas du tout. A présent on considére le problème d'implémenter des associations avec Java. Les associations peuvent être réalisées de multiples façons. On donne ici quelques exemples en fonction de l'arité de l'association et de la navigabilité souhaitée. Ces exemples sont des bonnes pratiques car ils font en sorte que les arités soient respectées en permanence, mais ils ne sont pas les seules manières de faire.

4.2.1 Associations unidirectionnelles

On parle d'une association unidirectionnelle lorsque la navigation se fait toujours dans la même direction. Dans les exemples suivants, un objet de la classe [A] pourra accéder à un (ou des) objet de la classe [B]. Exemple1 : Association unidirectionnelle, arité 1-1 : on crtée un objet [B] privé directement à l'intérieur de la classe [A] : personne d'autre n'y a accès.

```
class A { private B b;
    public A() { this.b = new B();}
}
class B {...}
```

Association unidirectionnelle, arité *-1 :Ici, l'objet B est pris en paramètre : il a été défini à l'extérieur et peut être lié à d'autres objets A.

```
class A { private B b;
    public A(B b) { this.b = b; }}
class B {...}
```

Association unidirectionnelle, arité *-* : Un objet de la classe [A] est associé à une collection d'objets de la classe [B]. Cette collection peut être 1- ordonnée ou non, avec répétitions ou non, 2-réalisée par différentes structures de données, liste, ensemble, tableaux...

```
class A { private Set bs;
    public A() {
    /* Crée un ensemble vide */
        this.bs = new HashSet();}
   /* Ajout un élément [B] défini àl'extérieur.    */
    public void addB(B b) { bs.add(b);      }
}

class B {...}
```

4.2.2 Associations bidirectionnelles.

On parle d'une association bidirectionnelle lorsque la navigation peut se faire dans les deux sens. Dans les exemples suivants, un objet de la classe [A] pourra accéder à un (ou des) objet de la classe [B], qui pourra lui-même accéder à l'objet [A] en retour.

Association bidirectionnelle, arité 1-1 :Le constructeur de [B] prend en paramètre l'objet [A] auquel il lui faut s'associer. Lorsqu'il crée un objet [B], [A] se passe lui même en paramètre, de manière à ce qu'il soit associé à cet objet [B].

```
class A {private B b; /* un objet [A] pointe vers un objet [B]
    */
    public A() {this.b = new B(this); } }

class B {private A a; /* un objet [B] pointe vers un objet [A]
    */
   public B(A a) { this.a = a; } }
```

La version que nous venons de voir est asymétrique : c'est en créant l'objet [A] que l'on crée la paire d'objets associés. On peut rendre la situation plus symétrique en ajoutant un constructeur [public A(B b)] à [A] et un constructeur [public B()] à [B].

Association bidirectionnelle, arité 1-* : L'ajout d'un élément [b] à la liste des [B] pointée par un objet [A] implique aussi de mettre à jour cet objet [b], pour que il pointe vers cet objet [A].

```
class A {
    private List bs;
    public A() {this.bs = new ArrayList(); } /* au début aucun
        b */
    public void addB(B b) {
        this.bs.add(b);/* Pour l'instant l'état n'est pas cohé
            rent */
        b.setA(this);/* Maintenant c'est bon. } */
    /* Cette méthode sera appelée par [B.setA]. */
    protected void removeB(B b) {this.bs.remove(b); }
}

class B {
    private A a;
    protected void setA(A a) {
    /* Quand on ajoute l'objet àla liste d'un [A], il faut
        supprimer l'ancienne association àun autre [A].*/
        this.a.removeB(this);
        this.a = a;/*Enfin on peut faire la mise àjour */
    }
}
```

Exercice : faire une association bidirectionnelle *-*

5 Classes abstraites et polymorphisme

On rappelle d'abord ce qu'est une classe abstraite en contrastant avec les interface ; ensuite on programme en live pendant le cours, des classes pour des figures géometriques, illustrant la connection entre classe abstraite et polymorphisme. On programmera egalement un exemple illustrant le principe de mémoisation au programme du TP. Ce cours utilise en partie un jeu de slides sur les opérations, dont les commentaires ne sont pas dispo.

5.1 Les classes abstraites.

Une classe abstraite est une classe dont toutes les méthodes n'ont pas été implémentées. Elle n'est donc pas instanciable, mais sert avant tout à factoriser du code. Une classe qui hérite d'une classe abstraite doit obligatoirement implémenter les méthodes manquantes (qui ont été elles-mêmes déclarées « abstraites » dans la classe parente). En revanche, elle n'est pas obligée de réimplémenter les méthodes déjà implémentées dans la classe parente (d'où une maintenance du code plus facile).

Interface comme classe trés abstraite Une interface est un peu comme une classe abstraite dans laquelle aucune méthode ne serait implémentée : les méthodes y sont seulement déclarées. Cela permet de définir un ensemble de services visibles depuis l'extérieur (l'API : Application Programming Interface), sans se préoccuper de la façon dont ces services seront réellement implémentés. Une classe qui implémente une interface doit obligatoirement implémenter chacune des méthodes déclarées dans l'interface, à moins qu'elle ne soit elle-même déclarée... abstraite !

Différence classe abstraite et interface Classes abtraites et interfaces ont chacune une fonction bien distincte : les classes abstraites servent à factoriser du code, tandis que les interfaces servent à définir des contrats de service. Et pourquoi ne pas utiliser des classes abstraites (dans lesquelles aucune méthode ne serait implémentée) en lieu et place des interfaces ? La réponse est simple : dans la plupart des langages actuels (c'est notamment le cas de Java), il n'est possible pour une classe d'hériter que d'une seule classe parente (abstraite ou non), mais par contre, rien n'empêche d'implémenter plusieurs interfaces.

5.2 Exemple simple de classe abstraite

Le programme suivant réalisé en cours, est concu pour illustrer les points suivant en relation avec les classes abstraites :

1. utilisation du constructeur de la classe mére : super()
2. l'interêt de la visibilité "protected"
3. polymorphisme, un objet peut avoir plusieurs types
4. methode abstraite
5. une classe abstraite ne peut s'instancier

6. une classe fille doit implementer les méthode abstraites
7. usage de la redefinition pour implementer facilement une méthode qui a un petit nombre de cas particuliers.

```
import java.awt.*;
import java.util.ArrayList;

// Geom doit etre déclarée abstraite
// car elle posséde une méthode abstraite
public abstract class Geom {

    public static void main(String []args){
        Point p=new Point(0,0);
        Circle c=new Circle (p,10);
        Rectangle r=new Rectangle (p,10, 20);
        Triangle t=new Triangle (p,10, 20, 21);
        ArrayList<Geom> figures= new ArrayList<>();
        figures.add(c);figures.add(r);figures.add(t);
        for(Geom g:figures)
            // Polymorphisme = une variable a plusieurs types:
            // g est de type Geom mais aussi Triangle, Circle,
            // Rectangle. La méthode toString se
            // comporte différement dans chaque cas
            System.out.println(g);
    }


    protected Point centreG;
    //perimetre est une méthode abstraite, i.e. sans body,
// car son implémentation dépends de la figure
    public abstract int perimetre() ;
    //on ne pourrai pas instancer Geom car elle est abstraite
    // ca n'empeche pas de définir un constructeur qui pourra
    // être appellé par les constructeurs des classes filles.
    public Geom(Point p)
    { centreG =p;}
    @Override
    //programmer un méthode toString permet
    // d'afficher directement un Geom
    // on peut utiliser la fonction perimetre
    // même si elle n'est pas définie.
    public String toString(){
        return "figure de centre de gravité " + centreG +
        " de perimetre " + perimetre() +
        (aUnAxeDeSymetrieVertical()?"symetrique ":"pas
            symetrique");
    }

    // le comportement le plus courant de la méthode
    // aUnAxeDeSymetrieVertical() est de renvoyer vrai,
    // on la définie donc comme cela par défaut.
    public boolean aUnAxeDeSymetrieVertical(){return true;}
}

class Circle extends Geom{
    private int rayon;
    public Circle(Point p, int r) {
        super(p); //on fait appel au constructeur de la classe
            mère.
        rayon=r;
    }
    public int perimetre(){return (int) (rayon*3.14*2);}

    @Override
    public String toString() {
        return super.toString() +
                " qui est un cercle de rayon "+rayon;
    }
}

class Rectangle extends Geom{
    private int longueur,largeur;
    public int perimetre(){return 2*(longueur+largeur);}
    public Rectangle(Point p, int lo, int la) {
        super(p);
        longueur=lo; largeur=la;
    }

    @Override
    public String toString() {
        return super.toString() + "" +
                " qui est un rectangle de coin haut gauche "+
        //on peut accéder a centreG car il est en visibilité "
            protected"
                (centreG.x - (longueur/2) )+ " "+
                (centreG.y - (largeur/2)) ;
    } }


class Triangle extends Geom{
    private int base,cote1,cote2;
    // publicint perimetre(){return base+cote1+cote2;}
    public Triangle(Point p, int b,int lo, int la) {
        super(p);
        base=b; cote1=lo; cote2=la;
    }
    // lorsqu'on définit triangle, dés que l'on spécifie que
    // c'est une sous classe de Geom, intellij demande
    // a ce que soit programmé la méthode périmetre.
    @Override
    public int perimetre() {return base+cote1+cote2;}
    @Override
    //ici et seulement ici on doit préciser le comportement de
    // la méthode aUnAxeDeSymetrieVertical() en la rédefinissant
    public boolean aUnAxeDeSymetrieVertical(){return cote1==
        cote2;}
    @Override
    public String toString() { return super.toString() + " qui
        est un triangle" ;
    }
}
```

5.3 Memoisation

Le petit programme suivant utilise la mémoisation pour programmer fibonaci efficacement. Le but est de préparer le TD ou cette notion de mémoisation sera réalisée.

```
import java.util.HashMap;
import java.util.Map;
import java.util.function.Function;

public class Fibo implements Function<Integer, Integer> {
    @Override
    public Integer apply(Integer n) {
        //on affiche ce qui se passe pour suivre les appels
        System.out.println("calcule fibo "+n);
```

```
        if (n<2) return 1;
        else return (fibo(n−1))+fibo(n−2);
    }
 //Fibo est une classe, faut l'instancier pour l'utiliser
// on encapsule la creation dans une méthode, pour factoriser
     public Integer fibo(Integer n) {return new Fibo().apply(n)
         ;}
    public static void main(String[] args){
        //cet appel rame
        // System.out.println((new FiboMemoized()).apply(45));
        //cet appel est instantané
        System.out.println((new FiboMemoized()).apply(100));
    }
}

 class FiboMemoized extends Fibo{
    //on utilise un seul cache c'est donc une variable globale
    // ==> on la déclare avec le mot clef "static"
     private static Map<Integer,Integer> cache =new HashMap
         <>();
     public Integer apply(Integer n) {
         //on calcule la fonction seulement si c'est pas déja
             fait
         if(!cache.containsKey(n))
             cache.put(n,super.apply(n));
         return cache.get(n);
     }
     // faut penser a rédefinir fibo aussi, pour que les appels
     // récursifs soient aussi mémoisés.
     public Integer fibo(Integer n) {return new FiboMemoized
         ().apply(n);}

}
```

6 Conception/Spécification d'opération

Ce cours porte sur les information que on ne peut exprimer dans les diagramme de classes. On distingue 1- les préconditons/Postconditions, 2- les invariants, 3- les diagrammes de séquence objets.

1. Lorsqu'on décrit les méthodes, les préconditions portent sur les parametres, et les postconditions, sur le résultats. On faisait déja cela pour les cas d'utilisation détaillés. On donnera qq exemples. C'est le théme central du TP de cette semaine.
2. Les invariants sont décrits par des slides, commentés sur `https://www.youtube.com/watch?v=a3DWVNWg2Oo` Dans ce premier cours sur les invariants, on traite les trois premier slides et un exemple simple d'invariant illustré sur les nombres complexes, ainsi que egalement la notion d'attribut dérivés. La notion d'invariant sera revue plus en détail dans le TP du cours sur les classes internes.
3. La troisiéme partie étends la notion de diagramme de séquence aux objets, pour concevoir les opération tout en restant à un niveau plus abstrait que le code. Les même diagrammes de séquence qu'on utilise pour dérouler des scenarios regroupant plusieurs acteurs, peuvent être utilisés pour montrer le déroulement de l'éxecution d'une méthode faisant intervenir plusieurs objets. On utilisera aussi cela en TP. Cette partie du cours utilise un set de slides, dont les commentaires sont disponible sur `https://www.youtube.com/watch?v=fPm5NrvmXHc`

Le cours comprends une quatriéme partie pour expliquer le fonctionnement des tables de hashage étudiées en TD.

6.1 Precondition, Postcondition

Un des objectifs de la phase d'analyse est de préciser le comportement attendu de chacune des fonctionnalités du système à développer. Cela apparait dès la première phase d'analyse, dans la présentation en tableau de la description des cas d'utilisation, avec en particulier les lignes suivantes (illustrées pour un cas d'utilisation « vente d'un ticket pour une visite guidée ») :

Précondition. Indique les conditions devant être réunies pour que le cas d'utilisation puisse être activé. Par exemple : l'agent du musée est connecté sur sa machine de caisse.

Postcondition. Indique l'effet obtenu après réalisation du cas d'utilisation. Par exemple : la place est réservée, le billet est imprimé, le paiement est encaissé.

Situations d'erreur. Indique les cas particuliers pouvant donner lieu à des comportements alternatifs. Par exemple : la visite demandée est déjà complète, ou le paiment est impossible.

En cas d'erreur. Indique les effets des comportements alternatifs correspondant aux situations d'erreur. Par exemple : le billet n'est pas imprimé et aucun paiement n'est encaissé (fonctionne dans les deux situations), et éventuellement dans la première situation une autre visite est proposée.

Ce principe se transpose dans les phases de conception, par exemple lors de la définition des classes et de leurs méthodes. Supposons une méthode de la signature suivante :

```
public t m(t1 p1, ..., tn pn)
```

Nous parlons donc d'une méthode `m` prenant en entrée n paramètres `p1` à `pn` de types respectifs `t1` à `tn` et ayant un type de retour `t` (éventuellement, `t` est `void`). Lors de la conception de cette méthode, il convient de préciser les éléments suivants, qui forment la **spécification** de la méthode :

— une précondition qui décrit, dans l'hypothèse d'un appel de méthode `obj.m(o_1, ..., o_n)`, les conditions sur le paramètre implicite `obj` et sur les paramètres `o_1` à `o_n` pour que l'appel de méthode se déroule convenablement ; et

— une postcondition qui décrit le résultat de l'appel de méthode, cette description comportant :

1. une description du résultat renvoyé par la méthode (dans le cas où le type de retour `t` n'est pas `void`), et
2. une description de l'état des objets `obj` et `o_1` à `o_n` (dans le cas où la méthode est susceptible de modifier des attributs de l'un des objets).

Exemple 1. La méthode `double sqrt(double d)` de calcul de racine carrée a la spécification suivante :

Précondition. Le paramètre `d` doit être positif ou nul. Remarque : le fait que le paramètre `d` doivent être de type `double` ne fait **pas** partie de la précondition ; cette information est déjà présente dans la signature de la méthode.

Postcondition.

1. Le résultat renvoyé par la méthode est la racine carrée du paramètre `d`.
2. L'état des objets n'est pas modifié.

Si la méthode est privée, et est donc appelée en interne, depuis une autre méthode de la même classe, il est possible de facilement de démontrer que la précondition est toujours vérifié. Mais si la méthode est appelée par un client, alors il est pertinent de vérifier la précondition immédiatement, de sorte que si elle n'est pas vérifiée, on puisse détecter le bug au plus tôt. **Exemple 2.** La méthode `void add(T o)` d'ajout d'un élément à un `ArrayList` a la spécification suivante :

Précondition. Pas de précondition (la méthode peut toujours être appelée).

Postcondition.

1. Pas de résultat renvoyé.
2. Après l'appel `l.add(o)`, la liste `l` contient les mêmes éléments qu'avant, plus l'objet `o`.

6.2 Les tables de Hashage

C'est une structure de données que nous allons utiliser en TD pour travailler sur la spécification, pour cette raison, nous la traitons en cours afin qu'elle soit bien comprise.

Une combinaison tableau + liste chainée Une table de hachage Hashtable$< K, T >$ est un container générique qui permet de stocker un ensemble d'éléments de type $< T >$ dans un tableau associatif, et de les retrouver a l'aide de clef '(Keys) de type type $< K >$ qui joue le même rôle que les indices pour un tableau. L'ajout et le test d'appartenance d'éléments se faire en temps constant en moyenne, comme si la table était un vulgaire tableau, sauf que les indices qui sont les clefs ne sont pas obligé d'être des nombres entiers.

La structure repose sur une fonction de codage, appelée "hashcode" qui à chaque élément `elt` associe un entier `c` dont on déduit l'indice de la case du tableau dans laquelle l'élément doit être placé (On prends ce code modulo le nombre total de cases). Notez qu'il est possible que deux éléments distincts soit associés au même code (on parle de *collision*), ou qu'un code ne soit associé à aucun élément. Chaque case du tableau ne contient donc pas un unique élément, mais un paquet de zéro, un ou plus d'éléments. On utilise une liste chainée (structure de donnée dynamique) pour réprésenter un paquet, car leur taille doit augmenter au fur et a mesure qu'on ajoute des élements. Au final, la structure combine donc tableau et liste chainée : c'est un tableau de listes chainées. Le tableau permet un addressage simple et rapide des cases, la liste chainée donne la propriété de structure de données dynamique, i.e. le nombre d'elements contenable n'est pas borné.

Cependant, la manipulation de hastTable est efficace dans la mesure où les paquets restent en général petits, plus précisément d'une taille bornée que l'on note O(1) en complexité. Pour assurer cela, on augmente le nombre de paquet lorsque le nombre moyen d'elements par paquet dépasse un seuil fixe. On augmente en le multipliant par deux, le cout important de cette manip sera amorti par le fait que on n'aura pas besoin de la refaire pendant longtemps.

La table permet de retrouver un element donné très rapidement, c'est a dire en un temps borné O(1) en moyenne, cela constitue un gain de temps très important pour les grosses tables. Donner un exemple.

Les tableaux associatifs On utilise en général les tables de hashage non pas pour stoquer un ensemble d'élement, mais pour représenter une association d'un ensemble de clés vers des valeurs. Les valeurs sont stoquée avec leur clef, en cherchant les clef, on retrouve donc la valeur associée. Un tel tableau s'appelle un tableau associatif. Tout comme les tableaux ordinaires, les tables de hachage associative permettent un accès en O(1) en moyenne, quel que soit le nombre de paires clé–valeur dans la table. Toutefois, comme plusieurs paires clé–valeur peuvent se trouver dans la même paquet, le temps d'accès dans le pire des cas peut être de O(n) pour une table de n elements. Comparées aux autres tableaux associatifs, les tables de hachage sont surtout utiles lorsque le nombre de paires clé–valeur est très important

6.3 Invariant

La phase de conception intervient une fois complétée l'analyse du système à construire, et définit les structures de données et l'organisation du code. La phase d'analyse s'intéressait au "quoi", la phase de conception se concentre sur le "comment".

Prenons une classe représentant les nombres complexes. La phase d'analyse lui a associé quatre attributs : - partie réelle [re] - partie imaginaire [im] - module [rho] - argument [theta] Ces arguments ne sont pas indépendants les uns des autres. Un premier constructeur : il prend en paramètre une partie réelle et une partie imaginaire, et en déduit le module et l'argument.

```
class Complexe {
private double re, im, rho, theta;
public Complexe(double r, double i) {
    this.re  = r; this.im      = i; updatePolaire();

    private void updatePolaire(){
    rho=Math.sqrt(re*re+im*im);
    if (rho!=0) this.theta  = Math.arctan(i/r);
    else this.theta=Math.PI/2.0; }
```

Dans de telles conditions, toute modification de l'un des attributs doit être reflétée sur les autes. La relation entre les quatre attributs [r], [img], [rho] et [theta] est un invariant de notre classe [Complexe]. Une des raisons pour lesquelles les "setters" (méthodes publiques qui permettent de modifier

la valeur d'un attribut) sont dangereux, est qu'ils risquent d'invalider les invariants d'un objet s'ils sont mal écrits. En l'occurrence, si l'on veut fournir un setter pour la partie imaginaire d'un nombre complexe, alors il ne faut pas modifier uniquement l'attribut [img], mais aussi mettre à jour [rho] et [theta]. En conséquence.

```
public void setImg(double i) {
this.img = i; updatePolaire();}
```

Dans cette version nous avons donc :

- plusieurs attributs
- un accès immédiat aux valeurs de ces attributs
- un coût à payer pour faire des mises à jour à chaque fois que l'un des attributs est modifié.

Attributs dérivés, remplacés par un getter Un attribut est dit dérivé si leur valeur peut se calculer à partir des valeurs d'un ou plusieurs autres attributs. Les attributs [rho] et [theta] sont dérivés. En UML, un attribut dérivé doit être signalé en préfixant le nom de l'attribut avec une barre oblique « / » afin d'indiquer au lecteur que cet attribut est redondant. On peut remplacer un attribut dérivé par un getter, En éliminant les informations redondantes dans la classe, on élimine aussi la nécessité de mettre à jour les attributs dérivés.

```
class Complexe {
 public double getRho() {
     return Math.sqrt(this.rthis.r    + this.imgthis.img);  }
     public double gettheta() {...  }
 }
```

Dans cette deuxième version nous avons donc : 1- des attributs réels 2- un coût à payer pour accéder aux valeurs des attributs dérivés.

Il existe une troisiéme version qui consiste à On utilise un booléen "ajour" qui indique si la valeur de theta et rho sont à jour. Lors de la lecture de [rho,theta] on consulte l'attribut [ajour] pour savoir si les coordonnées polaires sont à jour ou si faut les recalculer. Lors de la création d'un Complexe, ou lors de la modification de [img], on ne met pas à jour les autres attributs, mais on signale qu'un changement a été apportée avec le booléen [ajour].

```
    private boolean ajour;

    public Complexe(double r, double i) {
        this.r  = r; this.img       = i;  this.ajour=false   }

    public verifieAjour(){   if  (!this.ajour)
     {updateRho(); updateTheta();this.ajour=true;} }

    public double getRho() {
        verifieAjour();    return   this.rho;} }

    public void setImg(double i) {
        this.img = i;  this.ajour   = false;}
}
```

7 Test

7.1 Les différentes sortes de tests

Effectuer un test c'est : faire tourner un morceau de programme ou un système entier, et contrôler que les résultats obtenus sont bien ceux attendus.

Le test est le moyen le plus simple à mettre en œuvre pour s'assurer qu'un programme fonctionne à peu près comme prévu. C'est un effort minimal et indispensable, à utiliser en toutes circonstances, de l'exercice de TP quelconque au grand projet industriel. Dans certains cas, le test sert même de guide au développement.

Différents types de test ont lieu à différents stades du développement. On a notamment :

- Lors du déploiement d'un système entier, des utilisateurs le testent en conditions réelles (**tests alpha, beta**...). Dans le diagramme en V, cette phase de test apparaît tout en faut à droite et valide le cahier des charges.
- Conjointement à l'écriture du code d'une classe, le **test unitaire** s'intéresse à chaque méthode indépendamment l'une de l'autre. Il s'agit de vérifier, dans des conditions contrôlées, que le comportement de chaque méthode est cohérent avec sa spécification, qui a été déterminée lors de la phase de conception.
- Entre les deux, les **tests d'intégration** testent des scénarios faisant intervenir plusieurs composants du système.
- En phase de maintenance, chaque modification du code appelle des **tests de régression**, qui visent à vérifier que les corrections ou évolutions n'ont pas introduit de problèmes dans les parties qui fonctionnaient précédemment.

Les tests unitaires et tests de régression sont en général nombreux et effectués souvent : ils sont donc automatisés. Dans ce cours, on regardera l'automatisation des tests unitaires pour Java avec l'outil JUnit.

7.2 Test unitaire

Un test unitaire s'intéresse au comportement d'une unique méthode. Une méthode `m` étant donnée, un test unitaire est défini par :

1. des entrées concrètes, c'est-à-dire un état de l'objet appelant et des paramètres pour l'appel de méthode, ces entrées devant respecter les préconditions de la méthode ;
2. le résultat devant être renvoyé d'après la spécification, si la méthode n'a pas un type de retour `void` ;
3. l'état attendu d'après la spécification, si la méthode modifie des attributs.

Le test consiste à effectuer l'appel de méthode pour les objets donnés en entrée, et de comparer le résultat obtenu au résultat attendu.

- Si les résultats correspondent, alors le comportement du programme est cohérent avec ce qui était attendu. Cela ne signifie pas que le programme est correct à coup sûr, mais de nombreux tests (bien choisis) réussis apportent une certaine confiance.

— Si les résultats ne correspondent pas, alors on vient de mettre au jour une erreur, et le programme doit être corrigé.

Exemple 1. Quatre tests pour la méthode `sqrt` sont donnés par le tableau suivant :

	Entrée	Sortie attendue
Test 1	`4.0`	`2.0`
Test 2	`1.0`	`1.0`
Test 3	`0.0`	`0.0`
Test 4	`1048576.0`	`1024.0`

Il serait absurde de donner un test avec une entrée négative : la précondition de la méthode `sqrt` demande une entrée positive ou nulle, et la spécification ne dit donc rien de ce qui devrait se passer dans le cas contraire. Autrement dit, nous n'avons aucune idée de la sortie qui serait attendue (à supposer qu'il y ait bien une sortie).

Exemple 2. Trois tests pour l'appel de méthode `l.add(e)` sont donnés par le tableau suivant :

	`l`	`e`	`l` attendu après appel
Test 1	vide	1	`{1}`
Test 2	`{1, 2, 4}`	8	`{1, 2, 4, 8}`
Test 3	`{1, 2, 4}`	2	`{1, 2, 4, 2}`

7.3 JUnit

Avec JUnit, les tests sont définis dans des fichiers `.java` à côté des fichiers de code. Un fichier de test contient une classe publique, et les tests sont des méthodes de cette classe.

Un test est défini par une méthode sans paramètre et sans valeur de retour (sa signature est donc de la forme `public void m()`), et précédée de l'annotation `@Test`.

Une méthode de test comporte trois parties :

1. une initialisation, qui doit construire les entrées à fournir à l'appel de méthode : il faut définir les objets sur lesquels portera le test et les placer dans l'état spécifié ;
2. l'appel de méthode ;
3. un diagnostic, qui doit vérifier le résultat obtenu et les éventuelles modifications ou non-modifications d'objets, en utilisant des assertions table 2 :

Lorsque tous les tests d'une classe de test ont une partie de leur initialisation en commun, on peut inclure dans la classe une méthode précédée de l'annotation `@Before` qui contient ce code d'initialisation commun.

Lorsque les tests sont effectués, toutes les méthodes annotées par `@Test` sont appelées l'une après l'autre, et l'éventuelle méthode annotée par `@Before` est appelée avant chaque appel d'une méthode de test. Si tous les tests ont réussi, l'outil rapporte un succès. Si des tests échouent (car le résultat n'est pas celui attendu ou car une erreur a eu lieu), l'outil indique quels tests ont posé problème.

Dans les cas simples, la méthode `@Before` redéfinit les principaux objets utilisés avant chaque test, et on peut donc considérer qu'on repart de zéro. Si certaines choses ne sont pas réinitialisées de cette manière, on peut ajouter une méthode annotée par `@After` qui sera appelée après chaque appel d'une méthode de test.

Un exemple de fichier de test. ci dessous.

```
import static org.junit.Assert.*;
import org.junit.Test;

public class EnsembleTest {
  // Vérifier que l'ensemble vide a pour cardinal 0.
  @Test
  public void ensembleVideEstVide() {
    // Trois étapes
    // 1. Construire l'exemple (l'ensemble vide)
    Ensemble e = new Ensemble();
    // 2. Appeler la méthode (exécuter le test)
    int res = e.cardinal();
    // 3. Comparer résultat attendu et résultat obtenu
    assertEquals(0, res);
    assertFalse(e.contient(1));
  }

  // Un ensemble àun élément a pour cardinal 1.
  @Test
  public void cardinalSingleton() {
 // On construit un ensemble avec uniquement l'élément 1
    Ensemble e = new Ensemble();
    e.ajoute(1); // On ajoute l'élément 1
    int res = e.cardinal();
    assertEquals(1, res); }

  // Ajouter un élément déjà présent ne change rien
  @Test
  public void ajoutEltDejaPresent() {
 //On construit un ensemble avec uniquement l'élément 1
    Ensemble e = new Ensemble();
    e.ajoute(1); // On ajoute l'élément 1
    e.ajoute(1); // On ajoute a nouveau l'élément 1
    // On vérifie qu'il n'y a toujours qu'un élément
    assertEquals(1, e.cardinal());
    assertTrue(e.contient(1)); }}
```

Mise en oeuvre Junit marche aussi bien sous eclipse que sous intellij. Il faut faire attention à inclure dans le class path la librairie Junit. Pour Intellij, il y a une autre librairie à inclure, appelée "Hamcrest". Pour éviter les problèmes, ont vous a déjà donné ces librairies dans l'archive du TP.

Également, avec intelij, il faut marquer le dossier qui contient les fichier java de tests. Sélectionner, et faire mark as test sources. Le dossier prends alors la couleur verte.

Developement dirigé par les tests. On regroupe tout les tests d'une classe Toto, dans une classe appelée TotoTest. L'environnement junit, permet d'exécuter tout les tests contenu dans une classe TotoTest, en même temps. Les tests qui s'exécute correctement s'affiche en vert, les autres en rouge si une exception est générée, ou en jaune si l'assertion n'est pas vérifiée, avec de plus une explication de pourquoi l'assertion n'est pas vérifiée. Avec le "Développement dirigé par les tests" on écrit les tests avant le code, ceux ci seront donc jaune ou rouge. On les fait passer aux vert, ensuite, en écrivant le code qui vérifie les tests. C'est ce que vous ferez en TPs.

Méthode	Rôle
`assertEquals(Object a, Object b)`	Vérifie que les objets a et b sont égaux
`assertSame(Object a, Object b)`	Vérifie que a et b sont des références vers le même objet
`assertNotSame(Object a, Object b)`	Vérifie que a et b ne sont pas des références vers le même objet
`assertNull(Object o)`	Vérifie que l'objet o est `null`
`assertNotNull(Object o)`	Vérifie que l'objet o n'est pas `null`
`assertTrue(boolean e)`	Vérifie que l'expression e est vraie
`assertFalse(boolean e)`	Vérifie que l'expression e est fausse
`fail()`	Provoque l'échec du test

Table 2 – Catalogue d'assertions

7.4 Couverture de code

En génie logiciel, la couverture de code est une mesure utilisée pour décrire le taux de code source exécuté d'un programme quand une suite de test est lancée. Un programme avec une haute couverture de code, mesurée en pourcentage, a davantage de code exécuté durant les tests ce qui laisse à penser qu'il a moins de chance de contenir de bugs logiciels non détectés, comparativement à un programme avec une faible couverture de code. Il y a de nombreuses méthodes pour mesurer la couverture de code. Les principales sont :

- Couverture des fonctions (Function Coverage) - Chaque fonction dans le programme a-t-elle été appelée ?
- Couverture des instructions (Statement Coverage) - Chaque ligne du code a-t-elle été exécutée et vérifiée ?
- Couverture des points de tests (Condition Coverage) - Toutes les conditions (tel que le test d'une variable) sont-elles exécutées et vérifiées ? (Le point de test teste-t-il ce qu'il faut ?)
- Couverture des chemins d'exécution (Path Coverage) - Chaque parcours possible (par exemple les 2 cas vrai et faux d'un test) a-t-il été exécuté et vérifié ?

Certaines méthodes sont liées, par exemple :

La couverture des chemins implique à la fois la couverture des instructions et la couverture des points de tests ; La couverture des instructions n'implique pas la couverture des points de tests, comme le montre le code ci-dessous (écrit en langage C) :

```
void foo(int bar)
{  printf("this is ");
   if (bar < 1) { printf("not "); }
   printf("a positive integer"); }
```

Si la fonction foo est appelée avec l'argument bar = -1, alors on assure la couverture des instructions (elles sont toutes exécutées) ; par contre, la couverture des points de tests n'est pas assurée. Faudrait pour cela une execution avec bar >= 1, pour que le test ne soit pas vérifié.

Une couverture complète des chemins est généralement très difficile à accomplir, voire impossible : Tout code contenant une succession de n points de tests contient potentiellement 2^n chemins différents ; les boucles peuvent aboutir à un nombre infini de chemins possibles ; Certains chemins sont parfois infranchissables, dans le sens où il n'existe pas de paramètre d'entrée du programme permettant de suivre ce chemin en particulier (les tests empêchant ce chemin sont souvent placés « par sécurité » au niveau d'une fonction) ; Il n'existe pas d'algorithme général déterminant les chemins infranchissables (sinon un tel algorithme permettrait de résoudre le problème de l'arrêt).

Pour faciliter les tests, on instrumente le code source de façon à tracer l'exécution au cours des tests. La « trace » résultante est ensuite analysée pour identifier les chemins et les zones de code qui n'ont pas été vérifiés, puis les tests sont mis à jour (si nécessaire) pour améliorer la couverture. Combinée à d'autres méthodes, il est possible d'aboutir à un ensemble abordable (réaliste) de tests de régression.

La couverture de code accomplie est exprimée sous forme de pourcentage du code total. La signification de ce pourcentage dépend des types de couvertures effectuées : tester « 67 % des chemins possibles » est plus utile que « 67 % des instructions ».

8 Cours dedié au projet.

8.1 Le patron de conception MVC

Découpage en trois Modèle-Vue-Vontrôleur ou MVC est un patron de conception logiciel destiné aux interfaces graphiques. Il est composé de trois types de modules ayant trois responsabilités différentes : les modèles, les vues et les contrôleurs.

- Le modèle contient les données à afficher.
- La vue contient la présentation de l'interface graphique.
- Le contrôleur contient la logique concernant les actions effectuées par l'utilisateur.

Mise en œuvre MVC La vue implemente une interface Observer qui déclare une méthode void update(), qui met à jour la vue, le plus simple étant de tout "repeindre" ce qui est dedans. Le modéle implémente une interface Observable, comportant un attribut avec la liste des observer qui l'observe. Cette liste est mise a jour avec une méthode addObserver(o :observer). Une autre méthode notifyObservers() déclenche la méthode update, chez la vue. Ces différent élements sont utilisés dans le fichier conway.java qui simule le jeu de la vie. Vous pouvez donc les récupérez la dedans.

Flux de traitement Lorsqu'un client envoie une requête à l'application : la requête envoyée depuis la vue est analysée par le contrôleur. le contrôleur demande au modèle d'effectuer les traitements approprié et notifie à la vue que la requête est traitée. la vue notifiée consulte le modèle pour se mettre à jour, i.e. repeindre ou bien tout, ou bien seulement ce qui a été modifié.

Avantage du modèle MVC C' est la simplicitié de l'architecture qu'il impose. Cela simplifie la tâche de du projet. En particulier, la modification du controle ne change en rien tout ce qui concerne la vue.

Le controleur Dans une architecture MVC, on lance les fenêtres, et c'est ensuite les actions des clients (pour nous les joueurs) qui font progresser le jeu ; Ces actions arrivent au au controleur, lequel maintient un état qui permet de savoir ou on en est dans les differentes phases du jeu : par exemple, est on en train de saisir une action d'un joueur, ou bien en train de l'executer ? et la quelle action de quel joueur ?

Mise en œuvre Controleur Parmis les elements graphiques, se trouvent les boutons "de commande" qui permettent de saisir les actions des joueurs. Les actions déclenché par ces boutons (lorsqu'ils sont pressés) devront modifier l'état du controleur, pour refléter la progression du jeu. L'exemple de Conway est trés simple à l'extréme, car le controleur n'a qu'un seul état. Cependant, elle est suffisante pour récupérer les elements de code java nécessaire.

8.2 La bibliothèque Swing de java.

Swing est une bibliothèque graphique pour le langage de programmation Java, faisant partie du package Java Foundation Classes (JFC), inclus dans J2SE. Swing offre la possibilité de créer des interfaces graphiques identiques quelque soit le système d'exploitation sous-jacent. Il utilise le principe Modèle-Vue-Contrôleur (MVC) et dispose de plusieurs choix d'apparence (de vue) pour chacun des composants standard. Swing permet de construire des interfaces graphiques en Java affichable sur toute JVM (Java Virtual Machine) de n'importe quel système.

Pour créer des interfaces graphiques, il est nécessaire de connaître et de maîtriser certains objets de base de Swing :

Les JFrame Les JFrame sont l'équivalent des fenêtres. Elles ont un titre, une dimension, un aspect et des éléments graphique affichés à l'intérieur.

Les JFrame font partie du package javax.swing, il faudra l'inclure avant toute utilisation avec un import.

```
import javax.swing.*;
public  class test
{  public static void main(String args * )
    {   JFrame  f  = new JFrame("Hello World!!");
        f.setVisible(true);
    } }
```

Tapez ce code dans "test.java" , compilez, lancez .

Vous verrez une petite fenêtre assez minuscule, si vous la redimensionnez à la main, vous verrez apparaître son titre et son contenu vide. Comme on n'a pas spécifié de dimension pour notre JFrame, Swing ouvre une fenêtre avec une dimension par défaut (ici 0x0). Nous verrons plus loin, que l'on n'a pratiquement jamais besoin de spécifier nous même la dimension d'une JFrame.

Par défaut, une JFrame est toujours invisible, c'est-à-dire que la fenêtre est créée mais jamais affichée par défaut. C'est pour cela que l'on rajoute la ligne f.setVisible(true) ; qui permet de rendre visible la fenêtre.

Lorsque vous fermez la fenêtre,vous constaterez que Java ne rends pas la main au système, le programme java tourne toujours. Lorsque vous créez une JFrame, un Thread c'est-à-dire un programme d'arrière plan est créé. Par défaut, Swing ne tue pas le processus lorsque l'on ferme la JFrame avec la souris.

Jframe complet Notre deuxième petit programme va nous ouvrir une fenêtre avec une dimension par défaut et va quitter le logiciel lorsqu'on ferme la fenêtre avec la souris :

```
import  javax.swing.*;
import  java.awt.*;
public class test
{ public static void main(String args[]) {
        JFrame f=new JFrame("Hello World!!");
        f.setSize(new Dimension(200,200));
        f.setDefaultCloseOperation(
            JFrame.EXIT_ON_CLOSE);
        f.setVisible(true);  } }
```

On n'a rajouté que deux lignes : - setSize permet de spécifier la dimension de la fenêtre - setDefaultCloseOperation permet de définir l'opération par défaut que l'on veut lorsqu'on ferme la fenêtre. Le paramètre `JFrame.EXIT_ON_CLOSE` spécifie que l'on quitte le logiciel lorsqu'on ferme la JFrame.

Jpanel, LayoutManager Vous voyez donc qu'il n'est pas très difficile d'ouvrir une fenêtre avec Swing. Ouvrir une fenêtre seule sans rien à l'intérieur n'a pas grand intérêt, nous allons voir maintenant comment placer des composants à l'intérieur. Contenu possible des fenêtres : les JPanel, les layouts et les JComponent.

Pour ajouter des objets à une JFrame, on a besoin de JPanel. Un JPanel est en quelque sorte une boite dans laquelle on peut placer des composants de l'interface graphique. Un JPanel sert uniquement à stocker les objets, il est secondé par un LayoutManager pour les placer au bon endroit dans une fenêtre.

En effet, dans Swing, on place les objets non pas à une abscisse/ordonnée précise mais on les place avec une stratégie gérée par les LayoutManager. C'est le LayoutManager qui se charge de définir la position de l'objet dans la fenêtre. Par exemple, une des stratégies possible est de placer tous les objets en ligne les uns à cotés des autres. Un JPanel possède une unique stratégie de placement, un unique LayoutManager.

Les composants de l'interface graphique sont des classes qui héritent de la classe JComponent. Il en existe déjà un bon nombre par défaut dans Swing : les labels, les zones de texte éditables, les scrollbars, les boutons, les tables etc... et il est bien sur possible de créer ses propres composants (sujet peut être d'un futur tutorial).

Voici une petite liste des noms des classes des composants :

- JLabel : un label (un texte)
- JTextField : un champ texte éditable
- JButton : un bouton
- JComboBox : permet de selectionner une option dans un menu

Vous verrez, qu'il est très facile de créer et de placer ces objets.

Pour commencer, nous allons rajouter à notre exemple précédent, un texte centré sur la largeur de la fenêtre. Le texte sera un JLabel. On créé simplement cet objet en faisant : `JLabel lab=new JLabel("Bonjour tout le monde");`

Un composant ne sert a rien si on a pas de JPanel où le stocker. Pour créer un JPanel : `JPanel pan=new JPanel();`

Il faut lui associer un LayoutManager, il en existe beaucoup dans Swing. Nous allons utiliser un FlowLayout qui a la particularité de pouvoir placer notre JLabel centré sur la largeur de la fenêtre : `FlowLayout bl = new FlowLayout(FlowLayout.CENTER)` Le paramètre FlowLayout.CENTER spécifie l'alignement que l'on désire, ici on veut centrer. Pour attacher le LayoutManager auPanel : pan.setLayout(bl) ; Enfin, il ne faut pas oublier d'ajouter le JLabel au panel (une des erreurs classiques !) : pan.add(lab) ; Pour finir, une JFrame lors de sa création, créé déjà un JPanel, ici nous avons créé notre propre JPanel pan, une deuxieme erreur classique est d'oublier de redéfinir le panel de la JFrame(sinon notre JPanel ne sera pas affiché !), on le fait tout simplement par : setContentPane(pan)

Voici le programme complet :

```
import  javax.swing.*;
import  java.awt.*;
public  class    test extends JFrame
{
    public test()
    {   super("Hello World !");
        setSize(new Dimension(200,200));
        JPanel pan=new JPanel();
        FlowLayout bl = new  FlowLayout(
                  FlowLayout.CENTER);
        pan.setLayout(bl);
        JLabel  lab=new JLabel(
               "Bonjour le monde!!");
        pan.add(lab);
        setContentPane(pan);
        setVisible(true);
       setDefaultCloseOperation(
            JFrame.EXIT_ON_CLOSE);
    }
    public static void main(String args[])
    {  new test();   }}
```

Imbrication de Jpanel On peut placer d'autre JPanel dans un JPanel. Chaque sous panel ayant son propre LayoutManager, on peut créer ainsi une interface graphique complète. Voici un exemple illustrant cela :

```
import  javax.swing.*;
import  java.awt.*;
public class test extends JFrame
{
    public test()
    {   super("Hello World!");
        setSize(new Dimension(300,300));
        JPanel pan=new JPanel();
        BoxLayout    bl=new BoxLayout(
            pan,BoxLayout.Y_AXIS);
        pan.setLayout(bl);
        JLabel  lab=new JLabel("Bonjour le monde!!");
        pan.add(lab);
        JTextField  tf=new JTextField("Editez !");
        pan.add(tf);
        JPanel  pan2=new   JPanel();
        bl=new BoxLayout(pan2,BoxLayout.X_AXIS);
        pan2.setLayout(bl);
        lab=new JLabel("ComboBox:");
        pan2.add(lab);
        String  c[] ={"Un","Deux",
                      "Trois","Zéro !"};
        JComboBox   cb=new JComboBox(c);
        pan2.add(cb);
        //ajoute le panel 2 dans le panel 1!
        pan.add(pan2);
        //un dernier composantpour la route...
        JButton but=newJButton(
                   "C'est un bouton !!");
        pan.add(but);
        setContentPane(pan);
        setVisible(true);
        setDefaultCloseOperation(
               JFrame.EXIT_ON_CLOSE);
    }

    public static void main(String args[])
    { new test();   } }
```

8.3 Exemple de modéle MVC

On commente ici le programme conway.java qui est disponible dans votre archive eTDTP.tar.gz . Ce programme est un exemple elémentaire d'architecture MVC, et par ailleurs, il ressemble a ce que vous devez produire, sur la premiére itération de votre projet. Un principe directeur est la séparation stricte des deux parties suivantes :

- Le coeur de l'application, appelé le modèle, où est fait l'essentiel du travail.
- L'interface utilisateur, appelée la vue, qui à la fois montre des choses à l'utilisateur et lui fournit des moyens d'interagir.

Notre cas d'étude : le jeu de la vie de Conway. Une grille bidimensionnelle de dimensions finies est peuplée de cellules pouvant être vivantes ou mortes. À chaque tour un nouvel état est calculé pour chaque cellule en fonction de l'état de ses voisines immédiates. Un bouton permet de passer au tour suivant (on dit aussi la génération suivante).

Lien observateur/observé entre vue et modèle : les informations montrées à l'utilisateur reflètent l'état du modèle et doivent être maintenues à jour. Pour réaliser cette synchronisation, on peut suivre le schéma de conception observateur/observé, dont le principe est le suivant :

- Un observateur (en l'occurrence la vue) est lié à un objet observé et se met à jour pour refléter les changement de l'observé.
- Un observé est lié à un ensemble d'objets observateurs et les notifie de tout changement de son propre état.

Java fournit une interface [Observer] (observateur) et une classe [Observable] (observé) assurant cette jonction. Voici une manière sommaire de les recoder.

```
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/** Interface des objets observateurs.*/
interface Observer {
  /** Un observateur doit posséder une méthode [update]
    déclenchant la mise àjour. */
    public void update();
  /** La version officielle de Java possède des paramètres
  précisant le changement qui a eu lieu. */
}

/** Classe des objets pouvant être observés.*/
abstract class Observable {
  /** On a une liste [observers] d'observateurs, initialement
    vide, àlaquelle viennent s'inscrire les observateurs
     via la méthode [addObserver]. */
    private ArrayList<Observer> observers;
    public Observable() {
        this.observers = new ArrayList<Observer>();
    }
    public void addObserver(Observer o) {
        observers.add(o);
    }

    /** Lorsque l'état de l'objet observé change, il est
    convenu d'appeler la méthode [notifyObservers] pour
    prévenir l'ensemble des observateurs enregistrés.
     On le fait ici concrètement en appelant la méthode
     [update] de chaque observateur. */
    public void notifyObservers() {
        for(Observer o : observers) {
            o.update();
        }
    }
}
```

Nous allons commencer à construire notre application, en voici la classe principale. L'amorçage est fait en créant le modèle et la vue, par un simple appel à chaque constructeur. Ici, le modèle est créé indépendamment (il s'agit d'une partie autonome de l'application), et la vue prend le modèle comme paramètre (son objectif est de faire le lien entre modèle et utilisateur).

```
    public static void main(String[] args) {
        CModele modele = new CModele();
        CVue vue = new CVue(modele); }
```

Le modèle : le coeur de l'application. Le modèle étend la classe [Observable] : il va posséder un certain nombre d'observateurs (ici, un : la partie de la vue responsable de l'affichage) et devra les prévenir avec [notifyObservers] lors des modifications. Voir la méthode [avance()] pour cela.

```
public class Conway {
class CModele extends Observable {
    /** On fixe la taille de la grille. */
    public static final int HAUTEUR=40, LARGEUR=60;
    /** On stocke un tableau de cellules. */
    private Cellule[][] cellules;

  /** Construction : on initialise un tableau de cellules.*/
    public CModele() {
        /** Pour éviter les problèmes aux bords, on ajoute une
        ligne et une colonne de chaque côté, dont les cellules
         n'évolueront pas.*/
        cellules = new Cellule[LARGEUR+2][HAUTEUR+2];
        for(int i=0; i<LARGEUR+2; i++) {
            for(int j=0; j<HAUTEUR+2; j++) {
                cellules[i][j] = new Cellule(this,i, j);
            }
        }
        init();
    }

   /** Initialisation aléatoire des cellules, exceptées celles
   des bords qui ont été ajoutés. */
    public void init() {
        for(int i=1; i<=LARGEUR; i++) {
            for(int j=1; j<=HAUTEUR; j++) {
                if (Math.random() < .2) {
                    cellules[i][j].etat = true;
                } } } }

    /** Calcul de la génération suivante. */
    public void avance() {
        /** On procède en deux étapes.
          1−pour chaque cellule on évalue quel sera son état
           àla prochaine génération.
          2− Ensuite, on applique les évolutions
           qui ont été calculées. */
        for(int i=1; i<LARGEUR+1; i++) {
            for(int j=1; j<HAUTEUR+1; j++) {
                cellules[i][j].evalue();
            }
        }
        for(int i=1; i<LARGEUR+1; i++) {
            for(int j=1; j<HAUTEUR+1; j++) {
                cellules[i][j].evolue();
            }
        }
        /** Pour finir, le modèle ayant changé, on signale aux
```

```
            observateurs qu'ils doivent se mettre àjour.*/
        notifyObservers();
    }

    /** Méthode auxiliaire : compte le nombre de voisines
     vivantes d'une cellule désignée par ses coordonnées. */
    protected int compteVoisines(int x, int y) {
        int res=0;
        /** Stratégie simple: on compte les cellules vivantes
        dans le carré 3x3 centré autour des coordonnées (x, y),
         puis on retire 1 si la cellule centrale est vivante.
         On n'a pas besoin de traiter àpart les bords du
         tableau de cellules grâce aux lignes et colonnes
         supplémentaires qui ont été ajoutées de chaque côté
         (dont les cellules sont mortes et n'évolueront pas).
              */
        for(int i=x-1; i<=x+1; i++) {
            for(int j=y-1; j<=y+1; j++) {
                if (cellules[i][j].etat)      { res++; }
            }
        }
        return (res - ((cellules[x][y].etat)?1:0));
/** L'expression [(c)?e1:e2] prend la valeur de [e1] si
         [c] vaut [true] et celle de [e2] si [c] vaut [false].
         Cette dernière ligne est donc équivalente à
         int v;
         if (cellules[x][y].etat)      { v = res - 1; }
         else { v = res - 0; }
         return v; */
    }

    /** Une méthode pour renvoyer la cellule aux coordonnées
     choisies (sera utilisée par la vue). */
    public Cellule getCellule(int x, int y) {
        return cellules[x][y];     }
    /** Notez qu'à l'intérieur de la classe [CModele],
     la classe interne est connue sous le nom abrégé
     [Cellule]. Son nom complet est [CModele.Cellule],
     et cette version complète est la seule àpouvoir
     être utilisée depuis l'extérieur de [CModele].
      Dans [CModele], les deux fonctionnent.
     */
}
```

Définition d'une classe pour les cellules. Cette classe fait encore partie du modèle.

```
class Cellule {
  /** On conserve un pointeur vers la classe
      principale du modèle. */
    private CModele modele;

    /** L'état d'une cellule est donné par un booléen. */
    protected boolean etat;
    /** On stocke les coordonnées pour pouvoir les passer
      au modèle lors de l'appel à[compteVoisines].
     */
    private final int x, y;
    public Cellule(CModele modele, int x, int y) {
        this.modele = modele;
        this.etat = false;
        this.x = x; this.y = y;
    }
  /** Le passage àla génération suivante
    se fait en deux étapes :
    1- D'abord on calcule pour chaque cellule ce que sera
     son état àla génération suivante (méthode [evalue]).
      On stocke le résultat dans un attribut supplémentaire
      [prochainEtat].
     2- Ensuite on met àjour l'ensemble des cellules
        (méthode [evolue]).
       Objectif : éviter qu'une évolution immédiate d'une
      cellule pollue la décision prise
      pour une cellule voisine.
      */
    private boolean prochainEtat;
    protected void evalue() {
        switch (this.modele.compteVoisines(x, y)) {
        case 2: prochainEtat=etat; break;
        case 3: prochainEtat=true; break;
        default: prochainEtat=false;
        }
    }
    protected void evolue() {
        etat = prochainEtat;
    }

    /** Un test àl'usage des autres classes
    (sera utilisé par la vue). */
    public boolean estVivante() {
        return etat;
    }
}
```

La vue : l'interface avec l'utilisateur. On définit une classe chapeau [CVue] qui crée la fenêtre principale de l'application et contient les deux parties de notre vue :

- Une zone d'affichage où on voit l'ensemble des cellules.
- Une zone de commande avec un bouton pour passer à la génération suivante.

On devra préciser un mode pour disposer ces deux zones à l'intérieur de la fenêtre. Quelques possibilités sont :

- BorderLayout (défaut pour la classe JFrame) : chaque élément est disposé au centre ou le long d'un bord.
- FlowLayout (défaut pour un JPanel) : les éléments sont disposés l'un à la suite de l'autre, dans l'ordre de leur ajout, les lignes se formant de gauche à droite et de haut en bas. Un élément peut passer à la ligne lorsque l'on redimensionne la fenêtre.
- GridLayout : les éléments sont disposés l'un à la suite de l'autre sur une grille avec un nombre de lignes et un nombre de colonnes définis par le programmeur, dont toutes les cases ont la même dimension. Cette dimension est calculée en fonction du nombre de cases à placer et de la dimension du contenant. Ce mode etait celui utilisé pour le démineur et pour le problème des Nreines.

```
class CVue {
 /** JFrame est une classe fournie pas Swing.
  Elle représente la fenêtre de l'application graphique. */
    private JFrame frame;
    /** VueGrille et VueCommandes sont deux classes
      définies plus loin, pour nos deux
```

```
    parties de l'interface graphique. */
    private VueGrille grille;
    private VueCommandes commandes;

    /** Construction d'une vue attachée àun modèle. */
    public CVue(CModele modele) {
        /** Définition de la fenêtre principale. */
        frame = new JFrame();
        frame.setTitle("Jeu␣de␣la␣vie␣de␣Conway");
        /** On précise le mode "Flow"pour disposer les
         différents éléments àl'intérieur de la fenêtre. */
        frame.setLayout(new FlowLayout());
        /** Définition des deux vues et ajout àla fenêtre. */
        grille = new VueGrille(modele);
        frame.add(grille);
        commandes = new VueCommandes(modele);
        frame.add(commandes);
        /**
         * Remarque : on peut passer àla méthode [add] des
             paramètres
         * supplémentaires indiquant où placer l'élément. Par
             exemple, si on
         * avait conservé la disposition par défaut [
             BorderLayout], on aurait
         * pu écrire le code suivant pour placer la grille à
             gauche et les
         * commandes àdroite.
         * frame.add(grille, BorderLayout.WEST);
         * frame.add(commandes, BorderLayout.EAST);
         */

        /** On ajuste de la taille de la fenêtre en fonction
           du contenu. On indique qu'on quitte l'application
            si la fenêtre est fermée. On préciser que la fenêtre
             doit bien apparaître àl'écran.*/
        frame.pack();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE
            );
        frame.setVisible(true);
    }
}
```

Une classe pour représenter la zone d'affichage des cellules. JPanel est une classe d'éléments graphiques, pouvant comme JFrame contenir d'autres éléments graphiques.

Cette vue va être un observateur du modèle et sera mise à jour à chaque nouvelle génération des cellules.

```
class VueGrille extends JPanel implements Observer {
    /** On maintient une référence vers le modèle. */
    private CModele modele;
    /** Définition d'une taille (en pixels) pour l'affichage
         des cellules. */
    private final static int TAILLE = 12;

    /** Constructeur. */
    public VueGrille(CModele modele) {
        this.modele = modele;
        /** On enregistre la vue [this] en tant qu'observateur
             de [modele]. */
        modele.addObserver(this);
        /** Définition et application d'une taille fixe
         pour cette zone de l'interface, calculée en fonction
         du nombre de cellules et de la taille d'affichage.*/
            Dimension dim = new Dimension(
                TAILLE*CModele.LARGEUR,
                TAILLE*CModele.HAUTEUR);
            this.setPreferredSize(dim);
    }

    /** L'interface [Observer] demande de fournir une méthode
     [update], qui sera appelée lorsque la vue sera notifiée
      d'un changement dans le modèle. Ici on se content de
      réafficher toute la grille avec la méthode prédéfinie
       [repaint]. */
    public void update() { repaint(); }

    /** Les éléments graphiques comme [JPanel] possèdent
     une méthode [paintComponent] qui définit l'action à
     accomplir pour afficher cet élément. On la redéfinit ici
       pour lui confier l'affichage des cellules.

       La classe [Graphics] regroupe les éléments de style
       sur le dessin, comme la couleur actuelle. */
    public void paintComponent(Graphics g) {
        super.repaint();
        /** Pour chaque cellule... */
        for(int i=1; i<=CModele.LARGEUR; i++) {
            for(int j=1; j<=CModele.HAUTEUR; j++) {
                /** ... Appeler une fonction d'affichage
                     auxiliaire.
             On lui fournit les informations de dessin [g] et
             les coordonnées du coin en haut àgauche. */
                paint(g, modele.getCellule(i, j),
                        (i-1)*TAILLE, (j-1)*TAILLE);
            } } }

    /** Fonction auxiliaire de dessin d'une cellule.
    Ici, la classe [Cellule] ne peut être désignée que
    par l'intermédiaire de la classe [CModele] àlaquelle
    elle est interne, d'où le type [CModele.Cellule].
    Ceci serait impossible si [Cellule] était déclarée
     privée dans [CModele]. */
    private void paint(Graphics g, Cellule c, int x, int y) {
        /** Sélection d'une couleur. */
        if (c.estVivante()) { g.setColor(Color.BLACK);
        } else { g.setColor(Color.WHITE); }
        /** Coloration d'un rectangle. */
        g.fillRect(x, y, TAILLE, TAILLE); } }
```

Une classe pour représenter la zone contenant le bouton. Cette zone n'aura pas à être mise à jour et ne sera donc pas un observateur. En revanche, comme la zone précédente, celle-ci est un panneau [JPanel].

```
class VueCommandes extends JPanel {
    /** Pour que le bouton puisse transmettre ses ordres,
     on garde une référence au modèle. */
    private CModele modele;

    /** Constructeur. */
    public VueCommandes(CModele modele) {
        this.modele = modele;
        /** On crée un nouveau bouton, de classe [JButton], en
```

```
        précisant le texte qui doit l'étiqueter. Puis on ajoute
         ce bouton au panneau [this]. */
        JButton boutonAvance = new JButton(">");
        this.add(boutonAvance);
```

Le bouton, lorsqu'il est cliqué par l'utilisateur, produit un événement, de classe [ActionEvent]. On a ici une variante du schéma observateur/observé : un objet implémentant une interface [ActionListener] va s'inscrire pour "écouter" les événements produits par le bouton, et recevoir automatiquements des notifications. D'autres variantes d'auditeurs pour des événements particuliers : [MouseListener], [KeyboardListener], [WindowListener]. Cet observateur va enrichir notre schéma Modèle-Vue d'une couche intermédiaire Contrôleur, dont l'objectif est de récupérer les événements produits par la vue et de les traduire en instructions pour le modèle. Cette strate intermédiaire est potentiellement riche, et peut notamment traduire les mêmes événements de différentes façons en fonction d'un état de l'application. Ici nous avons un seul bouton réalisant une seule action, notre contrôleur sera donc particulièrement simple. Cela nécessite néanmoins la création d'une classe dédiée.

```
        Controleur ctrl  = new Controleur(modele);
        /** Enregistrement du contrôleur  comme auditeur du
             bouton. */
        boutonAvance.addActionListener(ctrl);  }    }
```

Classe pour notre contrôleur rudimentaire. Le contrôleur implémente l'interface [ActionListener] qui demande uniquement de fournir une méthode [actionPerformed] indiquant la réponse du contrôleur à la réception d'un événement.

```
class Controleur implements ActionListener {
    /** On garde un pointeur vers le modèle, car le
    contrôleur doit provoquer un appel de méthode du modèle.
     Remarque : comme cette classe est interne,  cette
     inscription  explicite  du modèle est inutile.  On
     pourrait se contenter de faire  directement référence
    au modèle enregistré  pour la  classe  englobante
     [VueCommandes]. */
    CModele modele;
    public Controleur(CModele modele)
          { this.modele = modele; }
    /** Action  effectuée  àréception  d'un événement :
    appeler la  méthode [avance]  du modèle. */
    public void actionPerformed(ActionEvent e)
        { modele.avance(); } }
```

Variante : une lambda-expression qui évite de créer une classe spécifique pour un contrôleur simplissime.

```
  JButton boutonAvance = new JButton(">");
        this.add(boutonAvance);
        boutonAvance.addActionListener(
          e -> { modele.avance(); });
```

9 Classes internes/anonymes, lambda expressions.

Dans ce cours, on finis les set de slides sur les contraintes/invariants, que on illustre par des petits exercices de diagramme objets. On particulier, on considére le cas des contraintes qui s'exprimer en qualifiant les associations. Ensuite On étudie la notion de classe internes sous tous ses déclinaisons possible : classes anonyme, lambda expression. Les classes internes seront utilisée dans le TPs, pour programmer des invariants qui caractérisent les listes doublement chainée. On explorera aussi quels sont ces invariants, en cours, si le temps le permets.

9.1 Classe Interne

Idée générale : une classe interne est une classe déclarée à l'intérieur d'une autre classe. Elle peut donc accéder aux membres de la classe externe.

9.1.1 Classe interne non statique

Interêt Une classe interne non statique peut accéder a tout les membres de l'objet qui l'a créée, y compris les membres privés. Les membres comprennent attribut et méthodes. En fait, le compilateur crée un membre supplémentaire dans la classe interne référençant l'objet qui l'a créé. Cette propriété permet de factoriser du code, il n'y a pas besoin de répeter ces attributs dans la classe interne.

Une telle classe interne peut-être déclarée de manière globale dans l'objet ; elle sera accessible par l'ensemble des méthodes de l'objet. Elle peut aussi être déclarée de manière locale à une méthode de l'objet. Elle sera alors accessible depuis cette seule méthode.

Exemple (Classe non statique globale) :

```
public class ClasseExterne
{   private int compteur = 0;
     class ClasseInterne
{   private int index = 0;
public ClasseInterne()
{  compteur++; } } }
```

Adressage dans la classe externe. Depuis la classe interne, dans le cas où plusieurs variables ou méthodes portent le même nom dans la classe interne et la classe externe, le pointeur this seul désigne l'instance de la classe interne, tandis que le pointeur this précédé du nom de la classe externe désigne l'instance de la classe externe.

```
public class ClasseExterne
{  private int compteur = 10;
    class ClasseInterne
{ private int compteur = 0;
   public void count() {
  this.compteur++; // -> 1
  ClasseExterne.this.compteur--; // -> 9
} } }
```

Visibilité de la classe interne depuis l'extérieur Si une classe interne est déclarée publique, ou par défaut si rien n'est précisé, elle est connue du monde extérieur sous un nom construit selon le schéma suivant : NomClasseExterne . NomClasseInterne De plus, toute classe extérieure peut alors aussi faire référence aux champs de la classe interne. Si la classe interne est déclarée privée, ceci n'est pas possible.

Création d'instances Une instance de la classe interne ne peut être construite que par l'intermédiaire d'une instance de la classe externe. Cela peut se faire de deux façons : - soit au sein d'une méthode d'instance de la classe externe (c'est-à-dire une méthode dans laquelle "this" existe et désigne une instance de la classe externe) ; - soit en faisant explicitement référence à une instance "extObj" de la classe externe, en remplaçant "new" par "extObj . new "

Ainsi, chaque instance de la classe interne est liée à une instance de la classe externe (celle qui a servi à sa création).

Exemple Declarer LinkedListIterator comme une sous classe de LinkedList permet d'adresser directement firstBlock dans le constructeur, sans avoir a le passer en paramètre. D'autre part, cette classe est encapsulée dans LinkedList ce qui est logique car son interêt est limité précisément à LinkedList.

```
    class LinkedListIterator
           implements Iterator<T> {
private Block currentBlock;
public LinkedListIterator() {
    this.currentBlock = firstBlock;
}
    ..
    }
```

Exemple 2 Dans le contexte des déclarations suivantes :

```
class Externe {
    private int e;
    class Interne {
        int i;
        void f() { ... }
    }
}
```

voici les accès qui sont possibles ou non depuis la méthode `f()` :

- `i` et `this.i` désignent l'attribut `i` de l'objet de classe `Interne`.
- `e` et `Externe.this.e` désignent l'attribut `e` de l'objet de classe `Externe` qui a créé l'objet interne duquel on appelle la méthode.
- `this.e` n'est pas admis, car la classe `Interne` elle-même ne possède pas de champ `e`.

9.1.2 Classe interne statique

C'est beaucoup moins utilisé. Une classe interne statique ne peut accéder qu'aux membres statiques de sa classe contenante, représentée par ClassExterne dans l'exemple suivant :

```
public class ClasseExterne{
private int compteur = 0;
private static String nom = "Exemple";

static class ClasseInterne
{  private int index = 0;
   public ClasseInterne() {
 System.out.println("Création d'un objet dans "+nom);
  // compteur ne peut être accedé
} } }
```

9.2 Lambda-expressions

9.2.1 Passer une méthode en paramètre

Imaginons que nous voulons écrire une méthode `forEach` attendant deux paramètres :

1. Une collection d'objets de type `T`, implémentant par exemple l'interface `Iterable<T>`.
2. Une méthode `void f(T t)` à appliquer à chaque élément de la collection.

Le deuxième paramètre pose problème : nous ne pouvons utiliser comme paramètre que des objets, et non des méthodes. Nous pouvons néanmoins nous en sortir en fournissant comme paramètre un objet possédant cette méthode `f`.

Ainsi, nous pouvons définir une interface `FContainer<T>` des classes possédant une méthode `f` de la bonne signature, et utiliser cette interface pour définir notre méthode `forEach`. Cette dernière prend en particulier un paramètre `fc` de type `FContainer<T>`, et appelle la méthode `f` qui y est encapsulée.

```
interface FContainer<T> {
    void f(T t);
}

public class Main {
    public static <T> void forEach(
         Iterable<T> c, FContainer<T> fc) {
        Iterator<T> it = c.iterator();
        while (it.hasNext()) {
            T elt = it.next();
            fc.f(elt);
        }
    }
}
```

Pour utiliser la méthode `forEach`, il faut encore faire deux choses :

1. Créer une classe implémentant `FContainer<T>` et définissant la méthode `f` voulue.
2. Créer un nouvel objet de cette classe, à passer en paramètre à `forEach`

```
class Printer implements FContainer<String> {
    void f(String s) { System.out.println(s); }
}

public class Main {
    public static void main(String[] args) {
```

```
        Iterable<String> c = ...
        FContainer<String> fc = new Printer();
        forEach(c, fc);
    }
}
```

Sans donner de nom à cette instance `fc`, on pourrait également écrire l'appel

```
        forEach(c, new Printer());
```

9.2.2 Classes anonymes

On peut simplifier légèrement le mécanisme précédent en recourant aux *classes anonymes* pour éviter de définir une classe à part `Printer`.

Lorsque l'on dispose déjà d'une interface ou d'une classe abstraite, comme notre `FContainer`, on peut définir une classe sans nom implémentant cette interface à l'occasion d'une instruction `new`. Cette classe, dite *anonyme*, n'est définie que pour l'objet qui est en train d'être créé. Techniquement, l'instruction `new` est appliquée au nom de l'interface (ce qu'on ne pourrait pas faire en temps normal, puisqu'il est impossible de construire un objet d'une interface ou d'une classe abstraite), et un bloc entre accolades est ajouté ensuite pour définir la classe concrète dont on crée un objet.

Dans notre exemple, on peut ainsi déclarer le contenu de la classe `Printer` au moment exact où l'on veut passer une méthode en paramètre à `forEach`. On utilise pour cela une instruction `new FContainer()` à laquelle on ajoute, entre accolades, la définition de la méthode `f`.

```
        forEach(c, new FContainer() {
           void f(String s) {
              System.out.println(s); } } );
```

Note : comme `FContainter` est une interface paramétrée par un type `T`, le compilateur Java émet un avertissement ici. On évite ceci en précisant que le paramètre doit ici être instancié par `String` :

```
  forEach(c, new FContainer<String>() { ... });
```

Exemple d'utilisation de classes anonymes Pour pouvoir appeler la méthode sort sur un arrayList, il faut passer une méthode compare. Pour cela on passe en fait un objet intanciant une classe laquelle doit implémenter l'interface Comparator. Puisque on à pas besoin de référencer cette classe plus tard, on utilise une classe anonyme.

```
ArrayList<String> gfg = new ArrayList<>();
gfg.add("toto"); gfg.add("tata"); gfg.add("tutu");
gfg.sort(new Comparator<String>() {
   @Override
   public int compare(String s, String t1) {
      return  t1.compareTo(s);}  });
 System.out.println(gfg); //affiche la liste triée.
```

9.2.3 Lambda-expressions

Les lambda expressions, sont des cas particulier de classe anonymes, lesquelles sont elles même des cas particuliers de classe internes. Les classes internes anonymes permettent de réduire légèrement les manipulations nécessaires pour simuler le passage de méthode en paramètre, mais demandent néanmoins d'écrire plusieurs lignes de texte autour de la seule information pertinente (le corps de la méthode que l'on veut passer en paramètre).

Les lambda-expressions, empruntées au monde de la programmation fonctionnelle, sont la nouvelle solution à la mode dans le monde objet pour se passer de toute cette bureaucratie et ne rien écrire d'autre que le corps de la méthode à passer en paramètre. Elles sont disponibles dans Java depuis la dernière version majeure (Java SE 8, mars 2014).

Mécanisme de base

Ce "nouveau" mécanisme permet d'écrire notre appel à `forEach` en une seule ligne, de la manière suivante :

```
        forEach(c,
          (String s) ->
             { System.out.println(s); } );
```

La syntaxe de base des lambda-expressions est la mise bout-à-bout de trois parties :

1. Un n-uplet de paramètres avec leurs types. Exemples `(String s)` ou `(int n, boolean b)`.
2. Le symbole flèche `->`.
3. Un bloc de code entre accolades `{ ... }`.

Dans certains cas, on peut encore alléger légèrement cette syntaxe :

— Il n'est généralement pas nécessaire de préciser les types des paramètres, que le compilateur Java déduit (ou *infère*) du contexte d'utilisation de la lambda-expression (on parle en général d'*inférence de type*, et dans le jargon de Java de *context typing*).

— Si le corps de la méthode consiste uniquement à renvoyer le résultat d'une certaine expression `expr`, c'est-à-dire si le corps a la forme `{ return expr; }`, alors on peut se contenter de donner l'expression `expr`.

En combinant ces deux simplifications, une lambda-expression prenant en paramètre un entier et renvoyant le carré de cet entier peut être réduite de

```
  (int i) -> { return (i * i); }
```

à simplement :

```
        i -> i * i
```

Type des lambda-expressions

Dans notre exemple, la signature de la méthode `forEach` n'a pas changé, et est toujours

```
  public static <T> void forEach(
     Iterable<T> c, FContainer<T> fc)
```

Ainsi, `(String s) -> { System.out.println(s); }` est une lambda-expression reconnue comme instanciant l'interface `FContainer<String>`, alors même qu'elle ne fait pas explicitement référence à la méthode `f` qui est pourtant attendue par cette interface, et utilisée dans le corps de la méthode `forEach`.

Principe en œuvre ici : à la manière d'un objet d'une classe interne anonyme, une lambda-expression n'a pas de type que l'on peut nommer [1]. En revanche, une lambda-expression peut implicitement instancier n'importe quelle interface dite *fonctionnelle*, c'est-à-dire n'importe quelle interface (ou classe abstraite) contenant *exactement une méthode abstraite*. La méthode anonyme décrite par la lambda-expression est alors interprétée comme défissant cette unique méthode abstraite, quel que soit son nom.

Voici quelques interfaces fonctionnelles fournies dans le paquet `java.util.function` :

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

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

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

Ces interfaces sont instanciables respectivement par : une lambda-expression prenant un paramètre de type `T` et ne renvoyant pas de résultat, une lambda-expression prenance un paramètre de type `T` et renvoyant un résultat de type `R`, et une lambda-expression prenant un paramètre de type `T` et renvoyant un booléen.

Un cas d'utilisation typique : traitement d'un flux de données

Java 8 introduit, en plus des lambda-expressions, une class `Stream` représentant des flux de données et munie d'un certain nombre de méthodes permettant d'appliquer un traitement aux données successives. Nous avons par exemple :

- Une méthode `filter`, qui prend en paramètre une instance de l'interface `Predicate`, et qui élimine les éléments du flux pour lesquelle sa méthode `test` renvoie `false`.
- Une méthode `map`, qui prend en paramètre une instance de l'interface `Function`, et qui construit un nouveau flux obtenu en appliquant sa méthode `apply` à chaque élément du flux d'origine.
- Une méthode `forEach`, qui prend en paramètre une instance de l'interface `Consumer`, et qui applique sa méthode `accept` successivement à chaque élément du flux.

Toutes ces méthodes peuvent prendre en argument une lambda-expression du bon type. Ainsi, en supposant que `numbers` est une collection de nombres flottants, nous pouvons réaliser l'action "afficher la racine carrée de chaque nombre positif de la collection `numbers`" avec les instructions suivantes (la méthode `stream()` fait le lien entre notre collection d'origine, par exemple de type `ArrayList`, et la classe `Stream`) :

```
numbers
    .stream()
    .filter (r -> r >= 0)
    .map    (r -> Math.sqrt(r))
    .forEach(r -> { System.out.println(r); }) ;
```

10 Liason dynamique et transtypage.

10.1 Vue d'ensemble

Transtypage Une classe fille (autrement appelée sous-classe) peut être considérée comme une *spécialisation* de sa classe mère (autrement appelée super-classe). En particulier un objet de la classe fille peut être vu comme appartenant à la classe mère, car l'objet de la classe fille possède tous les attributs et méthodes des objets de la classe mère. En revanche, un objet de la classe mère ne peut pas être vu comme appartenant à la classe fille, car il lui manque peut-être certains des attributs ou méthodes de la classe fille. En d'autres termes, l'ensemble des objets de la classe fille peut être considéré comme un sous-ensemble des objets de la classe mère.

```
abstract class Vehicule {
    abstract public void quiSuisJe();
}
class Voiture extends Vehicule {
    public void quiSuisJe()
       { System.out.println("Une voiture"); }
    public void queFaisJe()
      { System.out.println("je roule"); }
}
Voiture voiture  = new Voiture();
Vehicule vevo = new Voiture();
```

Point important : une fois qu'un objet de la classe `Voiture` est enregistré comme appartenant à la classe mère `Vehicule` (c'est ce qui se passe avec `vevo`), on ne peut plus accéder qu'aux attributs et méthodes qui étaient déjà dans `Vehicule`. En particulier, on peut accéder à la méthode `quiSuisJe()`, qui est abstraite mais quand même déclarée, et on ne peut pas accéder à la méthode `queFaisJe()`, qui n'existe que dans la classe `B`.

Appel	Affichage	Erreur compilation
`voiture.QueFaisje()`	`je roule`	
`voiture.QuiSuisJe()`	`Voiture`	
`vevo.queFaisJe()`		`queFaisJe()` pas défini
`vevo.quiSuisJe()`	`Voiture`	

Liaison dynamique Supposons à présent que la classe vehicule n'est plus abstraite

```
class Vehicule {
  public void quiSuisJe()
    { System.out.println("Un vehicule"); }
}
Vehicule vehicule  = new Vehicule();
```

1. Pour les curieux, définissez une variable `l` de type `FContainer` par une lambda-expression, et utilisez la réflexion (`l.getClass().getName()`) pour observer le nom de la classe générée à la volée par le compilateur Java pour la lambda-expression `l`. Essayez aussi avec des classes internes éventuellement anonymes.

Lorsqu'une méthode `quiSuisJe()` est redéfinie par une classe fille `Voiture`, le choix de la méthode à appliquer n'est pas fait statiquement sur le seul critère du type. En effet, par l'effet du transtypage évoqué ci-dessus, le type d'un objet peut apparaître différent de ce qu'il est en réalité.

Appel	Affichage	Erreur ?
`vehicule.quiSuisJe()`	`Vehicule`	
`voiture.quiSuitJe()`	`Voiture`	
`vevo.quiSuisJe()`	`Voiture`	

Dans cet exemple, nous avons d'abord deux objets `vehicule` et `voiture`, respectivement de classe `Vehicule et de classe \verb`voiture+, pour lesquels sont appelées respectivement les méthodes `quiSuisJe()` définies dans la classe `Vehicule` et redéfinie dans la classe `Voiture`.

Le cas de l'objet `vevo` est un peu plus compliqué : il a été créé avec l'instruction `new Voiture();` comme un objet de classe `Voiture` (il s'agit de son type *réel*), mais par la déclaration `Vehicule vevo` il a été transtypé en un objet de classe `Vehicule` (il s'agit de son type *apparent*). Le type apparent est celui qui est vu par le compilateur et par le vérifieur de type. En revanche, lors de l'appel d'une méthode qui existe à la fois dans la classe mère et dans la classe fille, c'est le type réel qui est consulté : dans notre exemple, même si l'objet `vevo` est *apparemment* dans la classe `Vehicule`, l'appel de méthode `vevo.quiSuisJe()` utilise la définition de `quiSuisJe()` fournie par la classe `Voiture` (la classe *réelle* de l'objet).

Cette information sur la classe réelle d'un objet ne peut pas être connue statiquement : si un tableau est rempli d'objets qui sont *apparemment* de la classe `Vehicule`, mais dont certains sont *en fait* de la classe `Voiture`, il n'est pas possible lorsqu'on regarde une case au hasard de savoir à l'avance quelle sera la classe réelle de l'objet ("à l'avance" signifiant : "au moment de la compilation"). Le choix définitif de la méthode à appliquer est donc fait *dynamiquement*, pendant l'exécution du programme. On appelle ce phénomène la *liaison dynamique*.

D'un point de vue pratique, pour chaque appel d'une méthode `quiSuisJe()`, le compilateur détermine *statiquement* une *famille* (c'est-à-dire un ensemble) de méthodes `quiSuisJe()` potentiellement applicables. Cette famille regroupe typiquement une définition de méthode `quiSuisJe()` d'une classe mère et toutes les redéfinitions faites dans les classes filles, ou petites-filles, ou descendantes plus lointaines. Ce choix est fait en fonction des types *apparents*, et règle en particulier les cas de surcharge de `quiSuisJe()`. Ensuite, lors de l'exécution du programme, une méthode `quiSuisJe()` est choisie *dynamiquement* parmi cette famille en fonction du type *réel* de l'objet.

Exemple d'utilisation de liaison dynamique. En reprenant l'exemple du démineur, nous pourrions imaginer avoir deux classes pour les cases du jeu : une classe principale, et une sous-classe spécifique pour les cases piégées. Cette sous-classe redéfinirait alors la méthode décrivant la réaction aux clics.

```
class Case {
  public void clicGauche() {/* Code OK */ }
}
class Piegee extends Case {
  public void clicGauche() {/* Code Perdu */}
}
```

Ensuite, toutes les cases, piégées ou non, sont stockées dans un même tableau. Ce tableau ne peut pas avoir le type `Piegee[][]`, car il ne contient pas que des cases piégées. Son type sera donc `Case[][]` et il pourra contenir à la fois des cases normales et des cases piégées, grâce au transtypage. Ainsi, si l'on considère une case `c` prise dans le tableau, cette case `c` a le type apparent `Case`. En revanche, son type réel peut être `Case` ou `Piegee`, et l'on veut qu'un clic sur une case de type réel `Piegee` exécute le code perdant et non le code gagnant.

10.2 Liaison dynamique en détail

On écrit toujours un appel de méthode de l'une des deux façons suivantes :

— `x.f(y)`, où `x` est l'objet dont on déclenche une méthode, `f` est le nom de la méthode appelée, et `y` est l'ensemble des paramètres fournis. On appelle les arguments `y` les *paramètres explicites* et `x` le *paramètre implicite*.

— `f(y)`. Cette deuxième forme est en réalité un abréviation pour `this.f(y)`, où `f` et `y` sont comme avant le nom de méthode et les paramètres explicites, et où le paramètre implicite est `this`, l'objet depuis lequel on invoque la méthode.

Prenons donc un appel de méthode de la forme `x.f(y)`, où le paramètre implicite `x` est éventuellement `this`, et notons `C` la classe réelle de l'objet `x`. La méthode à appeler est déterminée ainsi :

1. On construit la liste L de toutes les méthodes appelées `f` se trouvant dans l'une des deux situations suivantes :
   — définie dans la classe `C`, ou
   — définie dans une super-classe de `C` et qualifiée par `public` ou `protected`.
2. On élimine de la liste L toutes les méthodes dont la signature ne correspond pas aux paramètres explicites `y`. Incidemment, cette étape résout les éventuelles surcharges de notre méthode `f`.
3. Si on a obtenu une méthode `f(y)` associée à l'un des qualificatifs `private`, `static`, ou `final`, alors on invoque celle-ci.
4. Sinon, on cherche la définition de `f(y)` *la plus récente* : si on a une définition de `f(y)` dans `C`, alors on invoque celle-ci ; sinon, on regarde dans la super-classe de `C` ; si on ne trouve toujours pas, on remonte à la super-classe de la super-classe de `C`, etc.

```
class Vehicule {
 private a(int n)  {
    System.out.println("Vehicule"+n); }
 public  a(char c) {
    System.out.println("Vehicule"+c); }
}
class Avion extends Vehicule {
```

```
  public  a(int n)  {
    System.out.println("Avion"+n); }
}
class Voiture extends Vehicule {
 public  a(int n)  {
   System.out.println("Voiture"+n); }
 public  a()       {
   System.out.println("Voiture");   }
}
class Peugeot extends Voiture {
  public  a(int n)  {
    System.out.println("Peugeot"+n); }
}
class P208 extends Peugeot {
 private b(int n)  {
   System.out.println("Classe P208"+n);}
}
P208 p = new  P208();
```

On considère un appel `p208.a(3)`, où le paramètre implicite `p` est de la classe `P208` et où le paramètre explicite `3` est un unique argument de type `int`. Les étapes sont les suivantes :

1. La liste L contient les définitions `Vehicule.a(char)`, `Voiture.a(int)`, `Voiture.a()`, `Peugeot.a(int)`. La définition `Vehicule.a(int)` n'est pas retenue car elle est qualifiée par `private`, et la définition `Avion.a(int)` non plus car `Avion` n'est pas une super-classe de `P208`. La définition `P208.b(int)` est écartée car elle n'a pas le bon nom.
2. On élimine les définitions `Vehicule.a(char)` et `Voiture.a()` car elles ne correspondent pas au type de l'argument explicite `3` qui est entier.
3. Il n'y a pas dans L de méthode qualifiée par `private`, `static`, ou `final`, donc on passe à l'étape suivante.
4. Il n'y a pas de définition pour `a(int)` dans la classe `P208`, donc on regarde dans sa super-classe immédiate `Peugeot`. Il y a une définition pour `a(int)` dans `Peugeot` : c'est celle-ci qu'on invoque. La définition `Voiture.a(int)` est donc ignorée (elle a été redéfinie par `Peugeot.a(int)`).

Finalement, l'appel utilise la définition `D.a(int)`.

En pratique, on ne reproduit pas cette procédure dynamiquement à chaque fois qu'un appel de méthode est exécuté, cela prendrai trop de temps. Pour chaque classe, une *table de méthodes* est précalculée statiquement par le compilateur, qui indique directement les endroits où aller chercher les définitions des méthodes pour chaque combinaison `f(y)` acceptable. Pour exécuter un appel de méthode `x.f(y)`, on va donc consulter la table des méthodes de la classe *réelle* de `x`.

Dans notre exemple, les tables de méthodes des classes `Voiture` et `P208` sont les suivantes :

Classe `C`			Classe `E`		
`a()`	:	`Voiture.a()`	`a()`	:	`Voiture.a()`
`a(int)`	:	`Voiture.a(int)`	`a(int)`	:	`Peugeot.a(int)`
`a(char)`	:	`Vehicule.a(char)`	`a(char)`	:	`Vehicule.a(char)`
			`b(int)`	:	`P208.b(int)`

Si un appel ne correspond à aucune des lignes de la table de méthodes de la classe concernée, alors une erreur est renvoyée (normalement, cela se fait à la compilation).

10.3 Transtypage vers le bas

Lorsqu'une classe `B` est une sous-classe d'une classe `A`, on a déjà vu que tout objet de la classe `B` pouvait être utilisé comme un objet de la classe `A` (en d'autres termes, les objets de la classe `B` forment un sous-ensemble des objets de la classe `A`).

Ceci est une opération de *transtypage* "vers le haut", dans laquelle un objet `obj` d'une sous-classe `B` prend l'apparence d'un objet d'une super-classe `A`. L'objet `obj` a alors un type *apparent* `A` tandis que son type *réel* reste `B`. Le transtypage est une modification du type apparent.

Le transtypage est également possible dans l'autre sens, "vers le bas", via la notation `(B)obj`, mais seulement si le type réel de l'objet `obj` le permet : le type réel de l'objet `obj` doit être un sous-type du nouveau type apparent souhaité `B`.

```
public static void f(Voiture v) { v.queFaisJe(); }
Vehicule ve=new Vehicule();  Avion a=new Avion();
Voiture vo=new Voiture(); Peugeot pe=new Peugeot();
Vehicule vepe=new Peugeot();
```

Ici, les objets `ve`, `a`, `vo`, et `pe` ont chacun un type apparent égal à leur type réel `Vehicule`, `Avion`, `Voiture`, ou `Peugoet`. L'objet `vepe` en revanche a le type réel `Peugeot` et le type apparent `Vehicule`, ce qui est légitime car `Vehicule` est une super-classe de `Peugeot`. Une définition `Voiture vove = new Vehicule();` n'aurait en revanche pas été possible : le compilateur l'aurait interdite car `Voiture` n'est pas une super-classe de `Vehicule` (et le compilateur aurait eu raison, en particulier car les objets de la classe `Voiture` sont censés posséder une méthode `queFaisJe()`, que n'ont pas les objets de type réel `Vehicule`).

La méthode statique `f(v)` attend en paramètre explicite un objet `obj` de la classe `Voiture`. Le compilateur vérifie ceci sur la base du type apparent de `obj`. Voici différents cas d'appel qui seraient acceptés ou rejetés.

Appel	Affiche	Erreur de compilation
`f(ve)`		`ve` n'est pas de type `Voiture`
`f(a)`		`a` n'est pas de type `Voiture`
`f(vo)`	`je roule`	
`f(pe)`	`je roule`	
`f(vepe)`		`vepe` pas de type `Voiture`
`f((Voiture)vepe)`	`je roule`	
`f((Peugoet)vepe)`	`je roule`	
`f((Voiture)a)`		`transtypage incompatible`

Appel	Erreur d'Execution
`f((Voiture)ve)`	transtypage impossible
`f((Voiture)(Vehicule)a)`	transtypage impossible

Le compilateur rejette les trois cas où le type apparent de l'argument n'est pas un sous-type du type `Voiture` attendu (objets `ve` et `vepe` de type apparent `Vehicule`, et objet `a` de type apparent `Avion`), et accepte les autres appels où l'argument a le type apparent `Voiture` ou `Peugeot` (éventuellement grâce à une opération de transtypage `(Voiture)vepe` ou `(Peugetot)vepe`).

Une deuxième sorte d'erreur de compilation apparaît au niveau du transtypage `(Voiture)a`, car le compilateur détecte que `Avion` et `Voiture` sont dans des branches différentes, et qu'il n'est pas possible de passer de l'une à l'autre. La combinaison `(Voiture)(Vehicule)a` en revanche, même si elle a le même effet, n'est pas bloquée par le compilateur, puisque ce dernier détecte d'abord un transtypage de `Avion` vers `Vehicule` (toujours possible) puis un transtypage de `\vehicule` vers `Voiture` (parfois possible, à déterminer à l'exécution en fonction du type réel).

Enfin, dans le cas d'un transtypage vers un sous-type, le compilateur accepte le code écrit, et laisse à la machine virtuelle la responsabilité de vérifier dynamiquement (à l'exécution) que le type réel de l'objet est compatible avec le transtypage. Les cas `(Voiture)vepe` et `(Peugeot)vepe` s'exécutent correctement car `vepe` est de type réel `Peugoet`. Les cas `(Voiture)ve` et `(Voiture)(Vehicule a` provoquent une erreur à l'exécution, car les types réels `Vehicule` et `Avion` ne permettent pas un transtypage vers `Voiture`.

Pour éviter ce genre d'erreur de transtypage à l'exécution, on utilise en général un test avec `instanceof` permettant de vérifier que le transtypage est possible :

```
if (obj instanceof Voiture) {
    f((Voiture)obj);
} else { ... }
```

11 Patrons de conceptions,

Ce cours utilise en partie un jeu de slides la programmation structurée, dont les commentaires ne sont pas dispo. Ce sont la avec les patrons de conceptions, des pratiques qui deviennent nécessaire lorsque l'on réalise de gros programmes. Ce cours inclue aussi une démo du debugger en cours.

Un patron de conception (souvent appelé design pattern) est un arrangement caractéristique de modules, reconnu comme bonne pratique en réponse à un problème de conception d'un logiciel. Il décrit une solution standard, utilisable dans la conception de différents logiciels.

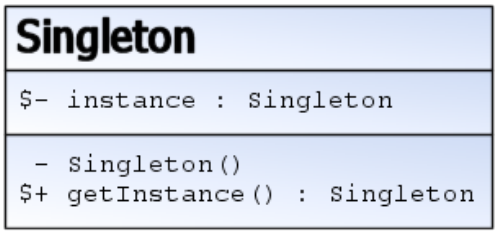
11.1 Patrons de creation

Un patron de création résouds des problèmes liés à la création et la configuration d'objets.

11.1.1 Singleton

Le singleton est un patron de conception dont l'objet est de restreindre l'instanciation d'une classe à un seul objet (ou bien à quelques objets seulement). Il est utilisé pour son efficacité, car il permet d'économiser de la mémoire.

On implémente le singleton en écrivant une classe contenant une méthode qui crée une instance uniquement s'il n'en existe pas encore. Sinon elle renvoie une référence vers l'objet qui existe déjà. Le constructeur est privé, afin de s'assurer que la classe ne puisse être instanciée autrement que par la méthode de création contrôlée.



11.1.2 Factory

```
public class Complex{
   public static Complex fromCartesian(double real, double
       imag)
    { return new Complex(real, imag); }
    public static Complex fromPolar(double rho, double theta)
    {return new Complex(rho * cos(theta), rho * sin(theta));   }
    private Complex(double a, double b) { /*...*/      }
}

Complex c = Complex.fromPolar(1, pi); //Identique à
     FromCartesian(-1, 0)
```

Le constructeur de la classe est ici privé, ce qui oblige à utiliser les méthodes de fabrication qui ne prêtent pas à confusion.

"Factory"-methode Les nombres complexes sont construit en général, en passant partie réelle et imaginaire. On pourrait vouloir écrire un deuxième constructeur basé sur le module et l'argument.

```
public Complexe(double r, double a) {this.rho  = r; this.theta
            = a; re=..  im=..}
```

Cependant, ce n'est pas possible : le mécanisme de surcharge est basé sur le type des paramètres, qui sont identiques pour nos deux constructeurs.

Le design pattern "factory" permet de donner plusieurs manières de construire un objet, en les distinguant de manière plus explicite que par le seul mécanisme de surcharge.Le constructeur est indiqué "privé" : il n'est pas directement accessible de l'extérieur. Pour construire des complexes, on fournit deux méthodes statique aux noms évocateurs, qui appellent le constructeur et renvoie l'objet construit.

```
    private Complexe(double r, double i) {
        this.re  = r;this.im  = i;  }

    public static Complexe complexeCartesien(double r, double i
        ) {
        return new Complexe(r, i);  }

    public static Complexe complexePolaire(double r, double a)
        {
        return new Complexe(r*cos(a),r*sin(a));
    }
```

En résumé, une méthode "factory" est une méthode statique qui retourne une instance d'une classe. Utiliser une factory plutot que un constructeur présente les avantages suivants :

1. Il n'y a pas besoin d'écrire new

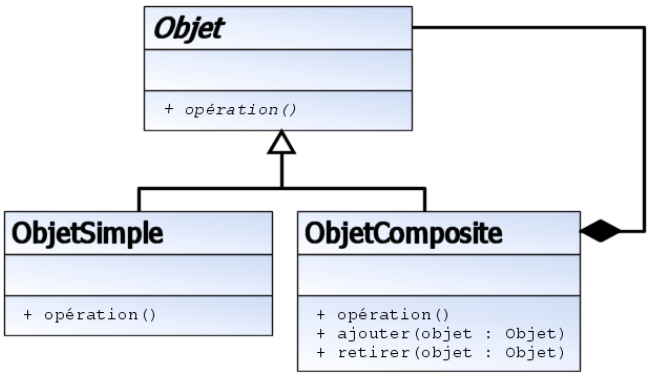
2. Une factory a un nom qui n'est pas le même que celui de la classe
3. Si on a deux constructeurs qui prennent les même paramètres, on pourra pas utiliser la surcharge pour les selectionner, les factory permettent de résoudre ce problème

11.2 Patron de structure

Un patron de structure résouds des problèmes liés à la structuration des classes et leur interface en particulier.

11.2.1 Objet Composite

Dans ce patron de conception, un objet composite est constitué d'un ou de plusieurs objets similaires (ayant des fonctionnalités similaires). L'idée est de manipuler un groupe d'objets de la même façon que s'il s'agissait d'un seul objet. Les objets ainsi regroupés doivent posséder des opérations communes, c'est-à-dire un "dénominateur commun".



La classe Objet déclare l'interface pour la composition d'objets et met en œuvre le comportement par défaut La classe ObjetSimple représente les objets manipulés, ayant une interface commune. La classe ObjetComposite définit un comportement pour les composants ayant des enfants stocke les composants enfants met en œuvre la gestion des composants enfants La classe utilisatrice manipule les objets de la composition à travers l'interface Objet.

Exemple : la classe container utilisée au TP visiteurs, tout a la fin, pour représenté des dessins pouvant etres soit des objets, soit des groupes d'objets. L'opération ici, consiste simplement à accepter un visiteur.

```
class Container extends Element {
    private ArrayList<Element> elements;

    public Container(String c) {
        super(c);
        this.elements = new ArrayList<Element>();
    }
    public void addElement(Element e) {elements.add(e);}

    @Override
    public void print() { }
    @Override
    public void accept(Visitor v) {
        v.visitContainerIn(this);
        for(Element e : elements) { e.accept(v); }
        v.visitContainerOut(this);
    }
}
```

11.3 Patron de comportements

Un patron de comportement permet de résoudre les problèmes liés aux comportements, à l'interaction entre les classes.

11.3.1 Patron Iterateurs

L'itérateur est un patron de conception comportemental qui vous permet de parcourir les éléments d'une collection sans exposer sa représentation sous-jacente (liste, pile, arbre, etc.).

11.3.2 Patron Modéle Vue Controleur

Le modèle MVC (Model View Controller) utilisé pour votre projet, spécifie qu'une application doit être constituée d'un modèle de données, d'informations de présentation et d'informations de contrôle. Le modèle exige que chacun de ces éléments soit séparé en différentes classes. Il permet de bien séparé tout ce qui est graphique du reste.

11.3.3 Visiteurs

Ce patron est utilisé dans le prochain TP. Il compléte le patron Objet Composite. Le visiteur est une manière de séparer un algorithme d'une structure de données. Un visiteur possède une méthode par type d'objet traité. Pour ajouter un nouveau traitement, il suffit de créer une nouvelle classe dérivée de la classe Visiteur. On n'a donc pas besoin de modifier la structure des objets traités, contrairement à ce qu'il aurait été obligatoire de faire si on avait implémenté les traitements comme des méthodes de ces objets.

L'avantage du patron visiteur est qu'un visiteur peut avoir un état. Ce qui signifie que le traitement d'un type d'objet peut différer en fonction de traitements précédents. Par exemple, un visiteur affichant une structure arborescente peut présenter les nœuds de l'arbre de manière lisible en utilisant une indentation dont le niveau est stocké comme valeur d'état du visiteur.

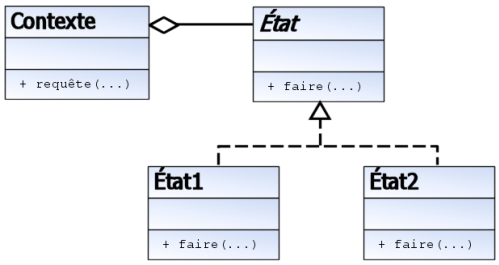
```
interface CarElementVisitor
{ void visit(Wheel wheel);
    void visit(Engine engine);
    void visit(Body body);
    void visitCar(Car car);
}
interface CarElement
{ void accept(CarElementVisitor visitor);
  // Méthode àdéfinir par les classes implémentant
        CarElements
}
class Wheel implements CarElement
{ private String name;
 Wheel(String name) { this.name = name;}
```

```
 String getName() {   return this.name;}
 public void accept(CarElementVisitor visitor)
     {visitor.visit(this);       }
}
class Engine implements CarElement
{ public  void accept(CarElementVisitor visitor)
     { visitor.visit(this);       }}
class Body implements CarElement
{ public void accept(CarElementVisitor visitor)
     { visitor.visit(this);        }}
class Car
{ CarElement[] elements;
     public CarElement[] getElements() { return elements.clone()
          ;  }
    // Retourne une copie du tableau de références.
     public Car() { this.elements = new CarElement[] {
      new Wheel("front␣left"),  new Wheel("front␣right"),
      new Wheel("back␣left"),new Wheel("back␣right"),
      new Body(),new Engine() };
     }}
class CarElementPrintVisitor implements CarElementVisitor
{public void visit(Wheel wheel)
     {System.out.println("Visiting␣"+   wheel.getName() + "␣wheel
          ");}

     public void visit(Engine engine)
     {System.out.println("Visiting␣engine");}

     public void visit(Body body)
     { System.out.println("Visiting␣body");     }

     public void visitCar(Car car)
     {System.out.println("\nVisiting␣car");
          for(CarElement element : car.getElements())
          {element.accept(this);  }
          System.out.println("Visited␣car");
     }}

class CarElementDoVisitor implements CarElementVisitor
{public void visit(Wheel wheel)
     { System.out.println("Kicking␣my␣"+ heel.getName()); }
 public void visit(Engine engine)
     {System.out.println("Starting␣my␣engine");   }
 public void visit(Body body)
       {System.out.println("Moving␣my␣body"); }
 public void visitCar(Car car) {
     System.out.println("\nStarting␣my␣car");
     for(CarElement carElement : car.getElements())
          { carElement.accept(this);  }
          System.out.println("Started␣car");
     }}

public class VisitorDemo
{static public void main(String[] args)
     {Car car = new Car();
          CarElementVisitor printVisitor   = new
               CarElementPrintVisitor();
          CarElementVisitor doVisitor  = new CarElementDoVisitor
               ();
          printVisitor.visitCar(car);
          doVisitor.visitCar(car);
     }
}
```

11.3.4 Etats

Le patron de conception "état" est utilisé lorsqu'il est souhaité pouvoir changer le comportement de l'État d'un objet sans pour autant en changer l'instance.

Etant donnée une classe dont les instances peuvent évoluer entre plusieurs états, le design pattern « state » sépare les parties communes à tous les états des parties variant d'un état à l'autre. Les parties variables sont déléguées à un objet de classe State, cette classe possédant une sous-classe concrète par état à implémenter.



Exemple : En 2019, ce patron avait était étudié en cours. Pendant l'examen 2019, il fallait comprendre le code suivant, qui permet de représenter un nombre binaire arbitrairement grand par sa suite de bits, et le l'incrementer. La méthode incr a un comportement different pour le bit 1 de poids fort, les autres bit 1, et les bits O. Par exemple, si la retenue se propage jusqu'au premier bit de poids fort, il faudra rajouter un nouveau bit.

```
class BinaryDigit {
private BinaryDigit next;
private State s;
public BinaryDigit() { this.s = new State1Final(); }
private String getValue() { return this.s.value;   }
public String toString() {
  if (this.next == null)
     return this.getValue();
  else return next.toString() + this.getValue();
}
public void incr() { this.s.incr();    }
private void requireNext() { this.next = new BinaryDigit(); }
private abstract class State {
  final String value;
  State(String d) { this.value = d; }
  abstract void incr();
}
private class State0 extends State {
  State0() { super("0");  }
  void incr() { s = new State1NonFinal(); }
}
private class State1NonFinal extends State {
  State1NonFinal() { super("1");  }
  void incr() {s = new State0();next.incr();}
}
private class State1Final extends State {
  State1Final() { super("1");  }
  void incr() {s = new State0();requireNext();}}}
```