Java : Introduction
Master PISE

Pierre Charbit, Jean-Baptiste Yunès

22 mars 2021

Table des matières

1 Variables et Types **9**
1.1 Identificateurs . . . 9
1.2 Typage . . . 9
1.2.1 Types primitifs 1 : entiers signés . . . 9
1.2.2 Types primitifs flottants . . . 10
1.2.3 Type primitif caractères . . . 10
1.2.4 Type primitif booléens . . . 10
1.2.5 Ordres des types primitifs . . . 10
1.2.6 Quelques abréviations à connaître . . . 11
1.3 Variables . . . 11
1.4 Expressions . . . 11
1.5 Structure simple d'un programme . . . 11

2 Instructions de Controle **13**
2.1 **`if`** . . . 13
2.2 **`for`** . . . 13
2.3 **`while`** et **`do-while`** . . . 14
2.4 **`switch`** . . . 14

3 Tableaux et chaines de caractères **15**
3.1 Généralités . . . 15
3.2 Tableaux à plusieurs dimensions . . . 16
3.3 La Classe String . . . 16

4 Méthodes (Fonctions) **17**
4.1 Intro . . . 17
4.2 Passage des arguments . . . 18

5 Entrées Sorties **21**
5.1 Entrées/Sorties basiques . . . 21
5.2 Écrire/lire dans un fichier . . . 22

6 Classes et Objets **25**
6.1 Un exemple, la classe Identité . . . 26
6.2 Créer un nouvel objet . . . 26
6.3 Objets et Références . . . 27
6.4 Exercices . . . 27
6.5 Initialisation des objets . . . 28
6.6 Constructeurs . . . 29
6.7 Méthodes . . . 30
6.8 La méthode **`toString`** . . . 31
6.9 Les statiques . . . 32

6.9.1 Les champs statiques 32
6.9.2 Les méthodes statiques 32

7 Héritage 35
7.1 Exemple 35
7.2 Principe 37
7.3 Héritage des Constructeurs 37
7.4 Héritage des Méthodes et redéfinition 38
7.5 Héritage - Polymorphisme 38

8 Collections et boucles «for each» 41
8.1 ArrayList 41
8.2 Boucles «for each» 41

9 Classes Object, Class 43
9.1 Classe Object 43
9.2 Classe Class 43
9.3 Operateur instanceof 44

10 Javadoc 45
10.1 Javadoc 45

11 Interfaces 47
11.1 Intro 47
11.2 Exemple : Comparable 47
11.3 Héritage multiple ? 48
11.4 Remarques 48

12 Entrées Sorties 49
12.1 Sérialisation 49

13 Java et Base de Données 51
13.1 Intro 51
13.2 Se connecter à la base. 51
13.3 Lire dans la base : **Statement** et **ResultSet** 52
13.4 Lire dans la base : **Statement** et **ResultSet** 52
13.5 Lire dans la base : **ResultSetMetaData** 53
13.6 Résumé 53
13.7 Ecrire dans une base 53
13.8 Écrire dans une base - Requêtes Préparées 54

14 Interfaces graphiques à la main 55
14.1 Introduction 55
14.2 Un aperçu rapide des composants Swing 55
14.3 Le composant conteneur le plus simple : **JFrame** 56
14.4 Quelques exemples 56
14.5 Remplir un **JFrame** 57
14.6 Agencement et LayoutManager 57
14.6.1 Principe 57
14.6.2 **FlowLayout** 58
14.6.3 **BorderLayout** 58
14.6.4 **BoxLayout** 58
14.6.5 **GridLayout** 58
14.7 Gérer la taille des composants, commande **pack()** 59
14.8 La Barre de Menu **JMenuBar** 59

14.9 Ça ferme (pas) vraiment ? 59
14.10 Evènements et Ecouteurs 60

Préliminaire

Ce que dit (disait) Pierre Charbit de ce document.

Ceci n'est pas un cours de java, mais une version un tout petit peu enrichie d'une compilation en format A4 d'une série de transparents projetés pendant le cours de java donné en master pise en 2012. Les «Chapitres»n'en sont pas vraiment et n'ont pas été pensés comme tels.

Ce que dit (disait) Jean-Baptiste Yunès de ce document.

Ceci n'est toujours pas un cours de Java, même si j'ai effectué diverses passes ici et là conduisant à y mettre un peu plus de progression pédagogique.

Des choses à consulter

Nous vous conseillons de consulter la documentation (très fournie) d'Oracle, la société qui pilote le projet Java ; même s'il en existe une version libre OpenJDK.
Si vous voulez consulter la documentation Java, la documentation de la version 6 suffit largement pour ce cours. Vous pouvez utiliser une version plus récente mais il faudra «oublier» certains concepts introduits tardivement dans Java... Oracle possède aussi de très utiles tutoriels donc consultez l'API et les tutoriels.

Chapitre 1

Variables et Types

1.1 Identificateurs

Pour désigner les éléments d'un programme, donc pour donner un nom à quelque chose dans un algorithme ou un programme (une variable, une fonction, ...), on utilise en Java un **identificateur**.
Un identificateur en Java est une lettre (le caractère _ est une lettre) suivie d'une suite de lettres ou de chiffres de longueur quelconque. La définition exacte de ce qu'est un identificateur Java est plus complexe (on peut utiliser certains symboles UNICODE) mais celle-ci est celle qui est très largement utilisée afin d'éviter les problèmes.
Exemples : **`x, y, x1, _var, CONST`**.

1.2 Typage

Un **type** correspond à la définition de :
- un ensemble de valeurs,
- un ensemble d'opérations applicables aux éléments de cet ensemble et la sémantique de ces opérations.

Toute variable a un type, toute expression a un type. Le typage impose des limites au mélange qu'on peut réaliser avec différentes valeurs.
En Java on distingue :
- les types **primitifs** (caractères, entiers, réels, booléens).
- les types **références** (qu'on appelle aussi **classes**) : les tableaux, les nombreuses classes fournies par les nombreux **paquetages** (package) Java (comme par exemple la classe **`String`** qui permet de représenter des chaînes de caractères), et les nouvelles classes définies par les utilisateurs.

1.2.1 Types primitifs 1 : entiers signés

Il existe 4 types différents selon la taille éventuelle des entiers concernés
- `byte` (1 octet),
- `short` (2 octets),
- `int` (4 octets),
- `long` (8 octets).

1 octet signifie 8 bits, et donc que un byte est compris entre $-2^7 = -128$ et $2^7 - 1 = +127$ (ce qui fait bien $2^8 = 256$ valeurs) alors qu'un `int` est compris entre -2^{31} et $+2^{31} - 1$.
On dispose des opérations suivantes :
- arithmétiques : **`+ - * /`** (division entière) **`%`** (reste)
- comparaison : égal (**`==`**), différent (`\! =`) plus grand (**`>`** ou **`>=`**), plus petit (**`<`** ou **`<=`**)

Remarques :
- par défaut l'écriture d'un entier est interprétée comme une valeur décimale de type **`int`**. Pour préciser que l'on veut un **`long`**, on fera suivre l'entier de l ou L. Exemples : **`231`**, **`243`**, **`2543L`**

— attention aux opérations arithmétiques, on compte modulo le taille de l'entier maximal dans le type utilisé. Par exemple pour les **int**, si $x = 2^{31}$, alors $x + 1 = -2^{31}$ et $2x = -1$.

Attention à l'écriture des nombres, il est possible d'utiliser d'autres bases que la base 10, notamment si le littéral commence par **0** c'est la base 8, **0x** pour la base 16 et **0b** pour la base 2. Le piège est la base 8... Il est recommandé aux débutants de ne jamais utiliser **0** pour débuter l'écriture d'un nombre !

1.2.2 Types primitifs flottants

- **float** (4 octet),
- **double** (8 octets).

Opérations :

- arithmétiques : **+ - * /**
- comparaison : égal (**==**), différent (**!=**) plus grand (**>** ou **>=**), plus petit (**<** ou **<=**)

Exemples : **123.345**, **0.12345** Par défaut un nombre à virgule est du type **double**, et du type **float** si on le fait suivre de **f** ou **F**.

1.2.3 Type primitif caractères

char (4 octets) : **'a'**, **'b'**, etc '
n' (saut de ligne) '
r' (retour chariot)
On ecrira par exemple

```
char c='a';
```

Attention le nom de la variable est **c**, mais contient le code du caractère correspondant à la lettre a. Il est très important de comprendre que les caractères ne sont en fait que des nombres... C'est pour faciliter les écritures que l'on autorise, par exemple, le litéral **'a'**, il ne s'agit en fait que du code UNICODE du caractère a de l'alphabet latin.
Les caractères sont codés sur 2 octets (il y a donc 2^{16} caractères différents) par des entiers et l'ordre alphabétique pour les lettres ordinaires est respecté.
Cela signifie que si **char c = 'c'**, alors **c+3** est le code de la lettre **f**.

1.2.4 Type primitif booléens

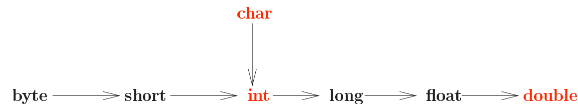
Un booléen représente une valeur de vérité vraie ou fausse.
Il ne peut prendre que deux valeurs possibles que sont les constantes **true** et **false**. Il ne s'agit pas de nombres, le type **boolean** est un type à part entière.
Toutes les expressions du type **x==2**, **x+y>= 24**, sont du type booléen.
Opérateurs :

- **!P** est la négation de **P**
- **&&** représente le ET
- **||** représente le OU
- ^ représente le OU disjonctif (c'est-à-dire **true^true** vaut **false**).

&& est paresseuse, c'est-à-dire qu'elle s'arrête des qu'elle rencontre **false** (pareil pour . **||** avec **true**).
Les opérateurs simples **&** et **|** permettent si besoin est de forcer l'évaluation du deuxième bloc.

1.2.5 Ordres des types primitifs

Cet ordre définit la compatibilité entre les types primitifs, en particulier les conversions implicites possibles. Ainsi, une valeur de type **int** peut être vue comme une valeur de type **long** mais pas le contraire. Cela permet d'additionner des nombres de types différents, ou de dire quelles opérations sont impossibles.



1.2.6 Quelques abréviations à connaître

- Au lieu d'écrire **x=x+3**, on peut écrire en Java **x+=3** et de même pour toutes les opérations arithmétiques.
- Dans le cas de **i=i+1** (resp **i=i-1**), on peut aussi écrire **i++** (resp. **i-**)

Ces écritures sont à connaître car elles sont très employées.

1.3 Variables

Une variable est un emplacement en mémoire pouvant contenir une valeur d'un type donné. Elle est désignée par un identificateur (son nom) valide (ne doit pas commencer par un chiffre). Toute variable doit être déclarée avant d'être utilisée. Elle a un type (voir section suivante), ce qui signifie qu'on spécifie quelles sont les valeurs qu'elle est autorisée à contenir.
La déclaration consiste à écrire par exemple **int x;**.
L'initialisation se fait par **x=2** si la variable déjà été déclarée.
On peut déclarer et initialiser en même temps en écrivant **int x =2**.
On peut déclarer plusieurs variables en même temps en séparant les identificateurs par des virgules : **int x,y;**
Attention, une variable n'existe que dans le bloc ou elle a été déclarée, un bloc étant l'ensemble des instructions comprises entre une accolade ouvrante l'accolade fermante associée.
Si on fait précéder la déclaration d'une variable du mot réservé **final**, on en fait une constante, c'est-à-dire qu'une fois la première affectation faite, on ne peut plus la modifier.
Si on essaie de compiler un programme contenant :

```
final int i=3;
i=4;
```

on va générer une erreur

1.4 Expressions

- une expression est composée de variables et/ou de constantes combinées avec des opérateurs.
- une expression a un type et une valeur qui dépendent des éléments qui la constituent. Pour un opérateur mélangeant des types compatibles entre eux, c'est celui de type le plus grand qui détermine le type de l'expression. Exemple : si **i** et **j** sont entiers, **i/j** est une division entière alors que si l'un des deux ne l'est pas la division est flottante.

  ```
  System.out.println(4/3);
  System.out.println(4/3.0);
  ```

  donne :

  ```
  1
  1.3333333333333333
  ```

1.5 Structure simple d'un programme

```
// importations (??) on verra plus tard
class Mon Prog{
   public static void main(String[] args){
    //liste de declaration de variables
    //liste dinstructions
    }
}
```

- le bloc **main** contient le programme proprement dit qui sera exécuté par `java MonProg`. Le sens des mots **public**, **static** et **void** sera expliqué plus tard.
- **String[] args** permet de passer des paramètres au programme

```
class Bonjour2{
   public static void main (String[] args){
      System.out.println("Bonjour " + args[0]);
   }
}
```

java Bonjour2 Pierre va renvoyer **Bonjour Pierre**

Chapitre 2

Instructions de Controle

2.1 if

```
if(condition){
...
}
else if(condition){
...
}
else{
...
}
```

condition est du type **boolean** (true ou false).
Exemple :

```
public class ex1 {
   public static int fonction(int i){
      if(i>0){return i;}
      else if(i==0) {return 12;}
      else{return -i; }
   }

   public static void main(String[] args){
      int p = fonction(3) + fonction(0) - fonction(-2);
      System.out.println(p);
   }
}
```

2.2 for

```
for (initialisation;test d'arret;incrementation){
..
}
```

test d'arret est du type **boolean** (true ou false).

Exercice 2.1 *Que produit l'instruction suivante ?*

```
      for(int i=1;i<4;i++){
         x*=2;
      }
      System.out.println(x);
   }
```

2.3 while et do-while

```
while (test d'arrêt) {
...
}
```

et sa variante :

```
do {
...
}
while (test d'arret)
```

Dans les deux cas, **test d'arrêt** est du type **boolean** (**true** ou **false**).

La différence entre les deux est que **do-while** exécute au moins une fois les instructions même si la condition d'arrêt n'est pas vérifiée

2.4 switch

Il arrive souvent qu'on écrive ce genre de code :

```
int x;
if(x==0) { ...}
else if(x==1) {..}
```

On peut alors écrire

```
switch (x) {
case 0 :
   // instructions
   break;
case 1:
   // instructions
   break;
default :
   // instructions
   break;
}
```

— cela ne fonctionne qu'avec des variables de type **byte**, **short**, **int**, **char**, **String**, ainsi que les types énumérés et les wrappers des types primitifs autorisés,
— mettre les **break** sinon l'exécution se poursuit dans les autres cas,
— **default** s'exécute si aucun cas ne correspond.

Chapitre 3

Tableaux et chaines de caractères

3.1 Généralités

Un tableau (*array*) est un regroupement ordonné d'objets (on dit une collection ordonnée), tous du même type et auxquels on accède par un indice entier. Si le tableau contient n éléments, le premier est d'indice 0, le second, d'indice 1,... , et le dernier d'indice $n-1$.
Toute classe ou type primitif peut être utilisé comme entrée d'un tableau :

```
MaClasse[] t;
int[] tableauDEntiers;
String[][] tableauDeTableauDeString;
```

Attention, un tableau est un objet et n'est donc manipulé qu'à travers une référence (même si ce qu'il contient est un type primitif). **`int[] t`** ne définit pas un tableau (même si on le dit fréquemment par abus de langage), **`t`** est une référence vers un tableau. Pour créer le tableau il faut le demander explicitement via, par exemple : **`new int[10]`**, qui créé un objet tableau de 10 **`int`**. Il faut bien entendu associer l'objet créé à la référence, l'idiome est :

```
int []t = new int[10];
```

La taille est fixée à l'initialisation est ne peut plus être changée par la suite. Elle est contenue dans le champ **`length`** donc on peut y accèder par **`tab.length`**.
Pour accéder à la case i d'un tableau on utilise **`tab[i]`**, où **`tab`** est une référence désignant un tableau.
Attention, ce n'est pas comme en C, si on tente d'accéder à une case d'indice négatif ou plus grand ou égal à la taille, Java renvoie à l'exécution une erreur :

```
int [] t = new int[10];
System.out.println(t[11]);
```

qui produit quelque chose comme :

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10 at bidule.main(bidu
```

signifiant que l'indice 10 n'est pas valide, en effet le tableau contient 10 éléments numérotés de 0 à 9.
Lorsqu'on créé un tableau, les éléments sont initialisés à des valeurs par défaut, pour les type primitifs c'est donc soit 0, soit **`false`**. Pour les types références c'est **`null`**.
On peut créer un tableau avec des valeurs spécifiées à l'aide de l'idiome suivant :

```
int [] t = { 111, 222, 333, 444, 555 };
```

qui créé un tableau de 5 entiers explicites.

3.2 Tableaux à plusieurs dimensions

Pour gérer des tableaux a plus d'une dimension en Java, il suffit de déclarer notre objet comme tableaux de tableaux.

```
int[][] t = new int[5][6]
```

designera ainsi un tableau a 5 lignes et 6 colonnes dont chaque case contiendra un entier.

t[0] designe ainsi la premiere ligne et est donc du type **int[]**, c'est un tableau d'entiers à une dimension de taille 6.
Pour savoir le nombre de lignes, on pourra alors ecrire

```
int l = t.length;
```

et pour le nombre de colonnes, demander la taille d'une quelconque des entrées de **t**

```
int c = t[0].length
```

La table des multiplications se construirait par exemple de la façon suivante.

```
int[][] t=new int[10][10]
for(int i=0;i<10;i++){
   for(int j=0;j<10;j++){
      t[i][j]=i*j;
      }
}
```

3.3 La Classe String

Cette classe fournie avec la librairie standard de Java permet de manipuler des chaînes de caractères.
Si on place une chaîne de caractères entre guillemets, c'est un **String** égal à cette chaîne de caractères.
L'opérateur **+** correspond à la concaténation. Ainsi

```
String s =''bonjour'';
s += "toto";
System.out.println(s);
```

affiche **bonjour toto**
Il existe une fonction qui permet de renvoyer le i-ème caractère d'un **String**. Attention les indices commencent à 0 comme pour les tableaux. Elle renvoie un **char** s'utilise de la façcon suivante :

```
String s = "bonjour 12";
char c = s.charAt(9);
```

Noter la syntaxe de cette méthode, on la reverra quand on abordera les types références.
Une autre fonction utile est celle qui permet d'obtenir la longueur d'une chaine de caractère (tous les caractères comptent pour un, y compris les espaces.)

```
String s = "bonjour toto";
int c = s.length();
System.out.println("s comporte "+ c + " caracteres");
```

Ici **c** vaut 12.

Il existe de nombreuses fonctions dans la classe String, reportez-vous à sa documentation.

Chapitre 4

Méthodes (Fonctions)

4.1 Intro

Souvent on veut écrire nos propres fonctions et leur donner un nom, de façon a pouvoir la réutiliser sans avoir à en réécrire le code.
En programmation objet, et donc en Java, on appelle en général une fonction une méthode.
La définition de nouvelles méthodes s'écrit dans le corps de la classe.
Exemple :

```
class Calcul{
   public static int puissance(int x, int n){
      int val=1;
      for(int i=1;i<=n;i++){
         val*=x;
      }
      return val;
   }
}
```

puissance est une méthode de la classe **Calcul**.
Attention ce n'est que la description d'une nouvelle fonction. Cela ne constitue pas un programme. Pour faire un programme informatique utilisant la fonction, on place comme toujours les instructions dans une méthode main.

```
class Calcul{
   public static int puissance(int x, int n){
      int val=1;
      for(int i=1;i<=n;i++){
         val*=x;
      }
      return val;
   }

   public static void main(String[] args){
      System.out.println("11*11=" + puissance(11,2));
   }
}
```

Pour appeler la fonction on a juste écrit **puissance**. Si une fonction est dans un fichier contenu dans le meme répertoire) on peut l'appeler avec son nom complet :**NomClasse.nomMethode** (par exemple ici **Calcul.puissance**).

La description d'une méthode se compose de :

1. Un en tête, ici **public static int puissance(int x, int n)**
   - **puissance** est le nom de la fonction
   - **int x** et **int n** sont les arguments. Il est obligatoire de préciser leur type.
   - juste avant le nom, figure le type de sortie de la fonction si elle renvoie quelquechose. Ici **int** signifie qu'elle renvoie un entier de type **int**.
   - **static** sera expliqué plus tard (mais il faudra toujours le mettre d'ici là).
   - Si une méthode ne renvoie rien (par exemple une méthode qui modifie un objet ou une méthode qui ne fait qu'afficher quelque chose à l'écran), son type de sortie est **void**.
2. Un corps de méthode composé d'une liste de déclarations de variables et d'une liste d'instructions.

   Le mot clé **return** précède ce qui va être renvoyé par la fonction en sortie, si elle en a une. Dans le cas où il y a plusieurs cas (par exemple à cause d'un **if .. else**), alors il y aura plusieurs fois le mot **return**.

Exemple :

```
public static int fonction(int i){
    if(i>0){return i;}
    else if(i==0) {return 12;}
    else{return -i; }
}
```

4.2 Passage des arguments

Pour expliquer le déroulement de l'évaluation d'une fonction, reprenons l'exemple de **puissance** :

```
class Calcul{
    public static int puissance(int x, int n){
        int val=1;
        for(int i=1;i<=n;i++){
            val*=x;
        }
        return val;
    }
}
```

Si on exécute **int res= 2*puissance(nombre,exposant)** :
- les variables **x** et **n** sont crées.
- ensuite tout se passe comme si on faisait **x=nombre** puis **n=exposant**, c'est à dire que les valeurs sont copiées dans x et n.
- les instructions de la méthode sont éxécutées, jusqu'à **return val**.
- la valeur de **val** est multipliée par 2 puis affectée à **res**

Exercice 4.1 *Que produit le programme suivant ?*

```
public class Exemple1{
    public static int fonction(int x){
        x=2*x;
        return x;
    }
```

```
    public static void main(String[] args){
        int x=2;
        int y=fonction(x);
        System.out.println("x vaut "+x+" et y vaut "+ y);
    }
}
```

Reponse **x vaut 2 et y vaut 4**

Il faudra faire attention à ce genre de questions quand les arguments seront de type référence et non plus de type primitif, comme par exemple avec des tableaux.

Exercice 4.2 *Que produit le programme suivant ?*

```
public class Exemple2{
    public static void fonction(int[] t){
        t[0]=18;
    }

    public static void fonction2(int[] t){
        t=new int[3];
        t[0]=99;
    }

    public static void main(String[] args){
        int[] tab = {1,3,5,7};
        System.out.print(tab[0] + ",");
        fonction(tab);
        System.out.println(tab[0]+ ",");
        fonction2(tab);
        System.out.println(tab[0]);
    }
}
```

Réponse

1,18,18

Chapitre 5

Entrées Sorties

Attention, lire ou écrire fait souvent intervenir des conversions entre représentations. C'est entre autres ce qui rend complexe leur manipulation. L'autre difficulté est que les entrées/sorties correspondent à des communications avec l'extérieur de l'application et que ces communications peuvent s'effectuer dans des modes différents. On insistera pas trop sur ces aspects ici, on se contentera de donner des recettes simples permettant de commencer.

5.1 Entrées/Sorties basiques

On a vu plusieurs exemples permettant d'afficher sur l'écran du terminal une chaine de caractères. Il s'agit de l'instruction **`System.out.println(String s)`**.
Si on remplace **`println`** par **`print`**, le texte est affiché sans retourner à la ligne après.
Nous n'avons pas encore parlé d'**`objets`** dans ce cours, mais ici on peut en voir certaines caractéristiques. Il existe en Java une classe nommée **`System`**, et cette classe a une propriété nommée **`out`**. Comme on l'a déjà vu pour les fonctions avec les tableaux ou les **`String`**s, pour aller chercher ce paramètre, on écrit le nom de la classe, suivi d'un point, suivi du nom de la propriété.
La valeur de cet objet **`System.out`**, appelée la sortie standard, est par défaut la fenêtre du terminal. Cet objet est du type Java **`PrintStream`**. Tous ces objets sont capables d'afficher des chaines de caractères, via la méthode **`print`** ou **`println`**.
Pour faire des lecture (entrées), c'est depuis la Java 6 assez simple (par rapport aux versions précédentes). On utilise pour cela un Objet appelé **`Scanner`** que l'on doit d'abord créer avec une instruction du type :

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

Les diverses méthodes de la classe **`Scanner`** permettent de lire les entrées. Par exemple, la méthode **`nextLine()`** permet de lire une ligne de saisie complète :

```
System.out.print("Quel est ton nom ? ");
String nom = clavier.nextLine();
```

Ici, nous utilisons la méthode **`nextLine()`** car la saisie pourrait contenir des espaces. Pour lire un seul mot (délimité par des espaces), on écrit :

```
String prenom = clavier.next();
```

Pour lire un entier, on utilise la méthode **`nextInt()`** :

```
int age = clavier.nextInt();
```

Attention, il existe quelques pièges avec ce type de lecture, notamment avec les fin de lignes qui ne sont pas consommées par les appels à **`nextInt`** par exemple. Si cette instruction est suivie d'une lecture de ligne alors la ligne lue est vide s'il la fin de ligne n'est pas consommée !

5.2 Écrire/lire dans un fichier

Écrire du texte dans un fichier

Il existe en Java de nombreuses classes permettant d'écrire dans un fichier. Pour du texte, il ne faut employer que les Reader/Writer. Les autres classes (InputStream/OutputStream) servent à lire des octets, et les lectures/écritures sont de plus bas niveau.
Pour le texte donc, une solution possible est proposée ci dessous :

```
try {
   FileWriter f = new FileWriter("monfichier.txt");
   f.write("Salut tout le monde");
   f.close();
} catch(Exception e){
   System.err.println(e);
}
```

On crée un objet **FileWriter** qui se charge d'écrire, on lui passe en argument du constructeur le nom du fichier dans lequel on veut écrire avec la méthode **write**. Il ne faut pas oublier de fermer le flot d'écriture via l'instruction **f.close()**.
Attention, le bloc try-catch est indispensable, car le constructeur et la méthode write sont susceptibles de renvoyer des erreurs. Il n'est pas nécessaire ici de comprendre comment fonctionne les exceptions. On se concentre sur les cas favorables (c'est-à-dire lorsque cela fonctionne bien).
On notera que les FileWriter sont assez basiques puisque ne permettent que d'écrire des **char** ou des **String**. Si l'on souhaite écrire d'autres choses comme par exemple une valeur du type primitif (comme par exemple avec **System.out**, il faut utiliser un **PrintWriter** :

```
try {
   PrintWriter f = new PrintWriter("monfichier.txt");
   f.println("Salut tout le monde");
   f.println(666);
   f.close();
} catch(Exception e){
   System.err.println(e);
}
```

Attention quand on dit «écrire une valeur de type entier», cela signifie qu'il y a un codage de cette valeur, la valeur est transformée en sa représentation sous la forme d'une chaîne de caractères (son écriture en base 10 avec les chiffres). Il ne faut pas confondre un nombre avec ses écritures.

Lire du texte dans un fichier

Pour lire du texte dans un fichier, on pourra utiliser la classe Scanner qu'on a déjà vue :

```
try {
   Scanner sc = new Scanner(new File("monfichier.txt"));
   while(sc.hasNext()) {
      System.out.println(sc.next());
   }
} catch(IOException e) {
   System.err.println("Erreur de lecture dans le fichier");
} catch (FileNotFoundException e) {
   System.err.println("Impossible de lire le fichier monfichier.txt");
}
```

La construction du **Scanner** s'effectue à l'aide d'un autre objet **File** qui permet de représenter à l'intérieur de l'application un fichier extérieur à celle-ci (ici sur le disque).
Le bloc try-catch est encore indispensable. Notez que l'on a choisi d'enrichir la capture d'exception, on peut deviner comment un filtrage est opéré.
Les fonctions classiques sont par exemple **hasNext()**, **next()**, **nextInt()**, **nextDouble()**, **nextLine()**.
La méthode **split()** de la classe **String** pourra s'avérer utile en combinaison avec **nextLine()**.

Chapitre 6

Classes et Objets

Un langage dit *orienté objet*, comme Java (ou C++, C#, Python...) est basé, comme son nom l'indique, sur le concept d'objet. Bien que le concept d'objet nous soit familier, il est difficile à définir (les philosophes s'y sont cassés les dents à plusieurs reprises). Un *objet* informatique est un élément qui représente une idée, un concept ou un object du monde réel, comme une personne, une voiture, un compte en banque, etc. Ce qui définit généralement un objet (ou ce qui permet de l'appréhender) sont ses attributs et moyens d'interagir avec lui. Par exemple, une voiture est caractérisée par son moteur, son nombre de portes, sa couleur, etc. et on peut la démarrer, la faire rouler, savoir à quelle vitesse elle va ou quelles portes sont ouvertes ou fermées.
L'idée de la programmation orientée objet (POO) est d'écrire les programmes comme des interactions entre objets. Afin que la structure du programme soit similaire à celle du problème que l'on cherche à «informatiser». Par exemple, si l'on desire programmer un jeu d'échecs, on pourra le décrire à l'aide d'objets joueurs, plateau, et pièces, ces dernières pouvant être catégorisées en tours, cavaliers, etc. Chaque pièce pouvant être caractérisée par sa couleur, sa position, et sa façon de se déplacer. Ensuite le jeu consistera en une suite d'actions réalisées à l'initiative des joueurs.
Un autre exemple, lorsque l'on réalisera des programmes avec interface graphique, on fera appel à des objets comme une fenêtre, un panneau, une image, un menu, et notre application sera un assemblage particulier de tous ces objets. La programmation est essentiellement une programmation par assemblage/composition.

Le concept central est celui d'objet mais il donc nécessaire de les décrire. Une *classe* est alors la description de l'ensemble des caractéristiques et des méthodes communes à un ensemble d'objets. Elle représente donc une catégorie d'objets, on aura donc la classe **`Voiture`**, la classe **`Fenetre`**, etc. Elle décrit aussi la façcon de fabriquer des objets de cette catégorie, via des fonctions particulières appelées *constructeurs*. Un objet de type **`Voiture`** sera dit *instance de la classe* **`Voiture`**.

Java fournit de nombreuses classes pouvant être utiles à la programmation, comme par exemple la classe **`String`**, que l'on a déjà vu et qui représente les chaînes de caractères. Une instance de la classe **`String`** est caractérisée par les caractères qui la composent et les méthodes dont on dispose dont les méthodes **`charAt()`**, **`length()`**...

Pour chacune de ces classes, il existe un fichier Java, qui fournit la description de ce qu'est un objet de ce type (quelles sont les valeurs qui y sont stockées) ainsi que les opérations valides que l'on peut effectuer.

Un des points forts des langages objets, comme Java, est de permettre à l'utilisateur de créer ses propres définitions d'objets, à la manière des **`struct`** en C mais en plus riche comme nous allons le voir. Le C ne décourage pas la programmation spaghetti, alors que la discipline Java si (même si tout est possible). Cela va notamment nous permettre de stocker dans un même objet plusieurs données de types différents, à la différence d'un tableau qui ne permet de stocker que des données du même type.

6.1 Un exemple, la classe Identité

Supposons que l'on veuille créer une classe pour représenter/manipuler des identités. Une Identité serait décrite par :
- le nom
- le prénom
- la date de naissance
- l'adresse

d'une personne.
Par conséquent on pourrait la définir en Java de la manière suivante :

```
class Identite {
   String nom;
   String prenom;
   int ddn;
   String adresse;
}
```

Les variables **nom**, **prenom**, **ddn**, et **adresse** qui décrivent le caractéristiques sont appelées *c*hamps.
Ces champs sont des variables qui peuvent être de n'importe quelle type ou classe, y compris celle que l'on est en train de définir.
Par exemple, supposons que l'on veuille gérer des comptes bancaires décrits par :
- un titulaire
- un solde
- un numéro de compte
- une caution (un autre compte qui se porte garant)

On pourra la définition de classe suivante :

```
class Compte {
   Identite titulaire;
   double solde;
   int numero;
   Compte caution;
}
```

6.2 Créer un nouvel objet

Les objets de Java ne se manipulent jamais directement, il vivent dans un monde particulier : le monde des objets Java. Pour manipuler un objet Java il est alors nécessaire de détenir une référence vers cet objet.
Pour créer une référence on écrira :

```
Identite pierre;
```

Pour l'instant on ne crée pas d'objet, on déclare juste une variable qui permettra (après association) de manipuler un objet.
Pour créer un objet, on fait appel à l'opérateur **new** puis par affectation de la valeur obtenue par l'opérateur **new** à la référence on obtient une référence qui désigne l'objet créé :

```
pierre = new Identite();
```

Pour accéder aux différents champs d'un objet, on doit utiliser la notation pointée qui à partir de la référence permet de désigner l'un des membres (ici des champs) contenus dans l'objet. On peut donc remplir les différents champs de la façon suivante.

```
pierre.nom="Dupont";
pierre.prenom="Pierre";
```

```
        pierre.adresse="78000 Versailles";
        pierre.ddn=1975;
    }
}
```

6.3 Objets et Références

Rappel : les variables de type objet (comme **pierre** dans l'exemple précédent), ne stockent pas elles mêmes les objets mais la référence vers l'objet (très grossièrement, le numéro de case mémoire dans laquelle l'objet est sauvé.)
Quelques règles importantes :

- une référence ne pointe que vers au plus un objet à un moment donné,
- plusieurs références peuvent, à un moment donné désigner le même objet.
- une même référence peut désigner plusieurs objets différents au cours du temps
- lorsqu'un objet n'est plus désigné par une référence, il est automatiquement effacé de la mémoire.

6.4 Exercices

Exercice 6.1 *Si l'on ajoute cette méthode* **main** *à la classe* **Identite** *et que l'on exécute, que va-t-on obtenir en sortie ?*

```
public static void main(String []args) {
    Identite idPierre, idJean;

    idPierre = new Identite();
    idJean = idPierre;
    idPierre.prenom = "Donald";
    idPierre.nom = "Trump";
    System.out.println("Prénom de Pierre = " + idPierre.prenom);
    System.out.println("Prénom de Jean = " + idJean.prenom);
}
```

Exercice 6.2 *Si l'on ajoute cette méthode* **main** *à la classe* **Identite** *et que l'on exécute, que va-t-on obtenir en sortie ?*

```
public static void main(String []args) {
    Identite idPierre, idJean;

    idPierre = new Identite();
    idJean = idPierre;
    idJean.prenom = "Donald";
    idJean.nom = "Trump";
    System.out.println("Prénom de Pierre = " + idPierre.prenom);
    System.out.println("Nom de Jean = " + idJean.nom);
}
```

Exercice 6.3 *Si l'on ajoute cette méthode* **main** *à la classe* **Identite** *et que l'on exécute, que va-t-on obtenir en sortie ?*

```
public static void main(String []args) {
   Identite idPierre, idJean;

   idPierre = new Identite();
   idPierre.prenom = "Donald";
   idPierre.nom = "Trump";
   idJean = idPierre;
   idJean = new Identite();
   idJean.prenom = "Manuel";
   idJean.nom = "Noriega";
   System.out.println("Prénom de Pierre = " + idPierre.prenom);
   System.out.println("Nom de Jean = " + idJean.nom);
}
```

Exercice 6.4 *Que penser des fonctions suivantes ?*

```
public static Identite copy0(Identite a){
   Identite b= new Identite();
   b.nom=a.nom;
   b.prenom=a.prenom;
   return b;
}

public static Identite copy1(Identite a){
   Identite b= a;
   return b;
}
```

```
public static void modif0(Identite a){
   a.nom="Gerard";
   a.prenom="Lambert";
}

public static void modif1(Identite a){
   a= new Identite();
   a.nom="Gerard";
   a.prenom="Lambert";
}
```

6.5 Initialisation des objets

L'initialisation des variables est un problème récurrent en informatique. En effet, la bonne initialisation des variables et l'une des propriétés essentielles à la bonne marche d'un programme. En effet, si les variables sont mal ou ne sont pas initialisées comme assurer un bon fonctionnement ? Pensez à ce qui se passe à l'ouverture d'un compte en banque si le dépôt n'est pas initialisé à 0 ! Constatez dans les exemples ci-dessus l'état des objets **Identite** pour lesquels certaines champs n'ont pas été initialisés ; on obtient des identités à moitié définies (pas de nom ou de prénom, etc).
Java permet d'obliger l'utilisateur d'un objet à fournir des valeurs au moment de la création d'un objet, de sorte que les objets soient toujours correctement construits/initialisés. Bien qu'il existe des règles régissant le comportement lorsqu'on initialise pas explicitement les champs d'un objet, il est fortement déconseillé de le faire.

6.6 Constructeurs

L'initialisation des champs passe (le plus souvent, mais on expliquera pas ici comme procéder autrement parfois) par la définition de constructeurs [1].
Un constructeur est une fonction très particulière :
- elle doit prendre le nom de la classe,
- elle ne produit pas de valeur de retour (rien, pas **void**, rien !),
- elle peut prendre autant de paramètre que souhaité,
- elle peut être surchargé autant de fois que nécessaire.

Ainsi :

```
class Identite {
   String nom;
   String prenom;
   Date ddn;
   public Identite(String n,String p,Date d) {
      this.nom = n; // initialisation du champ nom de l'objet (lui-même) avec le paramètre n
      this.prenom = p;
      this.date = d;
   }
}
```

Une telle définition oblige alors l'utilisateur de la classe à l'instancier en fournissant des valeurs, ainsi :

```
...
Identite i = new Identite("Obama","Barak",new Date());
...
```

La construction doit nécessairement prendre la forme d'un des constructeurs définis (en nombre de paramètres et en type). On notera l'utilisation du mot-clé **this** qui permet dans un objet de désigner l'objet lui-même (songez au mot «je» de la langue française qui permet à chacun de se désigner lui-même).
Note : dans les exemples précédents, nous n'avions pas défini de constructeur ce qui signifie que Java en ajoute un sans paramètre ce qui nous avait permis d'appeler **new Identite();**. Mais attention, une fois qu'un constructeur a été définit, ce constructeur sans paramètre n'existe plus, s'il y en a besoin il faut en construire un soi-même.

```
class Identite {
   String nom;
   String prenom;
   Date ddn;
   public Identite(String n,String p,Date d) {
      this.nom = n; // initialisation du champ nom de l'objet (lui-même) avec le paramètre n
      this.prenom = p;
      this.date = d;
   }
   public Identite() {
      this.nom = "Lennon";
      this.prenom = "John";
      this.date = new Date();
   }
}
```

Cette classe n'est donc instanciable que de deux façon différentes, soit avec trois paramètres, soit sans aucun paramètre...

1. mal nommés puisqu'ils servent en réalité à initialiser les champs, la constructions c'est autre chose

Attention, un constructeur n'est PAS une méthode, on ne peut l'invoquer autrement qu'indirectement en construisant un objet ;

6.7 Méthodes

On appelle méthode l'incarnation Java des opérations que l'on peut effectuer sur un objet. Par exemple, lorsqu'on écrit :

```
String s = "Salut la compagnie";
int l = s.length();
```

L'appel à **s.length()** consiste à invoquer la méthode **length()** sur l'objet désigné par **s**. On demande à l'objet de réaliser le service calculant sa longueur.
Ce genre de méthode d'instance s'applique donc toujours à un objet du type (la variable **s** dans l'exemple ci dessus), et peut avoir, entre parenthèses, d'autres arguments si nécessaire.

Supposons que dans une classe **Compte** (en banque), on veuille définir une opération **crediter(double montant)** qui rajouterait un certain montant sur le compte concerné et qui pourrait s'utiliser de la façon suivante : ß

```
Compte c = new Compte();
c.crediter(378.5);
```

La définition de la classe pourrait donc prendre la forme :

```
class Compte {
   double solde;
   public Compte() {
      this.solde = 0;
   }
   public void crediter(double montant) {
      this.solde += montant;
   }
}
```

Cette méthode s'applique à une instance particulière (l'objet désigné par **c** dans l'exemple précédent) et donc par définition son comportement dépend de cette instance. Son fonctionnement agit sur une instance particulière.
La définition d'une nouvelle méthode va se faire comme avant :

```
public type_de_sortie nom_de_la_fonction (arguments_typés) {
   //code avec des return si type_de_sortie n'est pas void
}
```

Un exemple plus complet et correct serait la définition suivante :

```
class Compte {
   private double solde;
   public Compte() {
      this.solde = 0;
   }
   public void crediter(double montant) {
      this.solde += montant;
   }
   public void debiter(double montant) {
```

```
      this.solde -= montant;
   }
   public double getSolde() {
      return solde;
}
```

Notez que dans cet exemple, on a rajouté deux services **debiter** (par symétrie avec **crediter**) mais aussi **getSolde()** (en partenariat avec le mot-clé **private** devant le champ **solde**).
Nous avons procédé ainsi car il est plus prudent, pour manipuler le solde d'un compte en banque de passer par des services bien définis. En effet, si nous avions laissé le champ **solde** sans protection (il y en a une mais pas assez forte), alors n'importe quel utilisateur se serait senti légitime pour modifier à sa guise le solde du compte (vous voyez le problème non ?). Même s'il pense le faire légitimement, le risque est grand d'introduire de la sorte une incohérence dans l'objet concerné (comment savoir s'il n'y avait pas d'autre opération à faire en lorsqu'on modifie le solde ? Par exemple calculer des agios, etc.). Le mot-clé **private** permet de limiter l'accès de la définition qui suit à l'intérieur de l'objet lui-même. Quelque chose de **private** est absolument invisible de l'extérieur et ne peut donc être utilisé directement depuis l'extérieur. Au contraire, quelque chose de **public** est visible depuis n'importe quel point du programme, cela fait partie de la facade de l'objet.
À travers cet exemple, on peut comprendre ce qui définit un objet : sa forte cohérence interne et son faible couplage parfaitement défini avec l'extérieur. Autrement dit, il peut régner à l'intérieur de l'objet un bazar innommable (enfin restons raisonnable tout de même) pourvu que l'on observe rien de particulier à l'extérieur. Pour vous faire une idée de la chose, encore une fois songez à votre compte en banque, en réalité qu'en savez-vous ? Rien du tout que ce qui est visible par les opérations qui sont possible dessus. Mais dans votre banque, quelle forme a ce compte ? Des écritures dans un livre de compte papier ? numérique ? sous quelle forme ? que fait la banque avec votre argent ? Rien de tout cela ne vous intéresse vraiment lors d'un usage quotidien. Que savez-vous du fonctionnement de votre voiture ? Probablement rien ou presque, vos notions de mécanique Newtonienne, mécanique des fluides, électronique, pilotage sont absentes voire très faibles. Bricoler une voiture n'est pas très facile (cohérence interne difficile à maintenir, les éléments dépendent fortement les uns des autres), mais vu de l'extérieur c'est si simple qu'à peu près n'importe qui peut conduire une voiture. C'est donc en ces termes qu'il faut songer aux objets : services externes simples et clairement définis, fonctionnement interne invisible quelqlue soit sa complexité.

6.8 La méthode `toString`

La méthode **System.out.print** (ou **println**) peut prendre en argument

- un type primitif : dans ce cas il écrit l'entier, le caractère, le booléen , etc.
- un objet quelconque : dans ce cas il fait appel à une méthode au nom réservé que l'on peut redéfinir (et c'est conseillé) dans chaque classe : **String toString()**

Le résultat de cette méthode doit être une chaine de caractères décrivant de façon lisible pour un humain le contenu de l'objet.

Si cette méthode n'est pas définie alors **System.out.print** affiche le nom de la classe auquel l'objet appartient suivi du symbole **@** suivi d'un entier décrivant l'objet pour java, appelé **hashcode**.
Ainsi avec la définition précédente de la classe **Compte** :

```
Compte c = new Compte();
System.out.println(c);
```

produit quelque chose comme :
`Compte@4dc63996`
Ce n'est pas très parlant, avouons-le, sinon qu'il s'agit d'un objet instance de classe **Compte** et d'identité **4dc63996**.
On peut donc redéfinir la méthode **toString** de la façon suivante :

```
public String toString() {
```

```
   return "C'est un compte ordinaire dont le solde est de "+getSolde()+"€";
}
```

ce qui produit cette fois :
`C'est un compte ordinaire dont le solde est de 0.0€`
Il est donc recommandé de redéfinir systématiquement cette méthode lorsqu'on définit une classe.

6.9 Les statiques

Certaines données ou fonctionnalités peuvent être plus globales qu'attachées aux instances. Par exemple, si l'on songe au nombre de d'instances d'une classe donnée ; si l'on souhaite connaître à tout instant le nombre de comptes en banque, ou le nombre de clients d'un magasin, etc. Ces nombres ne sont pas attachés aux instances (au compte, au client). C'est plutôt une donnée relative au concept lui-même. Si on raisonne par équivalence classe/instance vs ensemble/élément alors ce dont nous parlons est de propriétés ou d'attributs de l'ensemble. À près tout cela n'a rien de surprenant on peut fabriquer des ensembles d'ensembles et donc les ensembles peuvent être vus comme des objets et donc posséder des attributs !

6.9.1 Les champs statiques

La définition d'un attribut de classe, ou champ de classe s'effectue en le qualifiant à l'aide du mot-clé **`static`** :

```
class Compte {
   static int nombreTotalDeComptes;
}
```

qui définit un champ définit pour l'objet-classe **`Compte`**. Sa désignation s'effectue alors en utilisant la classe comme contexte :

```
System.out.println( Compte.nombreTotalDeComptes );
```

On peut alors envisager modifier le reste de la classe de sorte qu'elle permette à chaque instanciation le compte du nombre d'objet soit modifié :

```
class Compte {
   static int nombreTotalDeComptes;
   public Compte() {
      Compte.nombreTotalDeComptes++; // un compte de plus
   }
}
```

Puisqu'à chaque instanciation, le constructeur est appelé, l'attribut de classe reflète alors le nombre d'instances.
Cette façon de procéder n'est pas très raisonnable puisque l'attribut est visible/accessible et en particulier modifiable depuis l'extérieur de la classe.

6.9.2 Les méthodes statiques

Ces méthodes sont donc, comme les champs, non pas attachées aux instances mais à la classe elle-même. Elles définissent donc des fonctionnalités générales au concept que la classe représente. Comme par exemple des fonctions d'accès aux attributs statiques, des fonctions utilitaires, etc. Dans l'exemple précédent, si l'on rend l'attribut comptant le nombre d'instance caché alors il peut être naturel de définir des fonctions d'accès :

```
class Compte {
   private static int nombreTotalDeComptes;
   public static int getNombreDeComptes() { // permet de lire la valeur depuis l'extérieur du concept...
      return nombreTotalDeComptes;
```

```
    }

    public Compte() {
        Compte.nombreTotalDeComptes++; // un compte de plus
    }
}
```

Attention, puisque les méthodes statiques sont attachés à la classe elles ne peuvent utiliser le mot-clé **this** (qui, on le rappelle désigne dans une instance l'instance elle-même).

Chapitre 7

Héritage

7.1 Exemple

Supposons que l'on veuille faire un programme gérant un parc de location de véhicules. On aura besoin d'écrire une classe décrivant des voitures.

```
public class Voiture{
   Identite proprio;
   String modele;
   int annee;
   int nbPortes;
   int vitesseMax;
   boolean climatise;

}
```

On peut ensuite vouloir faire la meme chose pour les motos

```
public class Moto{
   Identite proprio;
   String modele;
   int annee;
   int vitesseMax;
   boolean coffre;
}
```

Ou les vélos

```
public class Velo{
   Identite proprio
   String modele;
   int annee;
   int nbVitesses;

}
```

On se rend compte qu'il y a beaucoup de choses en commun qui sont dues au fait que ces classes représentent toutes des véhicules. En java on peut écrire la chose suivante pour rassembler ces caractéristiques communes :

```
public class Vehicule{
   Indentite proprio
   String modele;
```

```
    int annee;
    int nbRoues;
}
```

Ensuite, on pourra alors utiliser la notion d'*héritage* pour écrire

```
public class Vehicule{
    Indentite proprio
    String modele;
    int annee;
    int nbRoues;
}

public class Voiture extends Vehicule{
    int nbPortes;
    int vitesseMax;
    boolean climatise;
}

public class Moto extends Vehicule{
    int vitesseMax;
    boolean coffre;
}

public class Velo extends Vehicule{
    int nbVitesses;
}
```

Ces classes disposeront des champs et des méthodes de la classe Vehicule, même si ceux-ci n'apparaissent pas dans leur corps.

Ici, on a un exemple avec une profondeur de 1 mais on pourrait par exemple envisager

```
public class Vehicule{
    Indentite proprio
    String modele;
    int annee;
    int nbRoues;
}

public class Velo extends Vehicule{
    int nbVitesses;
}

public class VehiculeMoteur extends Vehicule{
    int vitesseMax;
}

public class Voiture extends VehiculeMoteur{
    int nbPortes;
    boolean climatise;
}

public class Moto extends VehiculeMoteur{
    boolean coffre;
```

```
}
```

7.2 Principe

En Java, toute classe, sauf la classe Object, dérive d'une classe, qui s'appelle sa "superclasse". Une classe qui n'indique pas sa superclasse hérite automatiquement de la classe Object (toutes les classes qu'on a vu jusqu'à maintenant).

Lorsque l'on écrit une classe, on annonce sa superclasse à l'aide du mot **extends** en écrivant comme en-tête de la classe :

```
class B extends A{ ........
}
```

On dira alors que :
- **B** hérite de **A**.
- **A** est la superclasse de **B**.
- **B** est une sous-classe de **A**.
- **B** étend **A**.

La sous-classe **B** :
- peut accéder aux membres publics de sa classe de base, comme s'ils étaient écrits dans sa classe
- ne peut pas accéder aux membres privés.
- peut accéder aux membres protégés (**protected**) de sa classe de base.

En Java, une classe ne peut étendre qu'une seule classe, il n'y a pas d'héritage multiple.
On peut interdire qu'une classe soit étendue, il suffit pour cela de lui ajouter le modificateur **final** :

final class A{...}

7.3 Héritage des Constructeurs

Suposons que B soit une sous-classe de A.

Dans la définition d'un constructeur de la classe **B**, on peut vouloir faire appel au constructeur de la superclasse **A** pour remplir par exemple les champs hérités, avant d'initialiser les champs spécifiques à **B**. On utilisera alors le mot **super** qui désigne le constructeur de la superclasse. Attention, Java impose que cet appel soit la première instruction du constructeur de **B**.

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

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

class PointColore extends Point{
   private byte couleur;

   public PointColore(int x,int y, byte c){
      super(x,y);
      this.couleur=c;
   }
```

```
}
```

7.4 Héritage des Méthodes et redéfinition

Si l'on dispose d'une méthode dans A, on peut l'utiliser sur un objet de la sous-classe **B** sans la réécrire.
On peut aussi rédéfinir le comportement d'une méthode, c'est à dire réécrire dans la sous-classe une méthode de mêmes nom ET de signature (nombre et types d'arguments et type de sortie).
Lorsque l'on fait ceci, pour un objet de type **B**, c'est cette méthode qui sera appelée et non plus celle de la superclasse.

```
class Point{
   private int x,y;
...
   public void affiche(){
      System.out.println(''Je suis en ''+ x + '' '' + y)
   }
}
class PointColore extends Point{
   private byte couleur;
...
   public void affiche(){
      System.out.println(''Je suis en ''+ x + '' '' + y );
      System.out.println(''ma couleur est ''+ c);
   }
}
```

Dans l'exemple précédent on pourrait être tenté d'utiliser la méthode affiche de la super classe et d'écrire

```
class PointColore extends Point{
...
   public void affiche(){
       affiche();
      System.out.println(''ma couleur est ''+ c);
   }
}
```

Si on fait cela on provoque un appel récursif et un plantage. Il faut pouvoir préciser qu'on parle de la méthode **affiche** de la superclasse. On utilise pour cela le mot clef **super**.

```
   public void affiche(){
      super.affiche();
      System.out.println(''ma couleur est ''+ c);
   }
```

7.5 Héritage - Polymorphisme

Ce genre d'appel est illégal :

```
PointColore pc = new Point(3,5);
```

en effet, on se doute que cela pourrait poser des problèmes lorsque l'on essaie d'accéder à la couleur de **pc**.

En revanche, Jave autorise l'inverse :

```
Point p = new PointColore(3,5, (byte) 2)
```

Dans ce cas si on applique la méthode **affiche** à **p**, il va utiliser la version qui est contenue dans la classe **PointCol**. Autrement dit l'instruction ne se fonde pas sur le type de la variable **p**, mais bien sur le type effectif de l'objet référencé par **p** au moment de l'appel.

```
p.affiche()
```

va renvoyer

```
Je suis en 3 5
ma couleur est 2
```

Si on est dans le cas précédent :

```
Point p = new PointColore(3,5, (byte) 2)
```

on peut vouloir récupérer l'objet de type **PointColore** pointé par **p** dans une référence du bon type.
On NE pourra PAS écrire

```
PointColore pc = p;
```

cela proquerait une erreur, il faudra alors écrire

```
PointColore pc = (PointColore) p;
```

On parle de transtypage.
Il faudra bien faire attention à ce genre d'appel, cela signifie en gros qu'on dit à Java "ne t'inquietes pas je m'assure que cette référence p contient bien ce qu'il faut". Si jamais à l'éxécution il s'avère que ce n'est pas le cas cela provoquera bien sur une erreur.

Chapitre 8

Collections et boucles «for each»

8.1 ArrayList

On a vu que l'un des inconvénients des tableaux est le fait que l'on doive fixer sa taille à l'initialisation, ce qui implique de devoir en créer un autre et de tout recopier lorsque l'on veut par exemple ajouter un élément de plus.
Il existe en Java des classes permettant de gérer des collections dynamiques, comme la classe **ArrayList**.
On les peut les utilisre de la façon suivante :

```
ArrayList<A> v = new ArrayList<A>();
```

où **A** est le type d'objets stockés. Il s'agit nécessairement d'un type d'objet (comme **Identite** , **String**...) mais pas d'un type primitif (comme int, char, ...).
On écrira donc

```
ArrayList<String> = new ArrayList<String>()
```

On peut utiliser les méthodes :

```
boolean add(A element) // Ajoute un élément en fin de liste
void add(int indice, A element) // Ajoute un élément àun indice donné
void clear() // vide la structure de tous ces éléments
void remove(int indice) // retire l'élément d'indice donné
void remove(A element) // retiré l'élément
A get(int indice ) // retrouve l'élément d'indice donné
boolean contains(A element) // teste l'existence d'un élément dans la collection
int indexOf(T element) // retrouve l'indice d'un élément dans la collectin
int size() // retrouve le nombre d'éléments dans la collection
```

Ainsi, grâce a **get** on peut travailler comme sur un tableau et accéder au i-eme élément, etc. Ainsi on peut utiliser une **ArrayList** en remplacement d'un tableau :

```
ArrayList<String> tab = new ArrayList<String>(); // équivalent String [] tab = new String[2];
tab.add("Bonjour"); // équivalent tab[0] = "Bonjour";
tab.add("Au revoir"); // équivalent tab[1] = "Au revoir";
System.out.println(tab.get(0)); // équivalent: System.out.println(tab[0]);
```

8.2 Boucles «for each»

Depuis la version 5 existe en Java un nouveau moyen de parcourir les tableaux ou les collections. Il s'agit des boucles étendues dites «for each». Auparavant pour parcourir les éléments d'une structure comme un

tableau on utilisait un indice que l'on faisait varier pour aller ensuite chercher l'élément à l'indice donné. Désormais, il est recommandé, lorsque c'est possible, de ne pas utiliser d'indice explicite et d'itérer sur la structure à l'aide la boucle for étendue :

```
String[] t = {''toto'',''titi'', ''tata''};
    // Ancienne façon
    for(int i=0; i<t.length; i++) {
        System.out.println(t[i]);
    }

    // nouvelle façon de faire
    for(String s : t) { // pour s variant sur tous les éléments de t
        System.out.println(s);
    }
```

Cela marche de la même façon avec les **`ArrayList`**, on pourra donc écrire

```
ArrayList<String> v = .....
    for(String b : v){
        ....
```

Chapitre 9

Classes Object, Class

9.1 Classe Object

C'est la classe ont héritent toutes les classes de Java.
Elle contient les méthodes

```
public Class getClass()
//renvoie des informations sous la forme d'un objet de la classe Class
public boolean equals(Object autreObjet)
// retourne true si les 2 references autreObjet et this designent le meme objet.
public String toString()
//"par dEfaut" renvoie ''nom_de_la_classe@valeur de hachage de l'objet''
```

Ces méthodes sont bien sur a redéfinir dans la plupart des classes que l'on écrit.

La méthode **equals** est utilisée par exemple dans la méthode **contains** de la classe Vector.
La méthode **toString** est utilisée dans la méthode **System.out.println**.
Attention, comme on redéfinit la méthode **equals**, elle garde sa signature et prend donc un **Objet** en argument (et pas une variable du type que l'on est en train de définir).
Ainsi, dans la classe **Identite** par exemple cette méthode NE POURRA PAS être écrite de la façon suivante :

```
public boolean equals(Identite i){
   return i.nom.equals(i.prenom) && i.prenom.equals(i.prenom);
}
```

Il faut utiliser le transtypage et cela commencera souvent par quelque chose du genre :

```
public boolean equals(Objet autreObjet){
   Identite i = (Identite) autreObjet;
      .....
```

9.2 Classe Class

La classe **Objet** de Java, dont héritent toutes les classes, possède une méthode **Class getClass()**. La classe Class permet de décrire les propriétés d'une classe particulière. La fonction la plus utilisée de cette classe est la méthode **String getName()**.

```
Indentite i = new Identite(''Pierre'',''Charbit'');
Class c = i.getClass();
System.out.println(c.getName());
```

Il existe un raccourci pour créer un objet de type **Class** consistant à écrire le nom de la classe suivi de **.class**

```
Class c1 = Identite.java;
Class c2 = int.class;
Class c3 = double[].class;
```

Remarquez que **Class** peut aussi désigner un type primitif (qui n'est pas à proprement parler une classe Java) comme **int**.

Cette classe contient aussi une méthode **Class getSuperclass()** qui renvoie un objet de type **Class** correspondant à la super-classe de l'objet.

9.3 Operateur instanceof

L'opérateur boolean **instanceof** s'utilise de la facon suivante

```
Livre l = new Livre(...
System.out.prinltn(l instanceof Livre); //true
System.out.println(l instanceof Document); //true
System.out.println( l instanceof Dvd); //false
```

Un objet n'appartient qu'à une seule classe mais peut etre "instanceof" de plusieurs à cause de l'héritage

Chapitre 10

Javadoc

Java fournit un moyen de documenter ses classees automatiquement. Le résultat est une page html semblable à celles présentes sur ls site `html://java.sun.com/javase/6/docs/api/`

Le principe est d'attacher des élements de doucmentation aux classes, aux champs et aux méthodes grâce à des commentaires particuliers placés dans le code source.

La syntaxe est la suivante :
`javadoc [options eventuelles ] fichier java`
Les options peuvent être par exemple

- -public documente les classes, méthodes et champs public
- -private documente les classes, méthodes et champs private
- -d directory donne le répertoire où sera placée la documentation

10.1 Javadoc

```
/**
des commentaires sur la classe
@author pierre
*/
public class MaClasse {
    /** un champ entier prive */
    private int nombre;

    /** ceci est un constructeur de base
    @author pierre
    @param n un entier positif...*/
    public MaClasse(int n){
    }

    /** voila une fonction qui va renvoyer la valeur de nombre
    @return un entier */
    public int getnombre(){
        return nombre;
    }
}
```

Les tags peuvent être par exemple : **`@author, @param, @return, @see....`**

Exercice 10.1 *Reprenez vos classes Identite, Compte ou Point et documentez les.*

Chapitre 11

Interfaces

11.1 Intro

En java la notion interface peut sembler à première vue se rappocher de la notion de classe.
En effet , elle est aussi contenu dans un fichier .java et se définit de la façon suivante :

```
Interface I
<des champs final static>
<des methodes>
```

Le mot **`Interface`** remplace le mot **`Class`** et l'interface contient aussi des champs (qui sont des constantes) et des méthodes. Cependant, dans une interface, il n'y a pas de constructeur et les méthodes ne sont pas décrites. Les interfaces ne sont pas destinées a être instanciées. On ne pourra donc pas écrire l'instruction **`Interface I = new Interface()`**
En fait on va utiliser les interfaces un peu comme on utilise l'héritage.
(Note : dans le cours, on utilisera souvent les interfaces propsées par Java dans la librairie mais on n'en écrira pas nous mêmes).
On pourra dire dans la définition d'une classe qu'elle implémente une interface, un peu comme si on disait qu'elle hérite d'une autre classe.
La syntaxe est la suivante si I est du type Interface : **`class A implements I{....}`**

Alors la classe A dispose automatiquement des constantes définies dans l'interface et **est obligée** d'implémenter (cad de décrire le code de) toutes les méthodes de l'interface en respectant leur signature.
On impose à la classe qui implémente l'interface de respecter un cahier des charges, on lui impose des règels à respecter pour pouvoir implémenter cette interface.

11.2 Exemple : Comparable

Supposons que l'on s'interresse au tri d'objets. On a par exemple un tableau **`Classe [] tab`** où **`Classe`** peut etre **`String`** (par exemple ordre alphabétique), **`Compte`** (selon la valeur du solde), **`Identite`**....
Quel que soit le type d'objets, une fois qu'on a choisi un algorithme, on l'écrira de la meme façon, cet algo ne dépend que d'une chose : avoir à notre disposition une fonction de comparaison, un critere nous permettant de dire qu'un objet est plus petit ou plus grand qu'un autre.
Cela serait idiot de réecrire ce bout de code à chaque fois, on veut pouvoir le mettre dans une boite à outils sur les tableaux (cette boite existe : voir **`java.util.Arrays`**).

Il existe ainsi dans la libraire de base une Interface qui s'appelle **`Comparable`**.
Elle impose une seule chose : la méthode **`public int compareTo(Objet o)`** qui doit nécessairement :

- renvoyer -1,0,ou 1 (selon si l'élement **o** passé en argument est plus grand, egal, ou plus petit que celui passé en préfixe)
- être transitive (si $a < b$ et $b < c$, alors $a < c$)

Attention, ici l'argument doit nécessairement être du type **Object**, même si on est à l'intérieur d'une classe **Identite**, **Compte** .... puisque l'on rédéfinit la méthode décrite dans l'interface Comparable (on respecte donc sa signature)
Il faut faire donc appel au transtypage en écrivant qq chose du genre **Identite e = (Identite) o;** au début du corps de la méthode pour ensuite travailler avec la référence **e**.

11.3 Héritage multiple ?

Une des grandes différences entre héritage et utilisation d'interfaces est la possibilité d'implémenter plusieurs interfaces différentes. On écrira tout simplement :
class A implements I1, I2, I3{...
Cela correspond bien avec l'idée de structure imposée correspondant aux interfaces par rapport à l'idée de sous classe de l'héritage.

Par exemple, on pourra imaginer avoir une interface **Dessinable** qui décrit tout un tas de comportement propre aux objets destinés à être déssinés et certaines classes pourrontt être à la fois ordonnables et dessinables, d'autres justes ordonnables...

11.4 Remarques

- Dans une Interface, tous les champs et méthodes sont implicitement **public**, donc les classes qui implementent une interface doivent declarer **public** les méthodes dont elles "héritent".
- si une classe **A** implemente une interface, toute sous-classe de **A** impléménte automatiquement cette interface.

Exercice 11.1 *On veut implementer la classe* ***Identite*** *avec l'interface* ***Comparable****. Le but est de trier les identités par dates de naissance croissantes. Pour cela réécrivez la méthode* ***compareTo*** *pour qu'elle respecte les contraintes sur l'argument de type* ***Object****.*
Tester maintenant cette implémentation en triant un tableau d'

Chapitre 12

Entrées Sorties

12.1 Sérialisation

Java possède un mécanisme simple d'emploi pour sauvegarder ses objets dans des fichiers. Ceci afin qu'ils puisse exister encore une fois le programme terminé (on parle de transcendance temporelle) ou de pouvoir les échanger entre application (on parle de transcendance spatiale). Il s'agit de la **sérialisation**. Cela peut s'avérer très pratique pour conserver des données à la fermeture de l'application afin de les recharger au lancement suivant.
Pour pouvoir sauver un objet, il faut que leur classe implémente l'interface **`Serializable`**. Contrairement à la plupart des interfaces, celle-ci ne requiert la redéfinition d'aucune méthode particulière, on doit juste préciser que l'on donne la possibilité à cette classe d'être sérialisée.
Comme pour écrire du texte, on utilise des classes spécifiquement dédiées à la lecture ou à l'écriture d'objets.

Écrire un objet

C'est un poil plus compliqué que pour écrire du texte. On passe par un objet de type **`ObjectOutputStream`** qui représente un flot d'écriture d'objets et qui se construit a partir d'un flot d'écriture dans un fichier, de type **`FileOutputStream`**. Ce dernier se construit à partir d'un fichier (objet de type **`File`** ou directement chaine de caractères du nom)

```
try {
   ObjectOutputStream flot = new ObjectOutputStream(new FileOutputStream(fichier));
   flot.writeObject(objet);
   flot.close();
} catch (IOException e) {
   System.out.println("Erreur ecriture " + e) ;
}
```

Lire un objet

C'est quasiment la même chose que pour l'écriture, en remplaçant les mot **`Output`** par des **`Input`**. La méthode **`readObject()`** renvoie un **`Objet`** qui peut être stocké.

```
try {
   ObjectInputStream flot = new ObjectInputStream(new FileInputStream(fichier));
   Object objet = flot.readObject();
   flot.close();
   return objet;
} catch (Exception e) {
   System.out.println("Erreur lecture "+fichier+e.getMessage())
```

```
}
```

Une classe à ajouter à ses projets

Voici un exemple de classe utilitaire que l'on peut placer dans un projet quelconque et qui ajoute deux fonctions statiques **sauver** et **restaurer**.
À l'utilisation, il suffira de taper une instruction du genre **Serialise.sauver(monobjet,monfichier)** pour sauver dans un fichier, et **Object o = Serialise.restaurer(monfichier)** pour le restaurer à partir d'un fichier donné.

```
public class Serialise{
   public static void sauver(Object objet, String fichier) {
      try{
         ObjectOutputStream flot=new ObjectOutputStream(new FileOutputStream(fichier));
         flot.writeObject(objet);
         flot.close();
      }catch (IOException e) {
         System.out.println("Erreur ecriture " + e.getMessage());
      }
   }

   public static Object restaurer(String fichier) {
      try{
         ObjectInputStream flot = new ObjectInputStream(new FileInputStream(fichier)) ;
         Object objet = flot.readObject() ;
         flot.close();
         return objet;
      }catch (Exception e) {
         System.out.println("Erreur lecture "+fichier+" :" +e.getMessage()) ;
         return null;
      }
   }
}
```

Chapitre 13

Java et Base de Données

13.1 Intro

Il existe plusieurs bases de données différentes (PostgreSQL, MySQL, SQL server, Oracle, Access....). On choisit ici mySQL mais le fonctionnement des interactions avec java est tres similaire.
Dans tous les cas, java a besoin d'un **Driver** pour communiquer avec la base de donnée, et ce driver est spécifique à la base. Une recherche sur Google permet de télécharger le dirver en question qui est un **.jar**.
On n'expliquera pas ici comment fonctionnent les bases de données (vous avez eu un cours, non ?), on supposera qu'on a une base de données **`mabase`**.
Bien sur, tout ce qui sera raconté ici n'est qu'un bref aperçu des choses élémentaires, pour aller plus loin, il faudrait plus d'heures de cours que ce dont nous disposons.

13.2 Se connecter à la base.

Voici un exemple de code permettant de se connecter à une bdd.

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

   try {
      String url = "jdbc:mysql://localhost/mabase";
      String user = "root";
      String passwd = "";

      Connection conn = DriverManager.getConnection(url, user, passwd);

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

- Pour pouvoir communiquer il faut créer la connection avec notre base. La commande **`getConnection`** prend en argument l'url, le nom d'user et le password.
- l'url est consitituée du protocole (**`jdbc:mysql`**), suivi de l'adresse de la machine hébergeant la base (ici, **`localhost`**), suivi du nom de la base de données (ici **`mabase`**).
- Les méthodes manipulant des bases de données sont susceptibles de générer des exceptions, c'est pourquoi les instructions sont entourées d'un bloc **`try catch`**.

13.3 Lire dans la base : **Statement** et **ResultSet**

Pour pouvoir executer des requêtes SQL, on a besoin d'un objet **Statement**. On le crée avec l'instruction suivante :

```
Statement st = conn. createStatement();
```

où **conn** est notre objet de type **Connection**.

Ensuite on peut effectuer une requête. Cela s'effectue via la méthode **executeQuery** qui prend en argument une chaîne de caratères correspondant à la requête SQL.
Supposons que la base contienne la table **table1**, on pourra donc écrire :

```
ResultSet result = st.executeQuery("SELECT * FROM table1");
```

L'objet de type **ResultSet** est celui qui contient le résultat de la requête et se comporte comme une tête de lecture qui est positionnée au début avant la première ligne de la base.
Avec la commande **result.next()**, on passe à la ligne suivante. Cette commande renvoie false si on est à la dernière ligne. On peut donc écrire :

```
while(result.next()){
    .... // Une commande pour afficher le contenu, voir transparent suivant
}
```

13.4 Lire dans la base : **Statement** et **ResultSet**

On peut récupérer les infos de la ligne courante avec des instructions de type **get**.
Si on ne connait pas le nom des colonnes on utilisera **getObjet(int i)**. Cela renvoie l'objet contenu dans la i-eme colonne de la ligne courante (que l'on pourra convertir avec la méthode **toString()** si on veut l'afficher). Si la table a deux colonnes on pourra pour afficher la table écrire par exemple (attention les indices commencent à 1 dans mysql) :

```
while(result.next()){
   System.out.print(result.getObjet(1).toString());
   System.out.print("|");
   System.out.println(result.getObjet(2).toString());
}
```

A la fin il ne faut pas oublier de fermer les objets :

```
result.close();
st.close();
```

On peut récupérer les infos de la ligne courante avec des instructions de type **get**.

Si on connait le nom et le type de données de chaque colonne on pourra utiliser un get approprié, comme **getString**, ou **getInt**, **getDouble**... Si par exemple on a une colonne "nom" et une colonne "prenom", on pourra écrire :

```
while(result.next()){
   System.out.print(result.getString("nom");
   System.out.print(" | " );
   System.out.print(result.getString("prenom");
}
```

pour afficher la table.

13.5 Lire dans la base : **ResultSetMetaData**

L'objet **Result** nous permet de lire la requête mais ne permet pas d'obtenir des informations globales, comme le nombre de lignes le nombre des colonnes ou leur nom. Pour cela on passe par la classe **ResultSetMetaData**.

```
Statement st = conn.createStatement();
ResultSet result = st.executeQuery("SELECT * FROM mabase");
ResultSetMetaData resultMeta = result.getMetaData();
int n = resultMeta.getColumnCount();
for(int i=1 ; i<= n ;i++){
   String s= resultMeta.getColumnName(i);
   System.out.print(s +"\t");
}
```

13.6 Résumé

```
try{
   String url = "jdbc:mysql://localhost/mabase";
   Connection conn = DriverManager.getConnection(url, "root","");

   Statement st = conn.createStatement();

   ResultSet result = st.executeQuery("SELECT * FROM mabase");
   ResultSetMetaData resultMeta = result.getMetaData();
   int n = resultMeta.getColumnCount();
   for(int i=1 ; i<= n ;i++){
      String s= resultMeta.getColumnName(i);
      System.out.print(s +"\t");
   }

   while(result.next()){
      System.out.print(result.getString("nom");
      System.out.print(" | " );
      System.out.print(result.getString("prenom");
   }
   result.close();
   st.close();
}
catch(Exception e){
   e.printStackTrace();
}
```

13.7 Ecrire dans une base

Les requetes d'ériture se font via l'objet **Statement** déjà utilisé pour la lecture, en utilisant cette fois la méthode **executeUpdate(String s)** qui prend en argument une chaîne de caractères correspondant à l'instruction mySQL de type **UPDATE, INSERT, DELETE, CREATE**.

Si notre table **table1** avec les deux colonnes **nom** et **age** contient une ligne de nom pierre, alors l'instruction suivante change l'age de pierre.

```
Statement st = conn.createStatement();
st.executeUpdate("UPDATE table1 SET age=20 WHERE nom LIKE 'pierre' ");
```

À noter, il existe aussi des méthodes de la classe **ResultSet** permettant de modifier la table.

```
ResultSet result = st.executeQuery("SELECT * from table1");
result.next(); // on passe sur la premiere ligne
result.setString("age",25); // on change la valeur de "age " pour cette ligne.
```

13.8 Écrire dans une base - Requêtes Préparées

La Classe **PreparedStatement** étend la classe **Statement**. Deux différences notables :

- Les requêtes sont pré-compilées en SQL, ce qui implique un gain d'efficacité, surtout si la requête doit être appelée plusieurs fois
- On peut, à la manière des **prepare** en **php**, faire des requêtes à trous

Exemple :

```
String req = "SELECT * from matable WHERE nom = ? OR id =?" ;
PreparedStatement pst = conn.preparedStatement(req);

pst.setString(1,"Pierre");
pst.setInt(2,"32");
pst.executeUpdate();

pst.setString(1,"toto");
pst.executeUpdate();
```

Chapitre 14

Interfaces graphiques à la main

Il est possible de construire des interfaces graphiques (GUI : graphical user interface) à l'aide des IDEs qui souvent proposent de quoi les éditer simplement et directement en mode graphique. Toutefois, il est parfois nécessaire de mettre la main dans le code généré, auquel cas il est essentiel de connaître la façon dont les interfaces sont composées. On ne traitera ici que de la couche graphique Swing qui repose sur AWT. Programmer directement en AWT est considéré comme obsolète, mais si cela reste possible pour des raisons de compatibilité. Il existe depuis 2008 la couche d'interface JavaFX considérée comme plus «moderne» (facilement stylable, etc), mais pour diverses raisons elle ne fait plus partie de la distribution des dernières versions de Java par Oracle ; c'est devenu un produit libre à installer séparément. On en parlera donc pas.

14.1 Introduction

Une interface graphique est composé en java (mais comme toujours) de l'assemblage de plusieurs composants : fenêtre, zone dans des fenêtre, barre de boutons, boutons, zones de texte, etc. Les composants :
- peuvent être «terminaux» (boutons, listes, etc.),
- peuvent contenir d'autres composants (barre d'outil, menu, fenêtre, etc.) : ce sont des conteneurs (containers), ils sont tous dérivés de la classe **`Container`**.

Parmi les conteneurs, on distingue les conteneurs intermédiaires qui peuvent être contenus dans d'autres conteneurs (barre de boutons, panneau), et les conteneurs «racine» (top-level) : qui ne peuvent pas être contenus ans d'autres composants (fenêtre).
L'instruction de base de la composition est donc la méthode qui permet d'ajouter un composant à un conteneur : **`add(Component composant)`**

14.2 Un aperçu rapide des composants Swing

- Conteneurs
  - top level : **`JFrame`**, **`JApplet`**,
  - génériques : **`JPanel`**, **`JScrollPane`**, **`JSpiltPane`**, **`JTabbedPane`**,
  - spécifiques : **`JInternalFrame`**, **`JLayeredPane`**, **`JMenuBar`**.
- Contrôle de base
  - boutons : **`JButton`**, **`JToggleButton`**, **`JCheckBox`**, **`JRadioButton`**,
  - boites à déroulement : **`JComboBox`**,
  - menus : **`JMenu`** ,**`JMenuItem`**,
  - curseurs : **`JSlider`**,
  - textes : **`JTextField`**, **`JTextArea`**,
- Affichage simple : étiquette **`JLabel`** et barre d'avancement **`JProgressBar`**,
- Dialogues : **`JDialog`**, **`JOptionPane`**, **`JFileChooser`**, **`JColorChooser`**.

14.3 Le composant conteneur le plus simple : **JFrame**

Une fenêtre s'obtient en instanciant un objet de la classe **JFrame**. C'est le conteneur racine le plus utile. Elle contient de quoi accueillie une barre de menus, et un composant représentant le contenu de la fenêtre. En effet un objet tel que **JFrame** fait le pont entre l'interface graphique du système hôte et l'interface graphique sus le contrôle de l'application.
Pour fixer ou récupérer ces objets on dispose des méthodes suivantes :

```
JMenuBar getJMenuBar()
void setJMenuBar(JMenuBar b)
Container getContentPane()
void setContentPane( Container c )
```

Une fois l'objet construit pour qu'il soit visible à l'écran, il faut le rendre visible par

```
this.setVisible(true);
```

14.4 Quelques exemples

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

class TestFenetre1 {
   public static void main(String[] args){
      JFrame fen = new JFrame("premiere fenetre");
      fen.setSize(300,200);
      fen.setLocation(100,100);
      fen.setVisible(true);
   }
}
```

```

class TestFenetre2 {
   public static void main(String[] args){
      Dimension dim =
      Toolkit.getDefaultToolkit().getScreenSize();
      int larEcran = dim.width;
      int hauEcran = dim.height;
      int larFenetre = larEcran/3;
      int hauFenetre = hauEcran/4;
      JFrame cadre = new JFrame("Titre de la fenetre vide");
      cadre.setSize(larFenetre,larFenetre);
      cadre.setLocation((larEcran-larFenetre)/2,(hauEcran-hauFenetre)/2);
      cadre.setVisible(true);
   }
}
```

```
class TestFenetre3{
   public static void main(String[] args){
      JFrame fen= new JFrame("avec un bouton et un label");
      fen.setSize(60,60);
```

```
      fen.setLocation(100,100);
      fen.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      Container c = fen.getContentPane();
      c.setLayout(new FlowLayout());
      JButton b = new JButton("toto");
      JLabel l = new JLabel("ceci est une zone de texte");
      c.add(b);
      c.add(l);
      fen.pack();
      fen.setVisible(true);
   }
```

```
   public Fenetre(){
      this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      JPanel p1= new JPanel();
      JLabel l=new JLabel("encore une zone de texte");
      p1.add(l);

      JPanel p2 = new JPanel();
      JButton b1=new JButton("appuie");
      JButton b2=new JButton("valide");
      p2.add(b1):p2.add(b2);

      JPanel contenu=new JPanel();
      this.setContentPane(contenu);
      contenu.add(p1);contenu.add(p2);
   }

   public static void main(String[] args){
      Fenetre f = new Fenetre();
      f.setVisible(true);
   }
}
```

14.5 Remplir un JFrame

Si l'on dispose d'une instance **fen** de la classe **JFrame** (ou d'une classe dérivée), il existe principalement trois méthodes pour la remplir :

- récupérer le conteneur associé par **getContentPane()**, et lui ajouter des composants. Attention le conteneur récupéré est de la classe **Container**, qui est plus basique qu'un **JPanel** par exemple.
- créer un conteneur particulier (par exemple un JPanel) et l'affecter en tant que contenu de la fenêtre, par la méthode **setContentPane**.
- rajouter des composants à la fenêtre par **fen.add(...)**. Dans ce cas ils sont rajoutés directement au conteneur que l'on aurait obtenu avec un **get** par la première méthode. Cependant on ne peut ainsi décider du gestionnaire d'agencement (voir plus loin).

14.6 Agencement et LayoutManager

14.6.1 Principe

L'agencement et la taille des composants dans un conteneur est gérée de façon plus ou moins automatique par Java et n'est pas déterminé par le conteneur lui même, mais est délégué à un objet de la classe Layout

Manager. Fixer le gestionnaire d'agencement (Layout Manager) d'un conteneur s'effectue par la méthode suivante de la classe **`java.awt.Container`** : **`public void setLayout(LayoutManager manager)`**.
AWT et Swing fournissent (entre autres) les Layout Managers suivants :

- **`FlowLayout`**,
- **`BorderLayout`**,
- **`GridLayout`**,
- **`BoxLayout`**.

14.6.2 FlowLayout

Le gestionnaire le plus simple, et conteneur par défaut des Container standards.
Les composants sont ajoutés et placés comme on agence les mots d'un texte, c'est-à-dire de gauche à droite et si il n'y a plus de place, en passant à la ligne suivante, etc. Avec ce gestionnaire, la taille d'un composant est toujours la plus petite possible. On pourra utiliser ce gestionnaire pour commencer un projet, lorsque l'on ne veut pas se soucier encore de l'agencement. Il sera utile aussi pour mettre des objets facilement à l'intérieur d'un sous panneau de la fenêtre. C'est d'ailleurs toujours une bonne idée que de ne pas se préoccuper de l'agencement exact au moment du développement. Il n'y a aucun raison que le fonctionnement du cœur de l'application dépende en quoi que ce soit de l'agencement à l'écran.

14.6.3 BorderLayout

Un conteneur muni d'un **`BorderLayout`**, est logiquement divisé en 5 zones : North, South, East, West, et Center, chacune pouvant recevoir au plus un composant (si on veut en mettre plusieurs il faut un conteneur avec des composants en son sein). On peut choisir dans quelle case on place un composant en écrivant : **`container.add(composant,"north")`** ou **`container.add(composant,"Center");`** ou assez souvent **`container.add(composant.BorderLayout.SOUTH);`**.
L'agencement a les propriétés suivantes : le composant central prend le maximum de place sans écraser les autres, le composant est prend toute la place verticale mais le minimum horizontalement, etc. Si une zone est vide de composants, elle est réduite à zéro. Par exemple, on peut avoir un panel au centre et un autre au sud, pour avoir une fenêtre principale et une zone de boutons en bas.
Attention, lorsque l'on extrait le contenu d'une fenêtre par **`getContentPane()`**, le **`Container`** renvoyé dispose par défaut d'un **`BorderLayout`** (alors que sinon un **`Container`** dispose par défaut d'un layout stupide). Ainsi si **`fen`** est une instance de **`JFrame`**, on pourra écrire directement **`fen.add(comp,"North");`**.

14.6.4 BoxLayout

Ce Layout permet de disposer les composants dans une seule direction (ligne ou colonne).
Attention la syntaxe est un peu différente des autres layouts.

```
JPanel p = new JPanel();
BoxLayout ges = new BoxLayout(p,Y_AXIS);
// Le constructeur prend le panneau p en argument
p.setLayout(ges);
```

On utilise la constante **`Y_AXIS`** pour un mode vertical ou **`X_AXIS`** pour un mode horizontal.

14.6.5 GridLayout

Ce Layout permet de disposer des composants sur une grille régulière ou toutes les cases auront même taille (voir **`GridBagLayout`** pour des choses encore plus sophistiquées). Le constructeur prend en argument deux entiers correspondant aux nombre de lignes et colonnes .
Ainsi **`new GridLayout(5,3)`** permettra d'avoir 5 lignes et 3 colonnes.
On rajoute les composants par codeadd(Composant). On ne peut pas choisir directement où on les place, les composants sont rajoutés de gauche à droite et de haut en bas en partant du coin supérieur gauche pour finir au coin inférieur droit.

Attention dans cet agencement, les composants prennent la taille maximale possible dans leur case. Ainsi un **JButton** pourra occuper une place plus grande que prévue.

14.7 Gérer la taille des composants, commande pack()

On peut fixer la taille d'une fenêtre avec la méthode **setSize(int,int)** mais ce n'est absolument pas conseillé. En effet comment déterminer cette taille sin on ne connaît rien de l'environnement graphique d'exécution ? Cependant c'est donc une meilleure idée que de laisser le calcul de l'agencement à l'objet qui en est chargé, libre à lui de déterminer au mieux les tailles respectives vis-à-vis des contraintes imposées par les différent gestionnaires d'agencement et les composants eux-mêmes. Pour cela on conclut les instructions de constructions de la fenêtre par la méthode **pack()**.

Pour choisir la taille des sous panneaux, on utilisera de préférence la méthode : **void setPreferredSize(dim)**

qui prend en argument une variable **dim** de la classe **Dimension**. Une dimension est la donnée de deux entiers correspondant à largeur-hauteur exprimée en pixels.
Ainsi on pourra écrire par exemple :

```
JPanel p = new JPanel();
p.setPreferredSize( new Dimension(300,500) ) ;
```

14.8 La Barre de Menu JMenuBar

Une barre de menu est un **Container** particulier qu'on utilise généralement habituellement dans une **JFrame**, cet objet ayant de quoi gérer correctement une barre de menu. C'est un objet de la classe **JMenuBar** . Pour rajouter un menu déroulant, il faut lui ajouter un **JMenu** avec la commande **add**, le constructeur prenant en argument le nom du menu.
À l'intérieur d'un **JMenu**, on pourra rajouter des **JMenuItem** qui seront les choix possibles. On peut éventuellement rajouter un **JMenu** à l'intérieur d'un **JMenu** pour faire des sous-menus. Il existe quelques autres composants particulier que l'on peut ajouter dans des menus...

```
JMenuBar barre = new JMenuBar();

JMenu fic = new JMenu("File");
barre.add(fic);
JMenuItem fic1 = new JMenuItem("Open");
fic.add(fic1);
JMenuItem fic2 = new JMenuItem("Save");
fic.add(fic2);

JMenu ed = new JMenu("Edit");
barre.add(ed);
...
maFrame.setJMenuBar(barre);
```

14.9 Ça ferme (pas) vraiment ?

Lorsque l'on clique sur le bouton de fermeture d'un **JFrame**, la fenêtre disparaît mais le programme java qui a lancé la fenêtre ne se termine pas. Typiquement, si on a lancé le programme d'un terminal, alors on ne récupère pas la main.
Pour spécifier le comportement d'une fenêtre lors de la fermeture on utilise la méthode : **setDefaultCloseOperation(int**
qui prend en argument une constante parmi :

— **HIDE_ON_CLOSE** pour faire disparaître la fenêtre (comportement par défaut),
— **EXIT_ON_CLOSE** : pour quitter l'application qui a lancé la fenêtre,
— **DO_NOTHING_ON_CLOSE** pour ne rien faire du tout,
— **DISPOSE_ON_CLOSE** pour détruire la fenêtre mais pas le programme (utile lorsque l'on a une application avec plusieurs fenêtres).

Pour faire des choses plus spécifiques, il faudra munir la fenêtre d'un écouteur d'évènements de type **WindowListener** (voir plus loin pour les évènements).

14.10 Evènements et Ecouteurs

Jusqu'à maintenant, on a construit des fenêtres avec menus et boutons mais on ne s'est pas occupé du résultat d'un clic ou de n'importe quelle autre action.
En Java tout changement d'état, que ce soit un déplacement ou un clic de souris, l'appui sur un bouton, ou le déplacement d'une fenêtre génère un évènement (classe **Event**).

Un évènement va en suite être attrapé par un écouteur (classe **Listener**) qui va décider des opérations à effectuer en conséquence. Il existe plusieurs types d'évènements dont les plus courants sont

Source d'Interaction	*Ecouteur associé*	*méthodes de l'écouteur*
JButton, JTextField,..	ActionListener	actionPerformed(actionEvent e)
clic de souris	MouseListener	mouseClicked(), mousePressed() mouseEntered() mouseReleased() mouseExited()
mvt de souris	MouseMotionListener	mouseMoved(), mouseDragged()
JCheckBox	ItemListener	itemStateChaged()

Pour réagir à ces évènements il faut doter l'objet concerné d'un écouteur d'événement ou **Listener**.
Ces classes appartiennent au package **java.evt.events** qu'il faudra donc importer.
La commande pour ajouter un écouteur sur un composant est :ß
<composant>.addXXXListener(<un ecouteur approprie>)ß
où XXX est selon le cas **Action, MouseMotion, MouseMoved, ...**. L'écouteur est un objet est devra donc être instancié.
Prenons l'exemple d'un bouton et donc du type **ActionListener**. Ce n'est pas une classe mais une interface, ce qui implique que n'importe quel objet peut jouer le rôle d'écouteur, à condition qu'il implémente l'interface dans sa définition. C'est pourquoi plusieurs possibilités s'offrent à nous :
— Utiliser par exemple la fenetre comme écouteur pour toutes les actions en lui faisant implémenter l'interface **ActionListener**, ce qui force la fenêtre à redéfinir la méthode **actionPerformed()**
— Créer une nouvelle classe pour chaque objet écouté et qui ne sert qu' à ça. La classe en question ne contiendra en gros que la redéfinition de la méthode d'action.

La fenêtre comme écouteur

Ce n'est pas la "bonne méthode" si l'on veut faire du beau code, mais pour débuter cela permet quand même de faire des choses.
Pour cela il faut que la fenetre ne soit pas juste un JFrame mais une nouvelle classe héritant de JFrame et qui implémente l'interface.

```
public class MaFenetre extends JFrame implements ActionListener{.....
```

Dans ce cas, on ajoute la fenêtre comme écouteur d'un objet dans le constructeur de la fenetre par des commandes du type :

```
bouton.addActionListener(this)
```

La seule contrainte est de redéfinir les méthodes associées. La syntaxe pour une Action est par exemple

```
public void actionPerformed(ActionEvent e){<actions a effectuer>}
```

On récupère alors la source de l'évènement par **`e.getSource()`**.

Remarque 1 *Attention avec* ***`e.getSource()`*** *: comme on ne sait pas a priori quel sera l'objet qui déclenche, cette méthode renvoie un* ***`Object`****. Il faut donc écrire :*

```
Object o = e.getSource()
```

et ensuite faire des ***`if, else`****sur* ***`o`*** *pour savoir quel bouton, quel item de menu, a déclenché l'action pour agir en conséquence.*

Remarque 2 *Parfois (par ex, pour MouseListener) il existe 4 ou 5 méthodes mais on ne se sert que d'une seule, dans ce cas il faudra quand même écrire les autres mais en ne mettant rien dans le bloc d'accolades correspondant :*

```
public void mouseClicked(){ <du code> }
public void mousePressed{}
public void mouseEntered{}....
```

Si l'on adopte cette méthode, il ne faut pas oublier que les objets (champs de texte, panneaux...) qui interviennent dans la méthode actionPerformed doivent être déclarés comme champs de la classe qui définit l fenêtre et non uniquement dans le constructeur. Rappelons que la durée de vie d'une variable est limitée au bloc d'accolades dans lequel elle se trouve.

Ecouteurs perso

C'est la méthode recommandée pour des programmes plus sérieux. On définit sa propre classe :

```
class MonEcouteur implements ActionListener{
   public void actionPerformed(ActionEvent e){<instructions>}
}
```

Ensuite lors de la construction de notre interface graphique, on rajoute l'écouteur sur un bouton (par exemple) par : **`bouton.addActionListener(new MonEcouteur())`**

Là encore, se pose la question de comment faire intéragir l'écouteur avec des composants de notre fenêtre qui sont situés dans la classe principale. Une méthode peut être d'avoir ces objets comme champs de la classe écouteur mais cela oblige à les initialiser lorsqu'on crée l'écouteur, et donc de définir un nouveau constructeur qui les initialise.
La meilleure méthode est d'écrire la classe d'écouteur comme une *classe interne* de la classe de la fenetre. Nous n'avons pas vu dans ce cours ce concept mais il n'est pas très compliqué. Cela consiste à écrire le code de la classe directement à l'intérieur du code de l'autre classe, un peu comme si on écrivait une méthode supplémentaire. La classe n'est alors accessible que à partir de la classe qui la contient (on ne peut pas l'instancier à l'extérieur) mais l'intérêt est qu'elle a accès a tous les champs de la classe.

Exemple :

```
class MaFenetre{
//champs
JButton bouton ;
....
....
//constructeur
   public MaFenetre(){
```

```
        ....
        bouton=new JButton(''appuie'');
        bouton.addActionListener(new MonEcouteurBouton());
        ....
    }

//methodes
....
//classes internes
    class MonEcouteurBouton{
        public void actionPerformed(ActionEvent e){
            //instructions
        }
    }

}
```