La Programmation en Java

U.F.R. d'informatique

Juliette Dibie

Plan

Plan détaillé

Plan

Plan

I. Introduction

I.1. Petit historique

1991 Naissance de Java chez Sun Microsystems.

1995 Réalisation du logiciel HotJava, un navigateur Web écrit par Sun en Java.

Les autres navigateurs Web ont suivi, ce qui a contribué à l'essor du langage sous forme de versions successives :

- 1996 : apparition du premier JDK sur le site de Sun Microsystems
- 1998 : lancement de Java 2 avec le JDK 1.2
- 2002 : J2SE 1.4 (J2SE : Java 2 Standard Edition)
- 2004 : J2SE 5.0
- 2006 : Java SE 6

I.2. Caractéristiques

I.2.1. Java : un langage de programmation orientée objet

- **Objet** : ensemble de **données** et de **méthodes** agissant exclusivement sur les données de l'objet.

- **Encapsulation des données** : accès aux données d'un objet par ses méthodes, qui jouent ainsi le rôle d'**interface** obligatoire. On dit que l'appel d'une méthode d'un objet correspond à l'envoi d'un **message** à l'objet.
  - Pas d'accès à l'implémentation d'un objet.
  - Facilite la maintenance et la réutilisation d'un objet.

- **Classe** : description d'un ensemble d'objets ayant une structure de données commune et disposant des mêmes méthodes. Un objet est une **instance** de sa classe.

- **Héritage** : permet de définir une nouvelle classe à partir d'une classe existante, à laquelle on ajoute de nouvelles données et de nouvelles méthodes.
  - Facilite la réutilisation de classes existantes.

  Certains langages, tels C++, offrent la possibilité d'un héritage multiple (une même classe peut hériter simultanément de plusieurs autres). Ce n'est pas le cas de Java.

I.2.2. Java et la portabilité

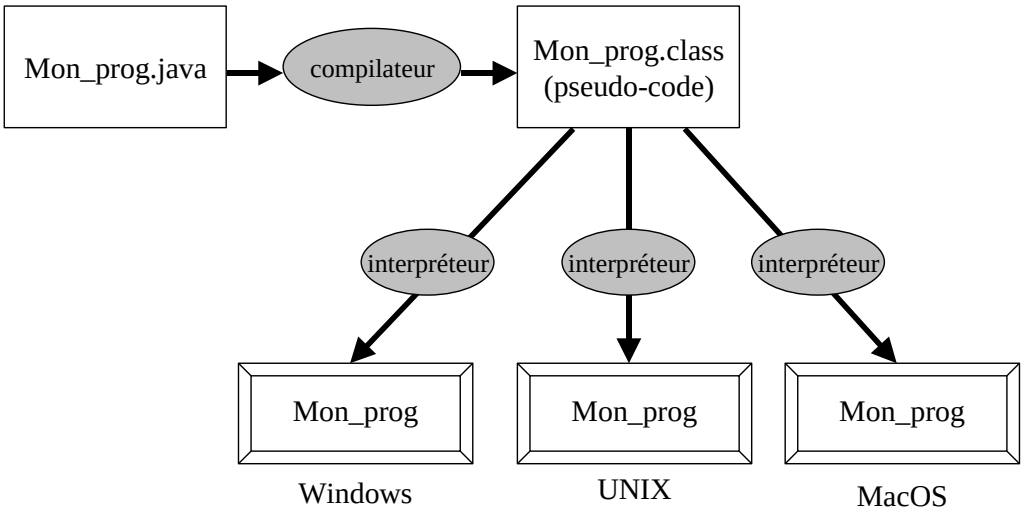
Dans la plupart des langages de programmation, un programme **portable** est un programme dont le code peut être exploité dans différents environnements moyennant simplement une nouvelle compilation.

En Java, la portabilité va plus loin…

En java, un programme est à la fois compilé et interprété.

- La compilation d'un programme Java produit un code intermédiaire exécutable formé de **bytecodes**, qui est un pseudo code universel disposant des fonctionnalités communes à toutes les machines.

- L'interpréteur Java appelé **machine virtuelle** (JVM pour Java Virtual Machine) permet d'exécuter le bytecode produit par le compilateur.



I.2.3. Applets et applications autonomes

A l'origine Java a été conçu pour réaliser des applets s'exécutant sous le contrôle d'un navigateur Web. Les applets sont un flop, réorientation vers les applications serveur avec J2EE (Java 2 Enterprise Edition) en 1999 (devenu aujourd’hui Java EE 5).

Java permet d'écrire des programmes indépendants du Web. On parle alors d'**applications autonomes**. Une application autonome s'exécute sous le contrôle direct du système d'exploitation. Il n'y a pas besoin de navigateur Web pour l'exécuter.

II. La syntaxe du langage Java

II.1.Un premier programme

```
public class PremProg
{
   public static void main(String args[])
   {
      int i, total=0, n=5;
      for(i=1;i<=n;i++)
      total=total+i;
      System.out.println("somme des entiers de 1 a " + n +
                                    " = " + total);
   }
}
```

Une application peut être formée d'une seule classe, la classe PremProg.

Le mot clé public dans public class PremProg sert à définir les droits d'accès des autres classes (en fait de leurs méthodes) à la classe PremProg.

Il n'y a pas de fonctions "non membres", il n'y a que des méthodes.

La méthode main est une méthode particulière qui représente le code à exécuter.

System.out.println correspond à une fonction d'affichage dans la fenêtre console.

Le résultat de l'exécution de ce programme est :

```
somme des entiers de 1 a 5 = 15
```

- *Java dispose de trois formes de commentaires :*
  commentaires usuels placés entre les délimiteurs / et */ ;*
  commentaires de fin de ligne introduits par le délimiteur // ;
  *commentaires de documentation entre les délimiteurs /** et */ : leur intérêt est de pouvoir être extraits automatiquement par des programmes utilitaires de création de documentation tels que Javadoc (création d'une aide en ligne au format HTML à partir de fichiers source Java correctement documentés).*

TD1. Un premier programme

Préparation des fichiers

Dans votre espace disque personnel, créez un répertoire TD_java et copiez, sous ce répertoire, les répertoires Java_td_classe, Java_td_heritage, Java_td_interface, Java_td_collection, Java_td_collection_v2, Java_td_exception indiqués par l'enseignant (copiez les répertoires avec toute leur descendance).

Un premier projet Java

Lancez Eclipse et créez un projet Java TD01 en suivant les indications de l'animateur de session.

Créez une nouvelle classe PremProg.java « File – New – Class », qui correspond à la première version de la classe PremProg, telle qu'elle est donnée dans le cours.

Le code source d'une classe publique doit toujours se trouver dans un fichier portant le même nom et possédant l'extension java.

Compilez cette classe « CTRL + s ». Si la compilation s'est bien déroulée, vous obtiendrez un fichier portant le même nom que le fichier source avec l'extension class, donc ici PremProg.class.

Exécutez cette classe « Run – Run As – Java Application ». Le processeur Java exécute alors la méthode main définie dans la classe PremProg :

```
public class PremProg {
   public static void main(String args[])
   {
      int i, total=0, n=5;
      for(i=1;i<=n;i++)
         total=total+i;
      System.out.println("somme des entiers de 1 a " + n
+ " = "    + total); } }
```

Vous devez voir apparaître le résultat suivant dans l'onglet « console » :

```
somme des entiers de 1 a 5 = 15
```

II.2.Les types primitifs

II.2.1. Les entiers relatifs

a. Les différents types

type	taille	domaine de valeurs
byte	8 bits	-128 à 127
short	16 bits	-32 768 à 32 767
int	32 bits	-2 147 483 648 à 2 147 483 647
long	64 bits	-9 223 372 036 854 775 808 à 9 223 372 036 854 775 807

b. Représentation en mémoire : rappel

Un bit est réservé au signe (0 pour positif et 1 pour négatif).
Les autres bits servent à représenter :

- la valeur absolue du nombre pour les positifs,

  Exemple d'un entier positif représenté sur 8 bits

0	0	0	0	0	0	0	1
Signe	2^6	2^5	2^4	2^3	2^2	2^1	2^0

$1*2^0 + 0*2^1 + 0*2^2 + 0*2^3 + 0*2^4 + 0*2^5 + 0*2^6 = 1$

0	0	0	1	0	1	1	0

$2^1 + 2^2 + 2^4 = 22$

0	1	1	1	1	1	1	1

$2^7 - 1 = 127$ *(Valeur maximale)*

- ce que l'on nomme le "complément à deux" pour les négatifs (tous les bits sont inversés sauf celui réservé au signe et une unité et ajouté au résultat).

  Exemple d'un entier négatif représenté sur 8 bits

1	1	1	1	1	1	1	1
↓							
1	0	0	0	0	0	0	0

$-1*((0*2^0 + 0*2^1 + 0*2^2 + 0*2^3 + 0*2^4 + 0*2^5 + 0*2^6) + 1)$
$= -1 * (0 + 1) = -1$

1	0	0	1	0	1	1	0
↓							
1	1	1	0	1	0	0	1

$-1 * ((2^0 + 2^3 + 2^5 + 2^6) + 1) = -106$

1	0	0	0	0	0	0	0
↓							
1	1	1	1	1	1	1	1

$-1 * ((2^7 - 1) + 1) = -128$ *(Valeur minimale)*

Le nombre 0 est codé par : | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

II.2.2. Les types flottants

type	**taille**	**précision** *(chiffres significatifs)*	**domaine de valeurs** *(pour les valeurs positives)*
float	32 bits	7	1.40239846E-45 à 3.40282347E38
double	64 bits	15	4.9406564584124654E-324 à 1.797693134862316 E308

II.2.3. Le type caractère

Les caractères sont de type *char* et sont représentés en mémoire sur 16 bits en utilisant le code universel Unicode.

Notations des constantes de type caractère : classique entre apostrophes (Exemple : 'a', 'b', 'é')

II.2.4. Le type booléen

Le type *boolean* permet de représenter une valeur logique de type vrai/faux. Les deux constantes du type *boolean* se notent *true* et *false*.

II.2.5. Les variables de type primitif

Une **définition de variable** précise le type, l'identificateur et éventuellement une ou plusieurs valeurs d'initialisation

type identificateur [= valeur(s) d'initialisation]

Un **identificateur** est une suite de caractères formés avec des lettres, des chiffres ainsi que les caractères _ (souligné) et $. Le premier caractère ne peut pas être un chiffre. Les minuscules sont distinguées des majuscules.

La définition d'une variable peut apparaître à n'importe quel emplacement du programme. Mais, **toute variable doit être initialisée avant d'être utilisée**.

Exemple

```
int n=10,m,l;
int p=2*n;                          //ok
l=30;                               //ok
System.out.println("m = " + m);
  //erreur de compilation : la valeur de m n'est pas définie
```

II.2.6. Instructions, portée d'une variable

Une **instruction** en Java est :
une **instruction simple** toujours terminée par un point-virgule et pouvant être librement écrite sur plusieurs lignes ;
une **instruction composée**, encore appelée **bloc d'instructions**, qui est une suite d'instructions placées entre accolades.

La **portée** d'une variable va de la ligne de sa définition jusqu'à la fin du bloc dans lequel elle est définie.
Exemple

```
{
    int n=10;
    …
    {
        int p=20;
        float n=3.5;
            //erreur de compilation : définition de n invalide.
        …
    }
    System.out.println("p = " + p);
      //erreur de compilation : p inconnu
}
```

II.2.7. Les constantes

Le mot clé **final** permet de déclarer que la valeur d'une variable ne doit pas être modifiée pendant l'exécution du programme.
Exemple

```
final int n=10,m;
n++;
  //erreur de compilation : n a été déclaré final
m=n*n;
  //ok bien que la valeur de m ne soit connue qu'à l'exécution
int p=2*m;
```

II.3.Les opérateurs et les conversions

II.3.1. Les opérateurs arithmétiques

Les types de base *byte*, *short*, *int*, *long*, *float*, *double* et *char* admettent comme opérateurs : **+ - * /** dont les priorités et l'associativité sont celles des autres langages.

- *Le quotient de deux entiers est un entier (5/2=2), alors que le quotient de deux flottants est un flottant (5.0/2.0=2.5).*

L'opérateur **modulo** noté **%** peut porter sur des entiers ou des flottants. Il fournit le reste de la division entière de son premier opérande par son second.
Exemple : 11%4 = 3; 12.5%3.5 = 3

- *Le dépassement de capacité n'est jamais détecté et conduit à des résultats erronés.*

- *La division par 0 pour les entiers conduit à une erreur d'exécution. Par contre, aucune opération sur les flottants ne conduit à un arrêt d'exécution.*

II.3.2. Les conversions implicites de types

Il existe deux conversions implicites de types : les conversions d'ajustement de type et les promotions numériques.

Les **conversions d'ajustement de type** (*int* → *long* → *float* → *double*) sont automatiquement mises en œuvre par le compilateur, à condition qu'il n'y ait pas rétrécissement du domaine.

Exemple

```
int n; long p; float x;
n*p+x;
```

conversion d'ajustement de n en *long*, multiplication par p ;
le résultat de n*p est de type *long*, conversion d'ajustement en *float* pour être additionné à x ;
le résultat final est de type *float*.

Les **promotions numériques** convertissent automatiquement toutes valeurs de type *byte*, *short* ou *char* apparaissant dans une opération arithmétique en *int*.

Exemple

```
short n,p; float x;
n*p+x;
```

promotions numériques de n et de p en *int* et multiplication ;
conversion d'ajustement du résultat de n*p en *float* pour être additionné à x ;
le résultat final est de type *float*.

II.3.3. L'opérateur d'affectation

L'affectation renvoie la valeur de la variable affectée.

Cet opérateur possède une associativité de droite à gauche. On peut ainsi écrire des affectations en cascade.
Exemple : i=j=k=5 ;

Les **conversions implicites par affectation** sont se font suivant l'une des deux hiérarchies suivantes :

byte → short → int → long → float → double
char → int → long → float → double

Vous pouvez **affecter** n'importe quelle **expression constante entière** à une variable de type *byte*, *short* ou *char*, à condition que sa valeur soit représentable dans le type voulu.
Exemple

```
final int N = 50 ;
short p = 10 ;
char c = 2*N + 3 ;      //ok : c est le caractère de code 103
byte b = 10*N;          //erreur de compilation : 500 est
                        supérieur à la capacité du type byte
```

II.3.4. Les autres opérateurs

Opérateurs relationnels <, <=, >, >=, == et != dont le résultat est de type *boolean*.

✵ *Pour les caractères, on a :*
'0'<'1'<…<'9'<'A'<'B'<…<'Z'<'a'<'b'…<'z'

Opérateurs logiques dont voici la liste classée par priorité décroissante. Le résultat est de type *boolean*.

Opérateurs	Signification	Valeur de vérité	Remarques
!	non	!cond : vrai si la condition est fausse	
&	et	cond1 & cond2 : vrai si les deux conditions sont vraies	*les deux opérandes cond1 et cond2 sont évalués.*
|	ou	cond1 | cond2 : vrai si **au moins une** des deux conditions est vraie	*les deux opérandes cond1 et cond2 sont évalués.*
^	ou exclusif	cond1 ^ cond2 : vrai si **une et une seule** des deux conditions est vraie	*les deux opérandes cond1 et cond2 sont évalués.*
&&	et conditionnel	comme &	*l'évaluation de la condition est finie dès qu'elle devient fausse.*
||	ou conditionnel	comme |	*l'évaluation de la condition est finie dès qu'elle devient vraie.*

Opérateur ternaire ? : qui a le même comportement qu'en C/C++.

Exemple

```
int a=10, b=5, max ;
max = a>b ? a : b ;        //max vaut 10
```

Opérateurs d'incrémentation et de décrémentation

Les opérateurs ++ et -- peuvent être postfixés ou préfixés.

Exemple

```
int i = 3 ;
int j = i++ ;          //j vaut 3 et i vaut 4
int k = ++i ;          //k et i valent 5
```

💥 *Les opérateurs d'incrémentation n'appliquent pas de conversion à leur opérande.*

Exemple

```
byte i ;
i=i+1 ;        //erreur de compilation : i+1 de type int
               ne peut pas être affecté à un byte
i++ ;          //ok
```

II.3.5. Les conversions explicites de types : l'opérateur cast

Le programmeur peut forcer la conversion d'une expression quelconque dans un type de son choix, à l'aide de l'opérateur **cast**.

Exemple

```
short x=5, y=15 ;
x = (short) (x+y);        //ok grâce au cast
```

L'opérateur **cast** a une grande priorité.

Exemple

```
short x=5, y ;
y = (short) x+2;          //erreur de compilation :
```

conversion de x en *short*, promotion numérique de x en *int* ; le résultat de l'addition est de type *int* et ne peut pas être affecté à un *short*.

💥*Le résultat d'une conversion explicite par un **cast** peut conduire à un résultat fantaisiste.*

Exemple

```
int n=500 ;
byte b=(byte) n ;  //légal mais b aura pour valeur -24,
                   //500 étant supérieur à la capacité du
                   //type byte
```

II.3.6. Récapitulatif des priorités des opérateurs

Liste des opérateurs classés par ordre de priorité décroissante et accompagnés de leur mode d'associativité.

Opérateurs	**Associativité**
. [] ()	gauche à droite
++ -- ! cast new	droite à gauche
* / %	gauche à droite
+ -	gauche à droite
< <= > >= instanceof	gauche à droite
== !=	gauche à droite
&	gauche à droite
|	gauche à droite
&&	gauche à droite
||	gauche à droite
? :	gauche à droite
=	droite à gauche

L'opérateur . permet de faire référence à un champ ou à une méthode d'un objet.
L'opérateur [] permet de faire référence à un élément d'un tableau.
L'opérateur () permet d'appeler une méthode.
L'opérateur instanceof permet de tester si un objet est l'instance d'une classe donnée.

II.4.Les instructions de contrôle

II.4.1. Enoncés conditionnels

if (*condition***)**
instruction_1
[**else**
instruction_2]

condition est une expression booléenne.
instruction_1 et *instruction_2* sont des instructions quelconques.
Les instructions **if** peuvent être imbriquées.

switch (*expression***)**
{ case *constante_1* **:** [*suite_d'instructions_1*]
case *constante_2* **:** [*suite_d'instructions_2*]
…
case *constante_n* **:** [*suite_d'instructions_n*]
[**default :** *suite_d'instructions*]
}

expression est une expression de l'un des types *byte*, *short*, *char* ou *int*.
constante_i est une expression constante d'un type compatible par affectation avec le type de *expression*.
suite_d'instructions_i est une séquence d'instructions quelconques.
Exemple

```
char c ;
switch (c)
 { case 48 :     System.out.println(c) ; break ;
   case 'a' :    System.out.println("c est la lettre a") ;
                 break ;
 }
```

II.4.2. Enoncés itératifs

while (*condition***)** *instruction*

do *instruction* **while (***condition***) ;**

for ([*initialisation*] ; [*condition*] ; [*incrémentations*] **)**
instruction

condition est une expression booléenne.
instruction est une instruction quelconque.
initialisation est une déclaration ou une suite d'expressions quelconques séparées par des virgules.
incrémentations sont des suites d'expressions quelconques séparées par des virgules.

Contrairement à la boucle **while**, la boucle **do … while** est toujours parcourue au moins une fois.

Exemple

```
for (int i=1, j=3 ; i<=5 ; i++, j+=i)
        System.out.println("i = " + i + " et j = " + j) ;
```

II.4.3. Les instructions de branchement inconditionnel break et continue

L'instruction **break** permet de sortir d'un **switch** ou d'un énoncé itératif.

Exemple

```
int total = 0 ;
for (int i=1 ; i<=10 ; i++)
{
        total += i ;
        if (total >= 10) break ;
}
```

L'instruction **continue** permet de passer prématurément à l'itération suivante.

Exemple

```
int total_impair = 0 ;
for (int i=1 ; i<=10 ; i++)
{
        if (i%2 == 0) continue ;
        total_impair += i ;
}
```

TD2. Un deuxième programme

Ecrire un programme qui permet de calculer la factorielle d'un nombre.

Créez un nouveau projet qui contient une classe pour le programme principal.

III. Les classes et les objets

III.1. Une première classe *Personne*

```
public class Personne {
//Définition des champs de la classe Personne
    //Le mot clé private précise que les champs nom et
    departement ne sont accessibles qu'à l'intérieur de la
    classe Personne.
     private String nom ;
     private byte departement ;
       //département de résidence qui est un entier compris
       entre 1 et 95

//Définition des constructeurs de la classe Personne
    //Définition d'un constructeur qui construit un objet de type
    Personne dont le nom est une chaîne de caractères en
    majuscules et dont le département est inconnu.
     public Personne (String nomp) {
            nom = nomp.toUpperCase() ;
            departement = 0 ;
     }

//Définition des méthodes de la classe Personne
    //Le mot clé public précise que les méthodes sont
    accessibles partout où la classe Personne est accessible.
     public void setDepartement (int dept) {
            departement = (byte) dept;
     }
     public void affiche () {
            System.out.println("Je m'appelle " + nom) ;
            if (departement!=0)
                  System.out.println("J'habite dans le " +
                  departement) ;
     } }
```

III.2. Accès aux membres d'une classe

On appelle **membres** d'une classe les champs et les méthodes de la classe.

Accès public
Les membres d'une classe CCC, définies avec le **modificateur** public, sont accessibles partout où la classe CCC est accessible.

Accès private
Les membres d'une classe CCC, définies avec le **modificateur** private, ne sont accessibles que dans la classe CCC.

D'après la définition de l'accès private, seules les méthodes d'une classe peuvent accéder aux champs privés de cette classe. Cette autorisation concerne tous les objets de la classe et non seulement l'objet courant.

Exemple

```
class A {
   private int n ;
   public void copie(A a) {
      n=a.n ;
         //la méthode copie a accès au champ privé de l'objet a
   }
}
```

III.3. Les méthodes d'une classe

III.3.1. Typologie des méthodes

On distingue trois types de méthodes dans une classe :

les **constructeurs** qui permettent d'initialiser les champs d'un objet ;

les **méthodes d'accès** qui fournissent des informations relatives à l'état d'un objet, c'est à dire aux valeurs de certains de ses champs (généralement privés), sans les modifier ;

Exemple

On peut définir dans la classe Personne la méthode d'accès suivante :

```
public String getNom()
  { return nom ; }
```

les **méthodes d'altération** qui modifient l'état d'un objet, donc les valeurs de certains de ses champs. Les méthodes d'altération comportent souvent des contrôles qui permettent de valider la nouvelle valeur d'un champ.

Exemple

La méthode setDepartement de la classe Personne :

```
public void setDepartement (int dept)
      { departement = (byte) dept; }
```

Les méthodes d'accès et d'altération permettent de gérer l'accessibilité des champs privés d'un objet en lecture seule ou en lecture/écriture.

Il est conseillé d'adopter les préfixes **get** et **set** respectivement pour les méthodes d'accès et d'altération.

III.3.2. Règle d'écriture des méthodes

Une méthode est formé d'un bloc qui constitue son corps précédé d'un en-tête comme suit :

modificateur_d'accès type_de_retour nom_méthode (Type1 argument1, Type2 argument2, …)

Exemple

```
public void setDepartement (int dept)
  { departement = (byte) dept; }
```

Une méthode peut ne fournir aucun résultat, son type de retour est void.
L'argument d'une méthode peut être déclaré avec l'attribut **final**. Dans ce cas, le compilateur s'assure que sa valeur n'est pas modifiée par la méthode.

La **signature** d'une méthode comprend son nom et la liste et le type de ses arguments.

Un **constructeur** est une méthode particulière, sans valeurs de retour, portant le même nom que la classe.

Exemple

```
public Personne (String nomp) {
   nom = nomp.toUpperCase() ;
   departement = 0 ;
}
```

III.3.3. Surdéfinition de méthodes

Plusieurs méthodes peuvent porter le même nom si leur signature diffère, c'est à dire si le nombre et/ou le type de leurs arguments diffèrent. Le compilateur détermine alors la méthode à exécuter en fonction de sa signature.

Exemple

Dans la classe Personne, on peut définir les deux constructeurs suivants :

```
public Personne (String nomp) {
   nom = nomp.toUpperCase() ;
   departement = 0 ;
}
public Personne (String nomp, int dept) {
   nom = nomp.toUpperCase() ;
   setDepartement(dept) ;
}
```

III.4. Exemple d'utilisation de la classe *Personne*

La classe Personne permet d'**instancier des objets de type** Personne.

Pour utiliser la classe Personne, nous définissons une classe MonProgPers contenant une méthode main utilisant la classe Personne. La méthode main représente le point d'entrée du programme : elle est la méthode principale du programme appelée au début de son exécution.

💥 *La méthode* main *pourrait être directement définie dans la classe* Personne, *mais elle aurait alors directement accès aux champs privés de la classe* Personne, *ce qui ne correspond pas aux conditions usuelles d'utilisation d'une classe.*

III.4.1. Définition d'une classe *MonProgPers*

```
public class MonProgPers {
    public static void main(String args[]) {
        Personne p1, p2, p3 ;
            //déclaration de trois variables de type Personne
        p1 = new Personne("Durand") ;
            //ok : création d'un objet de type Personne
            référencé par p1
        p2 = new Personne() ;
            //erreur à la compilation : la classe Personne n'a
            pas de constructeur sans arguments
        p3 = new Personne(2) ;
            //erreur à la compilation : le constructeur de la
            classe Personne a comme argument un objet de
            type String
        System.out.println("Je m'appelle " + p1.nom) ;
            //erreur à la compilation : le champ nom de la
            classe Personne n'est pas accessible à l'extérieur
            de la classe Personne
        p1.affiche() ;
            //affiche :    Je m'appelle DURAND
        p1.setDepartement(75) ;
            //ok : l'objet référencé par p1 appelle la méthode
            setDepartement de la classe Personne avec
            l'argument 75
        p1.affiche() ;
            //affiche :    Je m'appelle DURAND
                           J'habite dans le 75
    }
}
```

III.4.2. Exécution du programme

Lors de l'**exécution d'un fichier** contenant le code compilé d'une classe, la **machine virtuelle** (cf. paragraphe I.2.2.) déclenche l'exécution de la méthode main de cette classe, si cette méthode existe et si elle est publique. Bien entendu, la machine virtuelle doit avoir à sa disposition toutes les classes nécessaires à l'exécution.

Lorsqu'un programme est formé de plusieurs classes, il faut **utiliser un fichier source par classe**.

Exemple

Soit *Personne.java* le fichier contenant le source de la classe Personne. Sa compilation donne naissance au fichier de bytecodes *Personne.class*. Il n'est pas question d'exécuter directement ce fichier puisque la machine virtuelle recherche une méthode main.

Soit *MonProgPers.java* le fichier contenant le source de la classe MonProgPers. Sa compilation nécessite que *Personne.class* existe et que le compilateur y ait accès.

Une fois le fichier *MonProgPers.java* compilé, il ne reste plus qu'à exécuter le fichier *MonProgPers.class* correspondant. L'exécution du fichier *MonProgPers.class* déclenche l'exécution de la méthode main de la classe MonProgPers.

TD3. Spécifier et définir une classe Ville

Une ville est décrite par deux informations :

- son nom ;
- son nombre d'habitants.

Le nom d'un objet de type Ville est toujours en majuscule. Il doit être connu dès l'instanciation de l'objet et ne peut pas varier.
Le nombre d'habitants d'un objet de type Ville est positif. Il peut être inconnu, sa valeur est alors 0. Il peut varier pour une même ville.
Précisez quels sont les champs et les méthodes (constructeurs, méthodes d'accès et méthodes d'altération) de la classe Ville.

Créez un projet TD03. Importez les fichiers Ville et Test du répertoire Java_td_classe. Pour tester la classe Ville, vous pourrez utiliser la classe Test dont la définition est donnée ci-dessous.

```
public class Test {
  public static void main(String args[]) {
    Ville v1 = new Ville("Paris");
    v1.afficher();
    v1.setNbHabitants(2000000);
    v1.afficher();
    Ville v2 = new Ville("Dijon",150000);
    v2.afficher();
    v2.setNbHabitants(-200000);
    v2.afficher();
    }
}
```

A l'exécution, on obtient :

```
v1 : Ville(PARIS, ?)
v1 : Ville(PARIS, 2000000)
v2 : Ville(DIJON, 150000)
Un nombre d'habitants doit etre > 0 !
v2 : Ville(DIJON, 150000)
```

III.5. Variables, valeurs et affectations

Il existe deux types de variables en Java :
les variables de type primitif (cf. paragraphe II.2.5.),

Exemple

```
int n ;
```

les **variables de type classe**.

Exemple

```
Personne p1 ;      //p1 est une variable de type Personne
```

III.5.1. Variables et valeurs

L'**emplacement mémoire** d'une variable de type primitif **contient la valeur** associée à la variable.

Exemple

```
int n ;          n  ?

n=10 ;           n  10
```

L'**emplacement mémoire** d'une variable de type classe **contient une référence à l'objet** associé à la variable.

Exemple

```
Personne p1 = new Personne("Durand") ;
```

Soient une classe CCC et une variable x de type CCC. Par abus de langage, on appellera **objet x** l'objet de type CCC dont la référence est contenue dans x.

III.5.2. Variables et affectations

Une affectation entre deux variables de type primitif porte sur leur valeur.

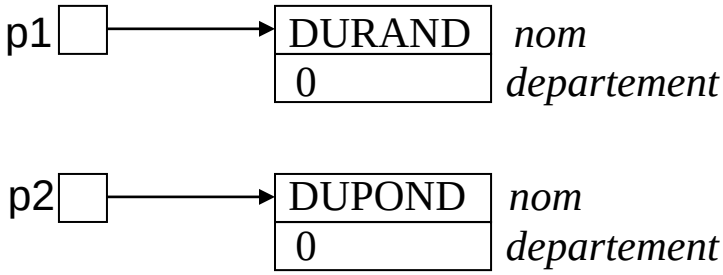
Exemple

```
int x=10, y=20 ;
x = y ;        //x et y valent 20
y+=5 ;         //les valeurs de x et de y sont distinctes : x vaut
               20 et y vaut 25
```

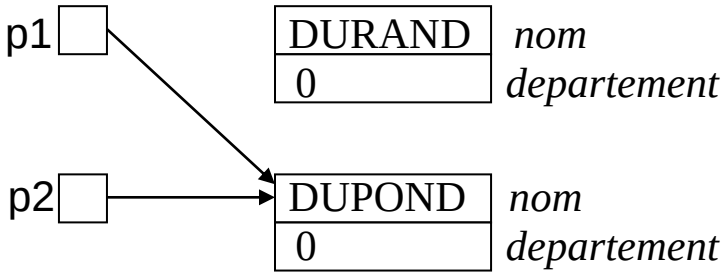
Une affectation entre deux variables de type classe porte sur les références et non sur les objets eux-mêmes.

Exemple

```
Personne p1, p2 ;
p1 = new Personne("Durand") ;
p2 = new Personne("Dupond") ;
```




```
p1 = p2 ;
```



//Les variables p1 *et* p2 *désignent le même objet.*

III.6. Création et initialisation d'un objet

Soit une classe CCC. Pour créer un **objet de type CCC**, il faut :

déclarer une **variable de type CCC** qui réserve un emplacement mémoire pour une référence à l'objet de type CCC ;

appeler l'opérateur unaire **new** qui permet d'allouer un emplacement mémoire pour l'objet de type CCC et fournit sa référence en résultat.

Exemple

```
Personne p1 ;                                   p1  ?
   //déclaration d'une variable de type Personne
p1 = new Personne("Durand") ;
```


```
   //création d'un objet de type Personne référencé par p1
```

La **création d'un objet** entraîne toujours, par ordre chronologique, les opérations suivantes :

une initialisation par défaut de tous les champs de l'objet,

une initialisation explicite lors de la déclaration des champs,

l'exécution des instructions du corps du constructeur.

✸ *On notera que les champs d'un objet peuvent être soit de type primitif, soit de type classe.*

III.6.1. Initialisation par défaut des champs d'un objet

Les champs d'un objet sont toujours initialisés par défaut. Dès qu'un objet est créé, et avant l'appel du constructeur, ses champs sont initialisés à une **valeur par défaut dite "nulle"** ainsi définie :

Type du champ	Valeur par défaut
boolean	false
char	caractère de code nul
entier (*byte*, *short*, *int*, *long*)	0
flottant (*float*, *double*)	0.
objet	**null**

- *Toutes les autres variables (celles qui ne sont pas des champs), qu'elles soient de type primitif ou de type classe, ne sont pas initialisées par défaut et doivent l'être explicitement avant d'être utilisées.*

III.6.2. Initialisation explicite des champs d'un objet

Les champs d'un objet peuvent être initialisés lors de leur déclaration.

Exemple

```
class A {
   private int n = 10 ;
   private int p ;
}
```

En règle générale, il est préférable d'effectuer les initialisations explicites dans le constructeur.

III.6.3. Initialisation par un constructeur

Un constructeur d'une classe CCC est une méthode particulière, sans valeurs de retour, portant le même nom que la classe. Il permet d'automatiser le mécanisme d'**initialisation** d'un objet de type CCC.

Exemple

```
public class Personne {
   private String nom ;
   private byte departement ;
   public Personne (String nomp) {
      nom = nomp.toUpperCase() ;
      departement = 0 ; }
}

Personne p1 = new Personne("Durand") ;
   //Création d'un objet de type Personne référencé par p1.
   //Le constructeur de la classe Personne initialise les
   champs nom et departement de l'objet p1 avec
   respectivement les valeurs DURAND et 0.
```

Une classe peut avoir plusieurs constructeurs ou aucun. Dans ce cas, Java fournit un **constructeur par défaut**. Le constructeur par défaut est sans arguments et initialise chaque champ de l'objet construit avec la valeur par défaut nulle de son type.

Dès qu'une classe possède au moins un constructeur, le constructeur par défaut n'est plus disponible, sauf si la classe contient explicitement un constructeur sans arguments.

Exemple

Si dans la classe Personne, on définit le constructeur :

```
public Personne () {
        nom = null ; departement = 0 ; }
```

On pourra écrire :

```
Personne p2 = new Personne() ;
```

III.7. Le ramasse miettes

L'opérateur **new** permet d'allouer un emplacement mémoire à un objet de type classe et de l'initialiser. En revanche, il n'existe aucun opérateur permettant de détruire un objet dont on n'aurait plus besoin.

Java dispose d'un mécanisme de gestion automatique de la mémoire connu sous le nom de **ramasse miettes**. Lorsqu'il n'existe plus aucune référence sur un objet, le ramasse miettes récupère l'espace occupé par l'objet : le programmeur n'a pas à se soucier de cette opération.

III.8. Echange d'informations avec les méthodes

En Java, la **transmission** d'un argument à une méthode et celle de son résultat ont toujours lieu **par valeur**.

Lors d'un appel de méthode, les arguments sont transmis par recopie de leur valeur.

Le résultat d'une méthode est une copie de la valeur fournie par la méthode.

On appelle **arguments muets** les arguments figurant dans l'en-tête de la définition d'une méthode, et **arguments effectifs** les arguments fournis lors de l'appel de la méthode.

III.8.1. Transmission par valeur pour les types primitifs

Une méthode ne peut pas modifier la valeur d'un argument effectif de type primitif. En effet, la méthode reçoit une copie de la valeur de l'argument effectif et toute modification effectuées sur cette copie n'a aucune incidence sur la valeur de l'argument effectif.

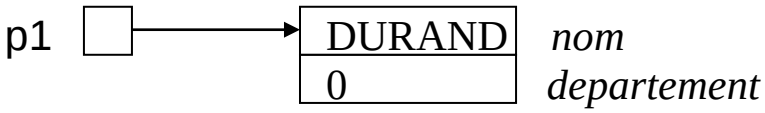
III.8.2. Transmission par référence pour les objets

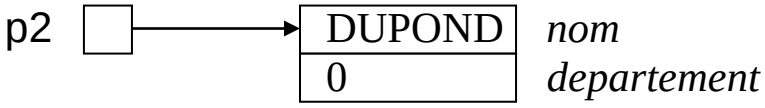
Une méthode peut modifier l'objet référencé par un argument effectif de type classe. En effet, la méthode reçoit une copie de la référence à un objet, puisque l'argument contient une référence à un objet. La méthode peut donc modifier l'objet concerné qui, quant à lui, n'a pas été recopié.

Exemple

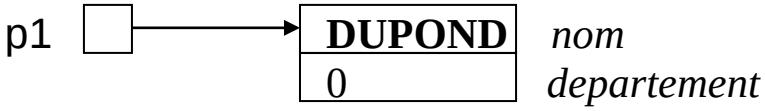
Soit la méthode permute définie dans la classe Personne :

```
public void permute (Personne p) {
   Personne x = new Personne("temp") ;
      //l'objet référencé par x est un objet local à la méthode
      permute qui sert à effectuer l'échange
   x.nom = p.nom ; x.departement = p.departement;
   p.nom = nom ; p.departement = departement;
   nom = x.nom ; departement = x.departement;
}

Personne p1 = new Personne("Durand") ;
Personne p2 = new Personne("Dupond") ;
```






```
p1.permute(p2) ;
```



//Ce ne sont pas les références contenues dans les variables p1 *et* p2 *qui ont changé, mais seulement les valeurs des objets correspondants.*

III.8.3. Autoréférence : le mot clé this

Utilisé dans un constructeur, le mot clé **this** désigne l'objet qui est construit.

Exemple

```
public Personne (String nom, int dept) {
  this.nom = nom.toUpperCase() ;
    //this.nom désigne le champ nom de l'objet de type Personne qui est construit. Le mot clé this permet de distinguer le champ de l'objet de l'argument du constructeur.
  setDepartement(dept) ; }
```

Utilisé dans une méthode autre qu'un constructeur, le mot clé **this** désigne l'objet qui a appelé la méthode.

Exemple

```
public void affiche () {
        System.out.println("Je m'appelle " + nom) ;
        if (departement!=0)
              System.out.println("J'habite dans le " + departement) ;
}
```

Au sein d'un constructeur, il est possible d'en appeler un autre de la même classe (et portant alors sur l'objet courant). Pour cela, on fait appel au mot clé **this** qu'on utilise cette fois comme un nom de méthode. L'appel **this(…)** doit alors être la première instruction du constructeur.

Exemple

Dans la classe Personne, on peut définir les deux constructeurs suivants :

```
public Personne (String nom, int dept) {
  this.nom = nom.toUpperCase() ;
  setDepartement(dept) ; }
public Personne (String nom) {
  this(nom, 0); }
```

TD4. Compléter la classe Ville par des méthodes de copie

Compléter la classe Ville en définissant :

- une méthode qui permet de copier les champs de la ville passée en paramètre dans la ville qui a appelé la méthode ;

- une méthode qui permet de créer une nouvelle ville par copie de la ville qui a appelé la méthode.

Compléter la classe Ville du TD3.

III.9. Champs et méthodes de classe

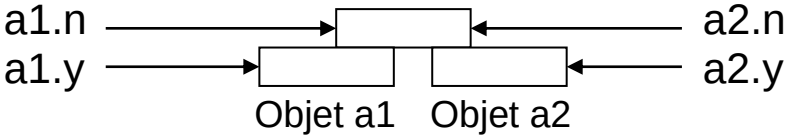
On peut souhaiter disposer de données globales partagées par toutes les instances d'une même classe.

III.9.1. Champs de classe

Un champ qui n'existe qu'en un seul exemplaire pour tous les objets d'une même classe est appelé un **champ de classe**. Un champ de classe est défini avec le modificateur **static**.

Exemple

```
class A {
   static int n ;
   float y ;
}

A a1 = new A() ;
A a2 = new A() ;
```



//Les notations a1.n *et* a2.n *désignent le même champ. Il est possible et même préférable de s'y référer en le nommant* A.n.

III.9.2. Initialisation des champs de classe

Alors que l'initialisation d'un champ usuel est faite à la création d'un objet de la classe, celle d'un champ de classe doit être faite avant la première utilisation de la classe (cet instant pouvant coïncider avec la création d'un objet). **L'initialisation d'un champ de classe** se limite donc à une initialisation par défaut ou éventuellement explicite.

Exemple

```
class A {
   private static int n = 10 ;
   private static int p ;
   …
}

A a ;
   //aucun objet de type A n'est encore créé, mais les champs de classe de A sont initialisés : n explicitement à 10 et p implicitement à 0
```

💥 *Un constructeur, comme d'ailleurs toute méthode, peut très bien modifier la valeur d'un champ de classe, mais il ne s'agit alors plus d'une initialisation, c'est à dire d'une opération accompagnant la création du champ.*

Un champ de classe déclaré avec l'attribut **final** doit obligatoirement être initialisé au moment de sa déclaration.

III.9.3. Méthodes de classe

Une **méthode de classe** représente un comportement associé à la classe elle-même et peut être appelée indépendamment de tout objet de la classe. Une méthode de classe est définie avec le modificateur **static** .

L'appel d'une méthode de classe ne nécessite plus que le nom de la classe correspondante.

Une méthode de classe permet notamment d'accéder aux champs de classe privés de la classe correspondante. Mais, une méthode de classe ne peut pas accéder à des champs "usuels" (champs qui ne sont pas des champs de classe), puisque, par nature, elle n'est liée à aucun objet en particulier.

Exemple

```
class A {
   private static int n ;        //champ de classe
   private float y ;             //champ usuel
   public static void f()        //méthode de classe
   {           //ici on peut accéder au champ de classe n,
      …        mais on ne peut pas accéder au champ usuel y
   }
}

A a = new A() ;
A.f() ;        //appel de la méthode de classe f de la classe A
a.f() ;        //reste autorisé, mais déconseillé
```

La méthode main qui est la méthode principale appelée au début de l'exécution d'une application autonome est une méthode de classe.

III.9.4. Exemple d'utilisation

Les méthodes de classe s'avèrent pratiques pour permettre à différents objets d'une classe de disposer d'informations collectives.

Les méthodes de classe peuvent également fournir des services n'ayant de signification que pour la classe elle-même. Ce serait par exemple le cas d'une méthode fournissant l'identification d'une classe (nom de classe, numéro d'identification…)

Enfin, on peut utiliser des méthodes de classe pour regrouper au sein d'une classe des fonctionnalités ayant un point commun et n'étant pas liées à un objet en particulier. C'est le cas, par exemple, de la classe Math qui contient des méthodes de classe telles que sqrt, sin, cos.

TD5. Compter le nombre d'objets Ville

Compléter la classe Ville en définissant le champ de classe et la méthode de classe permettant de compter le nombre d'objets de la classe Ville créés.

Compléter la classe Ville du TD4.

III.10. Les paquetages

La notion de **paquetage** correspond à un **regroupement logique** sous un identificateur commun **d'un ensemble de classes**. Elle est proche de la notion de bibliothèque que l'on rencontre dans d'autres langages.

III.10.1. Attribution d'une classe à un paquetage

L'attribution d'un nom de paquetage se fait au niveau du fichier source ; toutes les classes d'un même fichier source appartiendront donc toujours à un même paquetage. Pour ce faire, on place en début de fichier une instruction de la forme :

package *nom_paquetage*

En l'absence d'instruction **package** dans un fichier source, le compilateur considère que les classes correspondantes appartiennent au **paquetage par défaut**, qui est unique pour une implémentation donnée.

III.10.2. Utilisation d'une classe d'un paquetage

Lorsque dans un programme, vous faites référence à une classe, le compilateur la recherche dans le paquetage par défaut. Pour utiliser une classe appartenant à un autre paquetage, il est nécessaire de fournir au compilateur l'information lui permettant d'accéder à cette classe. Pour ce faire, on peut :

citer le nom du paquetage avec le nom de la classe ;

Exemple

La classe Date appartient au paquetage java.util, on peut l'utiliser simplement comme suit :

```
java.util.Date uneDate = new java.util.Date() ;
```

utiliser une instruction **import** en y citant une classe particulière d'un paquetage ;

Exemple

```
import java.util.Date ;
Date uneDate = new Date() ;
```

utiliser une instruction **import** en y citant tout un paquetage.

Exemple

```
import java.util.* ;
  //toutes les classes du paquetage java.util peuvent être utilisées en omettant le nom du paquetage correspondant
Date uneDate = new Date() ;
```

III.10.3. Les paquetages standard

Les nombreuses classes standard avec lesquelles Java est fourni sont structurées en paquetage.
Exemple : les paquetages java.util, java.net, java.awt qui doivent être importés explicitement.

Il existe un paquetage particulier nommé java.lang qui est automatiquement importé par le compilateur. Il contient notamment les classes standard System et Math.

- *La classe* System *permet d'accéder aux ressources indépendantes de la plate-forme du système. Elle déclare notamment trois variables, les variables* in, out *et* err, *qui permettent d'avoir une interaction avec le système. La variable* in *représente le flux d'entrée standard du système, alors que la variable* out *représente le flux de sortie standard. La variable* err *est le flux d'erreur standard.*

III.10.4. Les librairies externes

On peut soit même créer ses propres librairies externes. Elles sont généralement fournies sous forme d'un ensemble de fichiers compressés : zip ou jar

III.10.5. Localisation des paquetages : la variable d'environnement CLASSPATH

La plupart des environnements impose des contraintes quant à la localisation des fichiers correspondant à un paquetage. En particulier, un paquetage de nom X.Y.Z se trouvera toujours intégralement dans un sous répertoire de nom X.Y.Z, les niveaux supérieurs étant quelconques.

Avec le JDK (Java Development Kit) de Sun, la recherche d'un paquetage, y compris celle du paquetage courant, se fait à partir des répertoires déclarés dans la variable d'environnement CLASSPATH.

Exemple

```
CLASSPATH = . ; c:\Java\JDK\Lib\classes.zip ;
                c:\javalocal\monpackage ;
```

. désigne le répertoire courant.

c:\Java\JDK\Lib\classes.zip contient les différents paquetages de la librairie standard Java.

c:\javalocal\monpackage correspond aux paquetages personnels.

On peut définir la variable d'environnement CLASSPATH au niveau du système d'exploitation et/ou de l'environnement de développement.

Dans un paquetage, les classes sont compilées (fichiers *.class).

III.10.6. Paquetages et droit d'accès

Chaque classe dispose d'un droit d'accès qui permet de décider quelles sont les autres classes qui peuvent l'utiliser. Il est défini par la présence ou l'absence du modificateur **public** :

avec le modificateur public, la classe est accessible à toutes les autres classes (moyennant éventuellement le recours à une instruction **import**) ;

sans le modificateur public, la classe n'est accessible qu'aux classes du même paquetage.

💥 *Tant que l'on travaille avec le paquetage par défaut, l'absence du modificateur **public** n'a pas d'importance.*

Nous avons déjà vu qu'on pouvait utiliser pour un **membre** d'une classe l'un des modificateurs **public** ou **private** (cf. paragraphe III.2.) :

avec **public**, le membre est accessible depuis l'extérieur de la classe ;

avec **private**, le membre n'est accessible qu'à l'intérieur de la classe.

En l'absence de modificateurs public ou private, l'accès aux membres est limité aux classes du même paquetage.

TD6. Modifier le paquetage de la classe Ville

Modifier le paquetage de la classe Ville et regarder ce qui se passe.

Compléter la classe Ville du TD5.

III.11. Les classes enveloppes pour les types primitifs

Il existe dans le paquetage java.lang des classes nommées **Boolean**, **Byte**, **Character**, **Short**, **Integer**, **Long**, **Float** et **Double**, destinées à manipuler des valeurs d'un type primitif en les encapsulant dans une classe.

Toutes ces classes disposent d'un constructeur recevant un argument de type primitif.

Exemple

```
Integer objint = new Integer(5) ;
    //objint contient la référence à un objet de type Integer
    encapsulant la valeur 5
```

Toutes ces classes disposent aussi d'une méthode de la forme xxxValue, où xxx représente le nom du type primitif, qui permet de retrouver la valeur dans le type primitif correspondant.

Exemple

```
Integer objint = new Integer(5) ;
int n = objint.intValue() ;          //n contient la valeur 5

Float objfloat = new Float(3.5) ;
float x = objfloat.floatValue() ;    //x contient la valeur 3.5
```

IV. L'héritage

IV.1. Une classe *Employe*

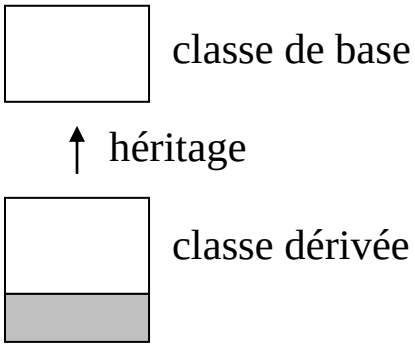
Un employé est une personne qui travaille dans une société. La classe Employe hérite de la classe Personne.

```
class Personne {
   private String nom ;
   private byte departement ;

   public void setNom (String nom) {
      this.nom = nom.toUpperCase() ; }
   public void setDepartement (int departement) {
      this.departement = (byte) departement; }
   public Personne (String lenom, int dept) {
      setNom(lenom) ;
      if(dept != 0) setDepartement(dept) ; }
   public Personne () {                    // constructeur par défaut
      nom = null ; departement = 0 ; }
   public void affiche () {
      if (nom != null)
         System.out.println("Je m'appelle " + nom) ;
      if (departement != 0)
         System.out.println("J'habite dans le " +
         departement) ; }
   }

   class Employe extends Personne {
   //Le mot clé extends indique au compilateur que la classe
   Employe est une classe dérivée de la classe Personne.
      private string societe ;
```

```
    public void setSociete (String societe) {
      this.societe = societe.toUpperCase() ; }
    public Employe (String entreprise) {
      setSociete(entreprise) ; }
  }

public class MonProgPersEmp {
  public static void main(String args[]) {
    Employe e = new Employe("SocieteX") ;
      //l'objet e de type Employe a accès aux membres
      publics de la classe Personne
    e.setNom("Durand");
    e.setDepartement(75) ;
    e.affiche() ;
      //affiche :     Je m'appelle DURAND
                      J'habite dans le 75
  }
}
```

IV.2. La notion d'héritage

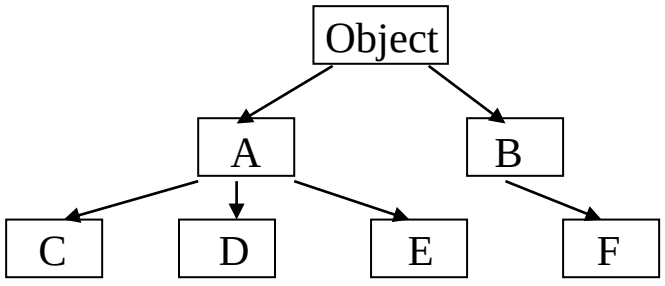
L'héritage permet de définir une nouvelle classe, dite **classe dérivée**, à partir d'une classe existante, dite **classe de base**. Cette nouvelle classe hérite des fonctionnalités de la classe de base (champs et méthodes) qu'elle pourra modifier ou compléter à volonté, sans remettre en cause la classe de base.



A la définition d'une classe dérivée, on utilise le mot clé **extends** pour désigner sa classe de base :

class B **extends** A {…}

Une classe dérivée ne peut hériter que d'une seule classe de base. On parle d'**héritage simple**.
D'une même classe peuvent être dérivées plusieurs classes différentes.
Les notions de classe de base et de classe dérivée sont relatives puisqu'une classe dérivée peut, à son tour, servir de classe de base pour une autre classe.
La hiérarchie de classe est une arborescence de racine unique.



La classe **Object** est la **super-classe** de toutes les classes Java. On dit que D est une classe **descendante** de Object et inversement que Object est une classe **ascendante** de D.

IV.3. Accès d'une classe dérivée aux membres de sa classe de base

IV.3.1. Accès public et privé

Les membres publics d'une classe de base restent des membres publics pour la classe dérivée.

Un objet d'une classe dérivée accède aux membres publics de sa classe de base.

Exemple : Nous avons vu dans l'exemple du paragraphe IV.1. que l'objet e de type Employe a accès aux membres publics de la classe Personne.

Une méthode d'une classe dérivée accèdent aux membres publics de sa classe de base.

Exemple

Soit la méthode afficheE de la classe Employe :

```
public void afficheE () {
   affiche() ;
   System.out.println("Je travaille dans la societe "
   + societe) ; }

Employe e = new Employe("SocieteX") ;
e.setNom("Durand");
e.setDepartement(75) ;
e. afficheE() ;
   //affiche :  Je m'appelle DURAND
                J'habite dans le 75
                Je travaille dans la societe SOCIETEX
```

Une méthode d'une classe dérivée n'a pas accès aux membres privés de sa classe de base.

IV.3.2. Le modificateur d'accès protected

Nous avons vu qu'il existe différents droits d'accès aux membres d'une classe : public (modificateur **public**), privé (modificateur **private**) et de paquetage (aucune mention). Il existe un quatrième droit d'accès aux membres d'une classe dit protégé (modificateur **protected**) .

Un membre déclaré **protected** est accessible aux classes du même paquetage, ainsi qu'à ses classes dérivées (qu'elles appartiennent ou non au même paquetage).

Exemple

```
class A {
   protected int n ;
}
class B extends A {
      …        //accès au champ n de la classe A
      //Attention : toute méthode de la classe B n'a accès au
      champ n de la classe A que pour des objets de la classe
      B ou d'une classe dérivée de B
      public void f(A a, B b, C c)
      {
        a.n ;        //erreur de compilation : n est considéré
                     comme un champ privé de la classe A
        b.n ;        //ok
        c.n ;        //ok
      }
}
class C extends B {
      …        //accès au champ n de la classe A
}
class D {
      …        //pas d'accès au champ n de la classe A, sauf si
               les classes A et D sont dans le même paquetage
}
```

IV.4. Récapitulatif sur les droits d'accès

Modificateurs d'accès pour les classes

Modificateur	**signification pour une classe**
public	accès toujours possible
aucun (droit de paquetage)	accès possible depuis les classes du même paquetage

Modificateurs d'accès pour les membres d'une classe

Modificateur	**signification pour un membre**
public	accès possible partout où la classe est accessible
aucun (droit de paquetage)	accès possible depuis toutes les classes du même paquetage
protected	accès possible depuis toutes les classes du même paquetage ou depuis les classes dérivées
private	accès restreint à la classe

IV.5. Création et initialisation d'un objet dérivé

En Java, le constructeur de la classe dérivée doit prendre en charge l'intégralité de la construction de l'objet. S'il est nécessaire d'initialiser certains champs de la classe de base et s'ils sont convenablement encapsulés, il faut :

- soit disposer de méthodes d'altérations publiques de la classe de base ;
- soit appeler explicitement un constructeur public de la classe de base à l'aide du mot clé **super**. Cet appel doit alors obligatoirement être la première instruction du constructeur de la classe dérivée.

Exemple

Soit le constructeur de la classe Employe :

```
public Employe (String lenom, String entreprise) {
    super(lenom, 0);    //appel du constructeur à deux arguments de la classe Personne
    setSociete(entreprise) ; }
```

Si la classe de base ne possède aucun constructeur, il est possible d'appeler son constructeur par défaut dans le constructeur de la classe dérivée à l'aide de l'instruction **super()**. Cette instruction peut être omise.

Exemple

Soit le constructeur de la classe Employe :

```
public Employe (String entreprise) {
    setSociete(entreprise) ; }
```

Ce constructeur est équivalent au constructeur suivant :

```
public Employe (String entreprise) {
    super() ;    //appel du constructeur par défaut de la classe Personne
    setSociete(entreprise) ; }
```

Si la classe dérivée ne possède aucun constructeur, Java fournit alors pour cette classe un constructeur par défaut. Ce constructeur par défaut appelle :

- soit le constructeur par défaut de la classe de base si cette dernière ne possède aucun constructeur,
- soit un constructeur sans argument de la classe de base (si la classe de base ne possède pas de constructeur sans argument, alors il y a une erreur de compilation).

TD7. Héritage

La définition proposée pour la classe Ville est donnée ci-dessous. Dans cette définition, la méthode afficher() a été remplacée par la méthode toString() qui renvoie une chaîne représentant l'objet courant.

```
public class Ville {
	private String nom;
	private int nbHabitants;
	private int valideNbHabitants(int nb) {
	    if (nb <= 0) {
	     System.out.println("Un nombre d'habitants doit etre > 0 !");
	     return nbHabitants; }
	    return nb ;
	}
	public Ville(String nomVille) {
	    nom = nomVille.toUpperCase();
	    nbHabitants=0;
	}
	public Ville(String nomVille, int nb) {
	    nom = nomVille.toUpperCase();
	    nbHabitants = valideNbHabitants(nb);
	}
	public String getNom() {
	    return nom;
	}
	public int getNbHabitants() {
	    return nbHabitants;
	}
	    public void setNbHabitants(int nb) {
	    nbHabitants = valideNbHabitants(nb);
	}
	public String toString() {
	    String description = new String("Ville(" + nom + ", ");
	    if (nbHabitants>0)
	      description = description + nbHabitants + ")";
	    else
	      description = description + "?)";
	    return description;
	}
}
```

Une classe Capitale qui hérite de la classe Ville

Par rapport à un objet Ville, un objet Capitale dispose d'une information supplémentaire : son pays, qui sera représenté par un nom en majuscules.
Définissez la classe Capitale comme sous-classe de la classe *Ville*, avec :

- de nouveaux constructeurs, qui appellent les constructeurs de la classe Ville,
- une redéfinition de la méthode toString() de la classe Ville.

Créez un projet TD07. Importez les fichiers Capital, Ville et Test du répertoire Java_td_heritage. Pour tester la classe Capital, vous pourrez utiliser la classe Test dont la définition est donnée ci-dessous.

```
public class Test {
  public static void main(String args[]) {
   Ville v1 = new Ville("Dijon",150000);
   System.out.println("v1 : " + v1);
   Capitale c1 = new Capitale("Paris",2000000,
"France");
   System.out.println("c1 : " + c1);
   c1.setNbHabitants(2100000) ;
   System.out.println("c1 : " + c1);
  }
}
```

A l'exécution, on obtient :

```
v1 : Ville(DIJON, 150000)
c1 : Capitale(FRANCE, Ville(PARIS, 2000000))
c1 : Capitale(FRANCE, Ville(PARIS, 2100000))
```

IV.6. Le polymorphisme

Le polymorphisme permet de manipuler des objets sans en connaître (tout à fait) le type. Il exploite la relation de type "est" introduite par l'héritage.

Un objet d'une classe dérivée peut être considéré comme un objet de sa classe de base. La réciproque est fausse.

Il existe une **conversion implicite** d'une référence à un objet de type classe en une référence à un objet de type ascendant.

Exemple

Un employé est une personne, on peut donc le traiter comme une personne. Par contre, une personne n'est pas un employé.

```
Personne p ;
Employe e ;
p = new Employe("Durand",75,"SocieteX") ;
   //p de type Personne contient la référence à un objet de type Employe
e = new Personne("Dupond",32) ;
   //erreur de compilation
```

IV.7. Redéfinition et surdéfinition de membres

IV.7.1. Redéfinition de méthodes

La **redéfinition de méthode** permet à une classe dérivée de redéfinir une méthode accessible (qui n'est pas privée) de sa classe de base. Les deux méthodes doivent avoir le même nom, la même signature et le même type de valeur de retour. Une méthode redéfinie dans une classe dérivée "masque" la méthode correspondante de la classe de base.

Exemple

```
class Employe extends Personne {
   private string societe ;
   public void setSociete (String societe) {
      this.societe = societe.toUpperCase() ; }
   public Employe (String lenom, int dept, String
   entreprise) {
      super(lenom, dept) ;
      setSociete(entreprise) ; }
   //redéfinition de la méthode affiche de la classe Personne
   public void affiche () {
      super.affiche() ;
        //appel de la méthode affiche de la classe Personne
      System.out.println("Je travaille dans la societe " +
      societe) ; }
}
public class MonProgPersEmp {
   public static void main(String args[]) {
      Employe e = new Employe("Durand",75,"SocieteX") ;
      e. affiche() ;
            //affiche :  Je m'appelle DURAND
                         J'habite dans le 75
                         Je travaille dans la societe SOCIETEX
   }
```

La redéfinition d'une méthode s'applique à une classe et à toutes ses classes descendantes jusqu'à ce qu'éventuellement l'une d'entre elles redéfinisse à nouveau la méthode.

Exemple

```
class A {
   public void f(int n) { … }
   …          //accès à la méthode f de la classe A
}
class B extends A {
      …       //accès à la méthode f de la classe A
}
class C extends B {
   public void f(int n) { … }  //redéfinition de f
   …          //accès à la méthode f de la classe C
}
class D extends C {
      …       //accès à la méthode f de la classe C
}
```

La redéfinition d'une méthode ne doit pas diminuer les droits d'accès à cette méthode. Par contre, elle peut les augmenter (paquetage → public, protected → public, public → public).

Exemple

```
class A {
   private void f(int n) { … }
}
class B extends A {
   public void f(int n) { … }
     //redéfinition de f avec extension des droits d'accès
}
```

Une méthode déclarée **final** ne peut pas être redéfinie dans une classe dérivée.

Une méthode de classe ne peut pas être redéfinie dans une classe dérivée.

IV.7.2. Surdéfinition de méthodes

Une classe dérivée peut **surdéfinir** (cf. paragraphe III.3.3.) une méthode accessible (qui n'est pas privée) d'une classe de base, ou plus généralement d'une classe ascendante.

Lors d'un appel de la forme x.f(…) où x est supposé être une variable de classe CCC, le compilateur détermine, dans la classe CCC ou ses classes ascendantes, la meilleure méthode f à exécuter en fonction de sa signature.

Exemple

```
class A {
   public void f (int n)  { … }
}
class B extends A {
   public void f (float x) { … }
      //surdéfinition de la méthode f pour B
}

A a = new A(…);
B b = new B(…);
int n, float x ;
a.f(n) ;        //appel de la méthode f(int) de la classe A
a.f(x) ;
   //erreur de compilation : la seule méthode acceptable est la
   méthode f(int) de la classe A et on ne peut pas convertir la
   variable x de type float en int.
b.f(n) ;        //appel de la méthode f(int) de la classe A
b.f(x) ;        //appel de la méthode f(float) de la classe B
```

TD8. Héritage : la méthode toString()

La méthode toString() de la classe Ville n'est pas adaptée à l'héritage car, si on l'appelle pour un objet d'une sous-classe XXX de Ville, elle renverra une chaîne commençant par "Ville(", indiquant ainsi que l'objet est une instance de Ville.
A l'exécution, on veut obtenir l'affichage suivant :

```
v1 : Ville(DIJON, 150000)
c1 : Capitale(FRANCE, PARIS, 2000000)
```

Pour être plus générique, la méthode toString() doit faire appel à deux autres méthodes :

- la méthode getNomClasse() pour afficher le nom de la classe. Elle peut être écrite de façon générique dans la classe Ville comme suit :

```
protected String getNomClasse() {
        return getClass().getName();
}
```

- la méthode getDescription() propre à chaque classe.

Définissez la méthode getDescription() pour les classes Ville et Capitale et redéfinissez la méthode toString() en conséquence.

Compléter les classes Ville et Capitale du TD7.

IV.8. Les classes abstraites

Une **classe abstraite** est une classe qui ne permet pas d'instancier des objets. Elle ne peut servir que de classe de base pour une dérivation et se déclare avec le mot clé **abstract**.

Exemple

```
abstract class A { … }
class B extends A { … }

A a ;                 //ok : a est une référence sur un objet de type A ou de type descendant
a = new A(…) ;        //erreur de compilation : on ne peut pas instancier d'objets d'une classe abstraite
a = new B(…) ;        //ok
```

L'intérêt d'une classe abstraite est que l'on peut y placer toutes les fonctionnalités dont on souhaite disposer pour ses classes descendantes :

soit sous forme d'une implémentation complète de méthodes et de champs dont héritera toute classe dérivée ;

soit sous forme d'interface de **méthodes abstraites** dont on est alors sûr qu'elles existeront dans toute classe dérivée instanciable.

Une **méthode abstraite** est une méthode déclarée avec le mot clé **abstract,** dont on ne fournit que la signature et le type de valeur de retour. Une méthode abstraite doit obligatoirement être déclarée **public**.

Exemple

```
abstract class A {
   public void f() { … }              //f est définie dans A
   public abstract void g(int n) ;
      //on ne fournit que l'en-tête de la méthode abstraite g
}
class B extends A {
   public void g(int n) { … }         //g est définie dans B
}
```

Une classe qui contient au moins une méthode abstraite est une classe abstraite, même si elle n'a pas été déclarée avec le mot clé **abstract**.
Une classe dérivée d'une classe abstraite qui ne définit pas toutes les méthodes abstraites de sa classe de base est une classe abstraite.

Exemple

```
abstract class Animal {
   protected String identifiant ;
   protected int poids ;
   public Animal (String identifiant, int poids) {
      this.identifiant = identifiant ; this.poids = poids ; }
   public String toString() {
      return identifiant + poids ; }
   public abstract void definirRation() ;
}

class Bovin extends Animal {
   private String nom ;
   public Bovin(String nom, String identifiant, int poids) {
      super(identifiant, poids) ; this.nom = nom ; }
   public String toString() {
      return nom + super.toString() ; }
   public void definirRation() { … }
      //implémentation de la méthode definirRation pour un
      bovin
}

class Porcin extends Animal {
   public Porcin (String identifiant, int poids) {
      super(identifiant, poids) ; }
   public void definirRation() { … }
      //implémentation de la méthode definirRation pour un
      porcin
}
```

IV.9. Les interfaces

IV.9.1. Définition

Une **interface** définit les en-têtes d'un certain nombre de méthodes, qui sont par essence abstraites et publiques, ainsi que des constantes.

Une interface est déclarée avec le mot clé **interface**. Elle est dotée des mêmes droits d'accès qu'une classe (public ou de paquetage).

Exemple

```
public interface I {
   void f(int n) ;
   void g() ;
      //les méthodes f et g sont par essence des méthodes
      abstraites et publiques ; il n'est pas nécessaire de les
      déclarer avec le mot clé abstract et le modificateur
      public
   static final int MAXI = 100 ;     //constantes
}
```

On peut dériver une interface à partir d'une autre interface.

Exemple

```
public interface I1 {
   void f() ;
   static final int MAXI = 100 ;
}
public interface I2 extends I1 {
   void g() ; }
```

La définition de l'interface I2 est totalement équivalente à :

```
public interface I2 {
   void f() ;
   void g() ;
   static final int MAXI = 100 ;
}
```

IV.9.2. Implémentation d'une interface

Une classe peut implémenter une ou plusieurs interfaces à l'aide du mot clé **implements**.

Les méthodes d'une interface doivent être définies par toutes les classes implémentant l'interface.
Les constantes d'une interface sont accessibles à toutes les classes implémentant l'interface.

Exemple

```
public interface I1 {
   void f() ;
   static final int MAXI = 100 ; }
public interface I2 {
   void g() ; }
class A implements I1, I2 {
   //les méthodes f et g des interfaces I1 et I2 doivent être
   définies dans la classe A
   //dans toutes les méthodes de la classe A on a accès à la
   constante MAXI
}
```

Une classe dérivée peut implémenter une ou plusieurs interfaces.

Exemple

```
public interface I {
   void f(int n) ;
   void h() ;
}
class A implements I1, I2 { … }
class B extends A implements I {
   //les méthodes f et g de l'interfaces I sont soit déjà définies
   dans la classe A, soit elles doivent l'être dans la classe B
}
```

IV.9.3. Interface et polymorphisme

Soit une classe CCC implémentant une interface I. Il existe une **conversion implicite** d'une référence à un objet de type CCC en une référence à un objet de type I.

Exemple

```
public interface Mammifere { … }
public interface Terrestre { … }
public interface Marin {
   public String vitesseDeNage() ;
}
class Tortue implements Terrestre, Marin {
   public String vitesseDeNage() { … }
}
class Baleine implements Mammifere, Marin {
   public String vitesseDeNage() { … }
}
```

La méthode suivante peut prendre indifféremment comme paramètre un objet de type Tortue ou un objet de type Baleine :

```
public void AfficheVitesseDeNage(Marin m) {
   System.out.println("Vitesse de nage : " +
   m.vitesseDeNage()) ; }
```

IV.9.4. Variables de type interface

Une interface ne permet pas d'instancier des objets, mais on peut définir des variables de type interface, qui peuvent référencer des objets d'une classe implémentant une interface.

Exemple

```
public interface I { … }
class A implements I { … }

I i ;                 //ok
i = new I(…) ;        //erreur de compilation
i = new A(…) ;        //ok
```

IV.9.5. Interfaces ou classes abstraites ?

Comme une classe abstraite,

- une interface joue le rôle d'un outil de spécification en forçant toutes les classes implémentant l'interface à définir ses méthodes ;
- une interface peut se dériver ;
- on peut utiliser des variables de type interface.

Mais :

une classe peut implémenter plusieurs interfaces (alors qu'une classe ne pouvait être dérivée que d'une seule classe abstraite) ;

la notion d'interface se superpose à celle de dérivation.

La notion d'interface est donc une notion plus riche que celle de classe abstraite.

TD9. Interface de comparaison

Soit une interface de comparaison ComparerObjet qui permet de tester l'**égalité** de deux objets et de tester si un objet est "**plus grand**" qu'un autre.

```
public interface ComparerObjet {
    public boolean classeCompatibleAvec(ComparerObjet obj);
    //true, si la comparaison est acceptable, sinon sortie du programme
    public boolean egalA(ComparerObjet  obj);
    //true, si this egalA obj
    public boolean plusGrandQue(ComparerObjet  obj);
    //true, si this plusGrandQue obj
}
```

Soit la classe Personne définie comme ci-dessous :

```
public final class Personne { //classe qui ne peut pas être dérivée
    private String nom;
    private int age;
    public Personne(String nomPers, int agePers) {
       nom = nomPers.toUpperCase(); age = agePers ; }
    public String getNom() { return nom; }
    public int getAge() { return age; }
    public void setAge(int agePers) { age = agePers ; }
    public String toString() {
       String description = new String("Personne(" + nom + ", " +
       age + ")"); return description; }
}
```

Une interface ComparerObjet

Implémentez l'interface ComparerObjet pour les classes Personne, Ville et Capitale de telle sorte que les points suivants soient vérifiés :

- on ne peut comparer une personne qu'avec une personne ;
- deux personnes sont égales si elles ont le même nom ;
- une personne p1 est "plus grande" qu'une personne p2 si elles n'ont pas le même nom et que p1 est plus âgée p2 ;
- on peut comparer des villes entre elles, des capitales entre elles et des villes avec des capitales (vous pourrez utiliser l'opérateur instanceof) ;
- une ville (ou capitale) est égale à une autre ville (ou capitale) si elles ont le même nom ;
- une ville x1 (ou capitale) est "plus grande" qu'une ville x2 (ou capitale) si elles n'ont pas le même nom et que le nombre d'habitants de x1 est supérieur au nombre d'habitants de x2.

Créez un projet TD09_a. Importez les fichiers Personne, ComparerObjet, Capital, Ville et Test du répertoire Java_td_interface\etape_1. Pour tester l'interface ComparerObjet, vous pourrez utiliser la classe Test dont la définition est donnée ci-dessous.

```
public class Test {
  public static void main(String args[]) {
    Ville v1 = new Ville("Paris",20000000);
    System.out.println("v1 : " + v1);
    Capitale c1 = new Capitale ("Paris",20000000,
"France");
    System.out.println ("c1 : " + c1);
    Capitale c2 = new Capitale ("Berlin",3500000,
"Allemagne");
    System.out.println ("c2 : " + c2);
    if(v1.egalA(c1))
      System.out.println (v1.getNom() + " et " +
c1.getNom() + " sont une seule et même ville");
    else
```

```
        System.out.println (v1.getNom() + " et " +
c1.getNom() + " sont des villes différentes");
      if(c1.egalA(c2))
        System.out.println (c1.getNom() + " et " +
c2.getNom() + " sont une seule et même ville");
      else
        System.out.println (c1.getNom() + " et " +
c2.getNom() + " sont des villes différentes");
      if(v1.plusGrandQue(c2))
        System.out.println ("la ville " + v1.getNom() + " est
plus grande que la ville " + c2.getNom());
      else
        System.out.println ("la ville " + v1.getNom() + " est
plus petite que la ville " + c2.getNom());
      Personne p1 = new Personne("Dupond",48);
      System.out.println ("p1 : " + p1);
      Personne p2 = new Personne("Durand",39);
      System.out.println ("p2 : " + p2);
      if(p1.egalA(p2))
        System.out.println (p1.getNom() + " et " +
p2.getNom() + " sont une seule et même personne");
      else
        System.out.println (p1.getNom() + " et " +
p2.getNom() + " sont des personnes différentes");
      if(p1.plusGrandQue(p2))
        System.out.println (p1.getNom() + " est plus age
que " + p2.getNom());
      else
        System.out.println (p1.getNom() + " est plus jeune
que " + p2.getNom());
      if(p1.egalA(v1))
        System.out.println (p1.getNom() + " et " +
v1.getNom() + " sont un seul et même objet");
      else
        System.out.println (p1.getNom() + " et " +
v1.getNom() + " sont des objets différents");
      }
   }
```

On obtient à l'exécution :

```
v1 : Ville(PARIS, 2000000)
c1 : Capitale(FRANCE, PARIS, 2000000)
c2 : Capitale(ALLEMAGNE, BERLIN, 3500000)
PARIS et PARIS sont une seule et même ville
PARIS et BERLIN sont des villes différentes
la ville PARIS est plus petite que la ville BERLIN
p1 : Personne(DUPOND, 48)
p2 : Personne(DURAND, 39)
DUPOND et DURAND sont des personnes différentes
DUPOND est plus agé que DURAND
Erreur : comparaison d'un objet Personne avec un objet Ville
```

sortie du programme

Mettre l'interface ComparerObjet dans un fichier jar

Vous allez maintenant conditionner l'interface **ComparerObjet** pour pouvoir l'utiliser facilement dans d'autres projets. Pour cela, vous allez créer un fichier d'archive Java (fichier jar), en procédant comme suit :

- Sélectionnez, dans la fenêtre du navigateur, l'interface ComparerObjet (on peut sélectionner plusieurs classes et interfaces en cliquant sur « CTRL ») et faites apparaître le menu contextuel, dans lequel vous choisirez « Export ».
- « Select an export destination » : choisissez « JAR file », puis bouton « Next ».
- Choisissez d'exporter « class file » et « source file », ne cochez pas « classpath » ni « project ».
- Indiquez de placer le fichier jar, sous le nom ComparerObjet.jar sous le répertoire jar que vous avez obtenu en copiant l'arborescence en début de TD.
- Cliquez sur « Finish ».

Utiliser le fichier JAR dans un projet

Fermez le projet TD09_a.

Créez un projet TD09_b, dont les fichiers se trouvent dans le répertoire Java_td_interface\etape_2.

La compilation de ce projet indique que les références à ComparerObjet n'ont pas pu être résolues. Comprenez pourquoi, en examinant les différentes classes du projet.

Résolvez le problème en procédant comme suit :

- Menu « Project – properties », ou menu contextuel du projet et « properties ».
- Réglage de Java Build path : onglet « Libraries » bouton « Add external JAR's ».
- Choisissez le fichier jar que vous avez créé à l'étape précédente.

Vérifiez que la classe Test.java peut être exécutée.

V. Les tableaux, chaînes, collections

V.1. Les tableaux

Un tableau est une collection indexée d'éléments dont le nombre est fixé à la création. Le premier élément d'un tableau correspond à l'indice 0.

Un tableau est un objet qui peut être référencé par une variable.

V.1.1. Déclaration d'un tableau

La **déclaration** d'une référence à un tableau précise simplement le type des éléments du tableau. Elle peut prendre les deux formes suivantes :

type nom_tableau [] ou type [] nom_tableau

Les éléments d'un tableau peuvent être d'un type primitif ou d'un type classe.

Exemple

```
int t1[], n ;          //t1 est une référence à un tableaux d'entiers et n est un entier
float [] t2, t3 ;      //t2 et t3 sont des références à des tableaux de réels
Personne t4[], p1 ;
    //t4 est une référence à un tableau d'objets de type Personne et p1 est une référence à un objet de type Personne
```

Une déclaration de tableau ne doit pas précisé de dimension. Cette instruction sera rejetée à la compilation :

```
int t[5] ;
```

V.1.2. Création et initialisation d'un tableau

Un tableau a une **taille invariable** qui est définie à la création du tableau.

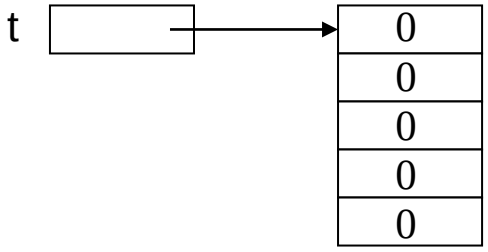
Pour créer un tableau, on peut :

appeler l'opérateur **new** qui permet d'allouer un emplacement mémoire pour le tableau ;

Exemple

```
int t[] = new int[5] ;
    //t est une référence à un tableaux de 5 entiers
```

Cette instruction alloue l'emplacement mémoire nécessaire à un tableau de 5 éléments de type *int* et en place la référence dans t. Les 5 éléments sont initialisés à une valeur par défaut "nulle" (0 pour le type *int*).



utiliser un **initialisateur** au moment de la déclaration du tableau.

Exemple

```
int t[] = {1, 3, 7, 5, 9} ;
    //t est une référence à un tableaux de 5 entiers de valeurs respectives 1, 3, 7, 5 et 9. Cette instruction est équivalente aux instructions suivantes :
      int t[] = new int[5] ;
      t[0]=1; t[1]=3; t[2]=7; t[3]=5; t[4]=9;
```

Le champ **length** permet de connaître le nombre d'éléments d'un tableau de référence donnée.

Exemple

```
int t[] = new int[5] ;
System.out.println("La taille de t est : " + t.length) ;
```

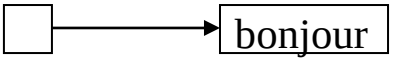
V.2. Les chaînes de caractères

La classe standard String du paquetage java.lang permet de manipuler les chaînes de caractères.

Les **constantes chaînes** telles que "bonjour" ne sont en fait que des objets de type String construits automatiquement par le compilateur.

Exemple

```
String ch ;
  //ch est destiné à contenir une référence à un objet de type
  String
ch = "bonjour" ;
```



La classe String possède **deux constructeurs**, l'un sans argument créant une chaîne vide, l'autre avec un argument de type String qui en crée une copie.

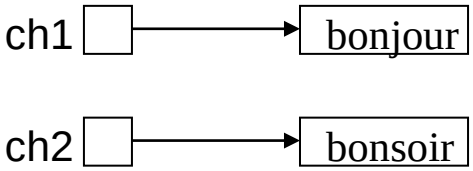
Exemple

```
String ch1 = new String() ;
  //ch1 contient la référence à une chaîne vide
String ch2 = new String("bonjour") ;
  //ch2 contient la référence à une chaîne contenant la suite
  de caractères "bonjour"
String ch3 = new String(ch2) ;
  //ch3 contient la référence à une chaîne, copie de ch2,
  donc contenant la suite de caractères "bonjour"
```

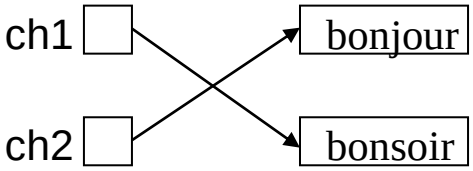
Les objets de type String sont **constants**. Il n'existera donc aucune méthode permettant d'en modifier la valeur.

Exemple

```
String ch1, ch2, ch ;
   //ch1 contient la référence à une chaîne vide
ch1 = "bonjour" ;
ch2 = "bonsoir" ;
```




```
ch = ch1 ;
ch1 = ch2 ;
ch2 = ch;
```



//Les deux objets de type String *n'ont pas été modifiés, mais les variables* ch1 *et* ch2 *l'ont été.*

V.2.1. Quelques méthodes

La méthode **length** fournit la longueur d'une chaîne de caractères.

Exemple

```
String ch= "bonjour" ;
int n = ch.length() ;     //n contient la valeur 7
```

La méthode **charAt** renvoie un caractère de rang donné.

Exemple

```
String ch= "bonjour" ;
char carac1 = ch.charAt(0) ;
   //carac1 contient le caractère 'b'
char carac2 = ch.charAt(2) ;
   //carac2 contient le caractère 'n'
```

La méthode **indexOf** permet de rechercher, à partir du début d'une chaîne ou d'une position donnée, la première occurrence d'un caractère donné ou la première occurrence d'une autre chaîne. Elle fournit la position du caractère (ou du début de la chaîne recherchée) si une correspondance a effectivement été trouvée, la valeur –1 sinon.

Exemple

```
String ch= "voici une chaîne" ;
int n ;
n = ch.indexOf('i') ;          //n vaut 2
n = ch.indexOf("cha") ;        //n vaut 10
n = ch.indexOf('t') ;          //n vaut -1
n = ch.indexOf('n', 9) ;       //n vaut 14
```

V.2.2. Comparaison de chaînes

La méthode **equals** permet de comparer le contenu de deux chaînes.

Exemple

```
String ch1 = "bonjour" ;
String ch2 = "bonsoir" ;
ch1.equals(ch2) ;              //retourne false
ch1.equals("bonjour") ;        //retourne true
```

La méthode **equals** de la classe String est une redéfinition de la méthode **equals** de la classe **Object**, super-classe de toutes les classes Java. La méthode **equals** de la classe **Object** permet de comparer les adresses de deux objets.

Exemple

```
Personne p1 = new Personne("Durand") ;
Personne p2 = new Personne("Durand") ;
p1.equals(p2) ;          //retourne false
```

La méthode **compareTo** permet de comparer le contenu de deux chaînes.

Exemple

```
String ch1 = "bonjour" ;
String ch2 = "bonsoir" ;
ch1.compareTo("bonjour") ;      //retourne 0
ch1.compareTo(ch2) ;        //retourne un nombre négatif
ch2.compareTo(ch1) ;        //retourne un nombre positif
```

V.2.3. Concaténation de chaînes

L'opérateur + est défini lorsque ses deux opérandes sont des chaînes. Il fournit en résultat une nouvelle chaîne formée de la concaténation des deux autres.

Exemple

```
String ch1 = "bonjour " ;
String ch2 = "toto" ;
String ch = ch1 + ch2 ;
  //ch contient la référence à une chaîne contenant la suite de caractères "bonjour toto"
System.out.println(ch1 + ch2) ;
  //création d'un nouvel objet de type String qui contient la chaîne "bonjour toto". Après affichage du résultat, cet objet est candidat au ramasse miettes puisqu'aucune référence ne le désigne plus.
String ch3 = ch1 + "\n" + ch2 ;
System.out.println(ch3) ;
  //affiche : bonjour
              toto
```

Lorsque l'opérateur + possède un opérande de type String, l'autre opérande est automatiquement converti en chaîne.

Exemple

```
int n = 26 ;
String ch = "résultat : " + n + " " + "$";
  //l'entier n est converti en chaîne. ch contient la chaîne "résultat : 26 $"
```

V.2.4. Modification de chaînes

La classe String dispose de quelques méthodes qui, à l'instar de l'opérateur +, créent une nouvelle chaîne obtenue par transformation de la chaîne courante.

La méthode **replace** crée une chaîne en remplaçant toutes les occurrences d'un caractère donné par un autre.

Exemple

```
String ch1 = "bonjour" ;
String ch2 = ch1.replace('o', 'a') ;
  //ch1 n'est pas modifié, ch2 contient la chaîne "banjaur"
```

La méthode **substring** permet de créer une nouvelle chaîne en extrayant de la chaîne courante : soit tous les caractères depuis une position donnée, soit tous les caractères compris entre deux positions données (la première inclue, la seconde exclue).

Exemple

```
String ch1 = "voici une chaîne" ;
String ch2 = ch1.substring(6) ; //ch2 vaut "une chaîne"
String ch3 = ch1.substring(6,9) ;    //ch3 vaut "une"
```

La méthode **toUpperCase** (**toLowerCase**) crée une nouvelle chaîne en remplaçant toutes les minuscules (majuscules) par leur équivalent en majuscules (minuscules).

Exemple

```
String ch1 = "bonjour" ;
String ch2 = ch1. toUpperCase() ;
  //ch2 vaut "BONJOUR"
```

La méthode **trim** crée une nouvelle chaîne en supprimant les éventuels séparateurs de début et de fin (espace, tabulations, fin de ligne).

Exemple

```
String ch1 = "\nvoici une chaîne " ;
String ch2 = ch1. trim() ;    //ch2 vaut "voici une chaîne"
```

V.2.5. Conversion entre chaînes, types primitifs et type classe

La méthode de classe **valueOf** de la classe String permet de convertir un type primitif en une chaîne.

Exemple

```
int n = 26 ;
String ch = String.valueOf(n) ;   //ch vaut "26"
```

Pour réaliser les conversions inverses, on dispose des méthodes suivantes qui sont des méthodes de classe des classes enveloppes associées aux types primitifs (cf. paragraphe III.11.) : Byte.parseByte, Short.parseShort, Integer.parseInt, Long.parseLong , Float.parseFloat, Double.parseDouble. Ces conversions peuvent ne pas aboutir.

Exemple

```
String ch1 = "26" ;
String ch2 = "3.4" ;
int n1 = Integer.parseInt(ch1) ;  //n1 vaut 26
int n2 = Integer.parseInt(ch2) ;  //erreur de compilation
float x = Float.parseFloat(ch2) ; //x vaut 3.4
```

La classe **Object** dispose de la méthode **toString** qui fournit une chaîne contenant :

- le nom de la classe concernée ;
- l'adresse de l'objet en hexadécimal précédé de @.

Exemple

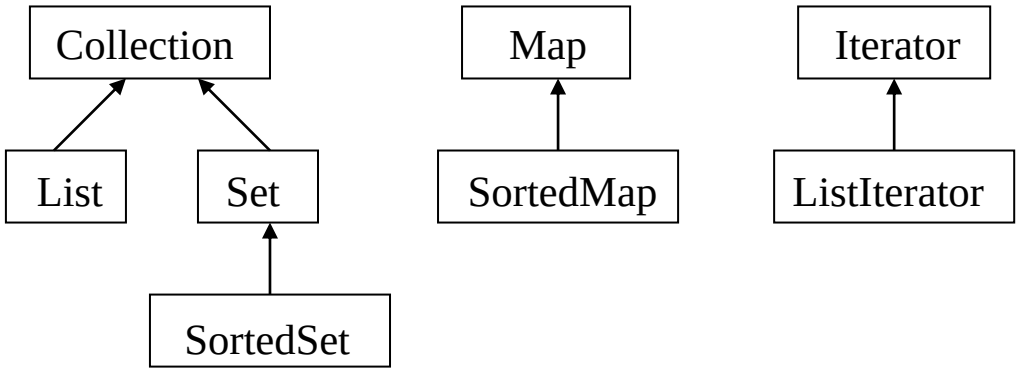
```
Personne p = new Personne("Durand") ;
String ch = p.toString() ; //ch vaut Personne@fc17aedf
```

V.3. Les collections

La bibliothèque de collections Java, du paquetage java.util, permet de gérer des collections d'objets qui s'agrandissent dynamiquement en fonction des besoins. Nous la présentons succinctement.

V.3.1. Les interfaces de collections

Les interfaces de collections Java sont les suivantes :



a. L'interface Collection

L'interface Collection possède trois méthodes essentielles :

boolean add(Object obj)
//ajoute l'élément obj *à la collection, et renvoie true si cet ajout a effectivement modifié la collection*

boolean remove(Object obj)
//supprime l'élément obj *de la collection et renvoie true si cette suppression a effectivement modifié la collection*

Iterator iterator()
//renvoie un itérateur pouvant être utilisé pour parcourir les éléments de la collection

Une **liste** est une collection triée d'objets. L'interface List, dérivée de l'interface Collection, possède quatre méthodes essentielles :

void add(int pos, Object obj)
//ajoute l'élément obj *à la position* pos *dans la liste*

Object get(int pos)
//renvoie l'élément qui se trouve à la position pos *dans la liste*

void remove(int pos)
//supprime l'élément qui se trouve à la position pos *dans la liste*

ListIterator listIterator()
//renvoie un itérateur de liste pouvant être utilisé pour parcourir les éléments de la liste

Un **ensemble** est une collection d'objets distincts. L'interface Set, dérivée de l'interface Collection, possède notamment la méthode add, qui permet d'ajouter un élément à l'ensemble s'il ne s'y trouve pas déjà. .

b. L'interface Map

Une **carte** est une collection de paires clé/valeur. L'interface Map possède trois méthodes essentielles :

boolean put(Object key, Object value)

//ajoute la paire key/value *dans la carte. Si la clé* key *était déjà présente dans la carte, l'ancienne valeur est remplacée par la nouvelle (une clé est unique)*

Object get(Object key)

//renvoie la valeur associée à la clé key *dans la carte ou* null *si la clé n'a pas été trouvée dans la carte*

void remove(Object key)

//supprime la paire de la carte dont la clé est key

L'interface Map possède trois méthodes qui permettent de parcourir le contenu d'une carte :

Set keySet()

//renvoie l'ensemble des clés de la carte

Collection values()

//renvoie l'ensemble des valeurs de la carte

Set entrySet()

//renvoie la collection des paires clé/valeur de la carte sous la forme d'objets de la classe Map.Entry

c. L'interface Iterator

Un **itérateur** permet de parcourir les éléments d'une collection. L'interface Iterator possède trois méthodes essentielles :

boolean hasNext()
//renvoie true s'il reste un élément à parcourir

Object next()
//renvoie le prochain élément à parcourir et déclenche une exception NoSuchElementException *si la fin de la collection est atteinte*

void remove()
//supprime l'élément renvoyé par le dernier appel à la méthode next. *Cette méthode doit impérativement être appelée après la méthode* next

Un **itérateur de liste** permet de parcourir les éléments d'une liste. L'interface ListIterator, dérivée de l'interface Iterator, possède trois méthodes essentielles :

void add(Object obj)
//ajoute l'élément obj *avant la position courante*

boolean hasPrevious()
//renvoie true s'il existe un élément avant la position courante

Object previous()
//renvoie l'élément précédent et déclenche une exception NoSuchElementException *si le début de la liste est atteinte*

d. Les classes abstraites de collections

Nous avons présenté les méthodes essentielles de chaque interface de collections. En fait, ces interfaces possèdent d'autres méthodes qui peuvent être implémentées très simplement. Il existe cinq classes abstraites qui fournissent des implémentations de ces méthodes : AbstractCollection, AbstractList, AbstractSequentialList, AbstractSet et AbstractMap.

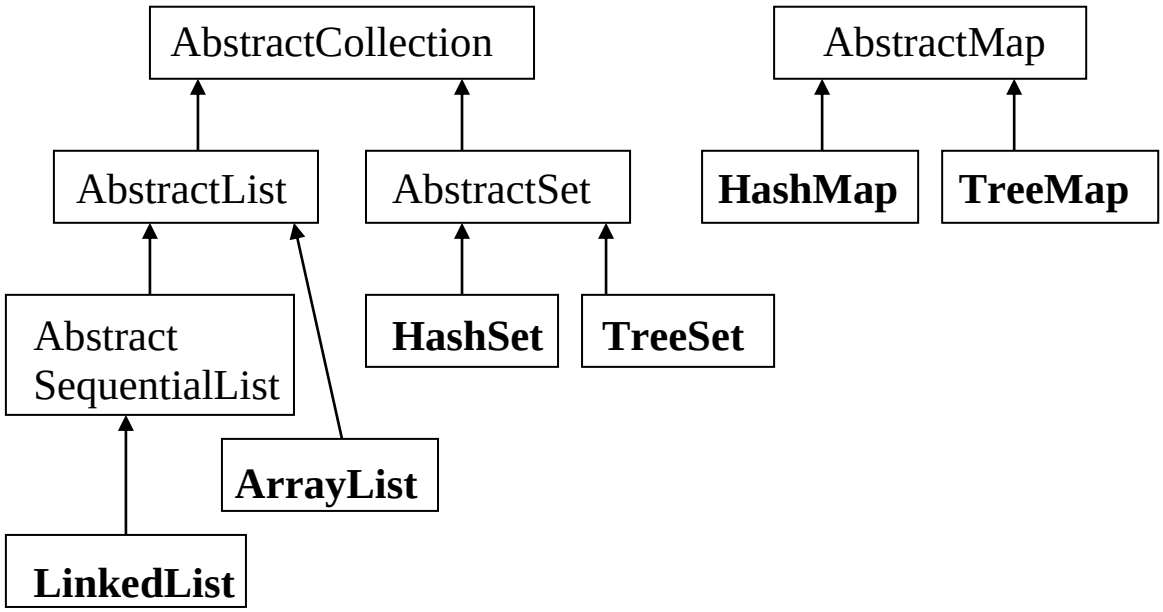
Exemple

La classe AbstractCollection implémente le corps de la plupart des méthodes de l'interface Collection.

```
public class AbstractCollection implements Collection
{
   …
   public abstract boolean add(Object obj) ;
   public abstract Iterator iterator() ;
   public boolean addAll(Collection other) {
      //ajoute tous les éléments de la collection other à la
      collection, et renvoie true si cet ajout a effectivement
      modifié la collection
      Iterator iter = other.iterator();
      boolean modified = false;
      while(iter.hasNext())
         if(add(iter.next()))
            modified = true;
      return modified;
   }
   …
}
```

V.3.2. Les classes de collections

La bibliothèque Java fournit six classes de collections : LinkedList, ArrayList, HashSet, TreeSet, HashMap et TreeMap.



La classe LinkedList permet de représenter des listes chaînées.

Exemple

```
LinkedList liste = new LinkedList() ;
liste.add("A"); liste.add("B"); liste.add("C");
   //ajoute les éléments A, B et C à la liste
Iterator iter = liste.iterator();
while (iter.hasNext())
   System.out.print(iter.next() + " ") ;
      //affiche A B C
iter.remove() ;
   //supprime le dernier élément parcouru : C
ListIterator iter2 = liste.listIterator()
   //renvoie un itérateur de liste
iter2.next() ; iter2.add("D") ;
   //saute le premier élément de la liste et ajoute D avant le
   deuxième élément. La nouvelle liste est : A D B
```

La classe ArrayList permet de représenter des tableaux dynamiques d'objets.

La classe HashSet permet de représenter des ensembles. La classe TreeSet permet de représenter des ensembles triés.

Exemple

```
TreeSet ens = new TreeSet() ;
ens.add("B"); ens.add("A"); ens.add("C");
Iterator iter = ens.iterator();
while (iter.hasNext())
   System.out.print(iter.next() + " ") ;  //affiche A B C
ens.add("A");      //l'élément A n'est pas ajouté à
                   l'ensemble car il s'y trouve déjà
```

La classe HashMap permet de représenter des cartes. La classe TreeMap permet de représenter des cartes dont les clés sont triées.

Exemple

```
HashMap carte = new HashMap() ;
carte.put("101", new Employe("Dupond"));
carte.put("102", new Employe("Durand"));
carte.put("103", new Employe("Hector"));
  //ajoute trois paires à la carte
Employe e = (Employe)(carte.get("101")) ;
  //renvoie les informations sur l'employé de nom Dupond.
  Le cast est obligatoire
Set valeurs = carte.entrySet() ;
Iterator iter = valeurs.iterator();
while (iter.hasNext()) {
   Map.Entry valeur = (Map.Entry)(iter.next());
   String cle = (String)valeur.getKey();
     //la méthode Object getKey() de la classe Map.Entry
     renvoie la clé de l'objet courant
   Employe e = (Employe)valeur.getValue();
     //la méthode Object getValue() de la classe Map.Entry
     renvoie la valeur de l'objet courant
   System.out.println(cle + " " + e.getNom()) ; }
```

TD10. Collections

Comprendre le code d'une classe

On a défini la classe Eleve, dont les instances représentent des élèves de seconde année d'AgroParisTech. Pour simplifier, on a supposé que chaque élève était représenté par un nom (homonymes possibles) et identifié par un matricule de cinq caractères, de la forme *2Annn*, dans laquelle *nnn* est un entier de trois chiffres (de 001 à 999).
Examinez cette classe et répondez aux questions qui suivent la définition.

```
public class Eleve {
  private String numeroEleve;      /* matricule-élève, identifiant
                                   unique : 2Axxx*/
  private static int DernierNumero = 0;  /*indique le dernier
                                         matricule utilisé et mis à jour
                                         au chargement de la BD */
  private String nomEleve;  // nom de l'élève (homonymes possibles)

  public Eleve(String leNom) {
     int numeroEntier = ++Eleve.DernierNumero;
    /* l'instruction précédente est équivalente à :
     Eleve.DernierNumero = Eleve.DernierNumero + 1;
     numeroEntier = Eleve.DernierNumero;*/
      String chiffres = String.valueOf(DernierNumero);
      if (numeroEntier < 100)
            if (numeroEntier > 10) chiffres = '0' + chiffres;
            else chiffres = "00" + chiffres;
      numeroEleve = "2A" + chiffres;
      nomEleve = leNom.toUpperCase(); }
  public String getNomEleve() {
      return nomEleve; }
  public String getNumeroEleve() {
      return numeroEleve; }
  public String toString() {
      return getClass().getName() + "(" + numeroEleve
      + ", " + nomEleve + ")"; }
  public boolean equals(Eleve autreEleve) {
      return this.getNumeroEleve().equals(
      autreEleve.getNumeroEleve()); }
}
```

a. Quelles sont les variables d'instance de la classe ?
b. DernierNumero est-elle une variable d'instance. Quel rôle joue-t-elle ?
c. Expliquez comment le constructeur utilise la variable DernierNumero.
d. Dans le constructeur de la classe, on utilise la méthode de classe valueOf de la classe String :
   - Quel est son rôle ? (Pour répondre, utilisez, si nécessaire, la documentation Java accessible via Eclipse).
   - Pourquoi a-t-on éventuellement besoin de compléter la valeur que renvoie cette méthode par un ou deux caractères 0 (zéro) ?
e. Si e1 désigne l'objet Eleve dont le matricule est 2A025 et dont le nom est Caroline, quel sera le résultat de l'exécution de : System.out.println(e1);
f. Que fait la méthode equals ?

Répertorier les inscriptions d'un module

On considère la classes Module utilisée par une application de gestion des inscriptions. Un objet Module décrit un module. Il indique le nom du module, le nombre maximal de places et la liste des élèves inscrits (dans un objet ArrayList). La classe Module est décrite ci-dessous.

```
public class Module {
        private String nomModule;
        private int nbPlaces;  // nb max. possible d'inscrits
        private ArrayList listeEleves; //répertorie les inscrits

        private int valideNbPlaces(int leNbPlaces) {
              if (leNbPlaces < 4) {
                    System.out.println("Sortie du programme :
                    nombre de places du module incorrect!");
                    System.exit(1);
          }
              return leNbPlaces;
        }
        public Module(String leNom, int leNbPlaces) {
          nomModule = leNom.toUpperCase();
```

```
        nbPlaces = valideNbPlaces(leNbPlaces);
        listeEleves = new ArrayList();
    }

    public boolean inscrire(Eleve nouvelInscrit) {
        // renvoie true si on a pu inscrire le nouvel élève
        //A COMPLETER
        /* Avant d'inscrire le nouvel élève, il faut vérifier qu'il reste
encore de la place dans le module et que le nouvel inscrit n'est pas déjà
inscrit dans le module */
    }
    public boolean annulerInscription(Eleve dejaInscrit){
        // renvoie true si on a pu annuler l'inscription
        // A COMPLETER
    }
    public String toString() {
        String description = getClass().getName() +
         "(" + nomModule;
        description += ", places: " + nbPlaces + ", inscrits: ";
        if (listeEleves.size() == 0)
            description += "0)";
        else {
            description += listeEleves.size();
            Iterator iter = listeEleves.iterator();
            while (iter.hasNext()) {
              description += ", " + ((Eleve) iter.next());
              description += ')';
        }
        return description;
    }
}
```

Examinez-cette classe et répondez à la question suivante : sachant que l'on envisage de gérer une liste d'attente pour chaque module dont le nombre d'inscrits atteint la limite, expliquez pourquoi la représentation de la liste des inscrits par un objet ArrayList est préférable à un tableau ou à un objet HashMap.

Créez un projet TD10. Importez les fichiers Eleve, Module et Test du répertoire Java_td_collection. Complétez les méthodes

inscrire et annulerInscription de la classe Module. Pour tester la classe Module, vous pourrez utiliser la classe Test dont la définition est donnée ci-dessous.

```
public class Test {
  public static void main(String args[]) {
    Eleve e1=new Eleve("Jean");
    Eleve e2=new Eleve("Anne");
    Eleve e3=new Eleve("Louis");
    Eleve e4=new Eleve("Henri");
    Module m1=new Module("CoursInformatique",20);
    m1.inscrire(e1); m1.inscrire(e2);
    m1.inscrire(e3); m1.inscrire(e4);
    System.out.println("m1 = " + m1);
    m1.inscrire(e3);
    m1.annulerInscription(e2);
    System.out.println("m1 = " + m1);
  }
}
```

On obtient à l'exécution :

```
m1 = Module(COURSINFORMATIQUE, places: 20, inscrits: 4, Eleve(2A001, JEAN)), Eleve(2A002, ANNE)), Eleve(2A003, LOUIS)), Eleve(2A004, HENRI))
L'eleve Eleve(2A003, LOUIS) était déjà inscrit !
m1 = Module(COURSINFORMATIQUE, places: 20, inscrits: 3, Eleve(2A001, JEAN)), Eleve(2A003, LOUIS)), Eleve(2A004, HENRI))
```

Utiliser un TreeSet

Modifier la classe Module en représentant la liste des inscrits non plus par un objet ArrayList mais par un objet TreeSet. Vous utiliserez l'interface Comparator<Eleve> qui contient la méthode compare pour pouvoir comparer deux élèves (une nouvelle classe ComparerEleve devra implémenter l'interface Comparator<Eleve> et définir la méthode compare).

TD11. Collections et classes enveloppes

L'objectif de ce TD est d'utiliser les classes enveloppes (cf. section III.11) et de manipuler les conversions entre chaînes, types primitifs et type classe (cf. section V.2.5).

Créez un premier programme (dans une classe contenant un programme principal) dans lequel il vous faudra :

- Créer une collection (en utilisant la classe ArrayList)
- Y ajouter 3 entiers sous la forme de chaîne de caractères (par exemple : "45")
- Utiliser la méthode static lireEntier() de la classe TDOutils pour y ajouter un entier saisi par l'utilisateur.
- Utiliser la méthode static lireTexte() de la classe TDOutils pour y ajouter un entier sous la forme d'une chaîne de caractères saisie par l'utilisateur.
- Afficher la collection obtenue
- Calculer la moyenne des entiers contenus dans la collection et l'afficher.

Créez un deuxième programme (dans une deuxième classe contenant un programme principal) qui fait la même chose mais en créant une collection d'entiers (en utilisant toujours la classe ArrayList, mais sans utiliser la classe String).

Créez un projet TD11. Importez les fichiers TDOutils et TDOutilsException qui sont dans le répertoire Java_td_collection_v2.

VI. La gestion des exceptions

VI.1. Introduction

La **gestion des exceptions** en Java est un mécanisme qui permet à la fois :

de dissocier la détection d'une anomalie de son traitement,

de séparer la gestion des anomalies du reste du code, donc de contribuer à la lisibilité du programme.

Une exception est une rupture de séquence déclenchée par une instruction throw comportant un objet de type CCC_Exception où CCC_Exception est une classe dérivée de la classe standard Exception (du paquetage java.lang). Il y a alors branchement à un ensemble d'instructions nommé "gestionnaire d'exception". Le choix du bon gestionnaire est fait en fonction du type de l'objet fourni à l'instruction throw.

VI.2. Un premier exemple d'exception

```
//La classe DepartementException, classe dérivée de la classe standard Exception, permet d'instancier des exceptions.
class DepartementException extends Exception {…}

class Personne {
  private String nom ;
  private byte departement ;

  public void setNom (String nom) {
    this.nom = nom.toUpperCase() ;
  }
  private byte validerDepartement(int dept) throws DepartementException {
    if (dept < 1 || dept > 95)
      throw new DepartementException() ;
    return (byte) dept ;
  }
  public void setDepartement (int dept) throws DepartementException {
    departement = validerDepartement(dept) ;
  }
  public Personne (String lenom, int dept) throws DepartementException {
    setNom(lenom) ;
    setDepartement(dept) ;
  }
  public void affiche () {
    if (nom != null)
      System.out.println("Je m'appelle " + nom) ;
    if (departement != 0)
      System.out.println("J'habite dans le " +
      departement) ;
  }
}
```

```
public class MonProgException {
   public static void main(String args[]) {
      try {
         Personne p1 = new Personne("Durand", 75) ;
         p1.affiche() ;
         Personne p2 = new Personne("Dupond", 100) ;
         p2.affiche() ;
      }
      catch(DepartementException e) {
         System.out.println("n° de département incorrect") ;
         System.exit(-1) ;
      }
   }
}
```

Je m'appelle DURAND
J'habite dans le 75
n° de département incorrect

VI.3. Comment lever une exception

L'instruction **throw** permet de **lever une exception** qui est représentée par un objet de type CCC_Exception où CCC_Exception est une classe dérivée de la classe standard Exception.

Exemple

//L'instruction throws DepartementException *indique que la méthode* validerDepartement *est susceptible de lever une exception de type* DepartementException. *Cette indication est obligatoire dans la mesure où l'exception n'est pas traitée par la méthode elle-même.*

```
private byte validerDepartement(int dept) throws
DepartementException {
  if (dept < 1 || dept > 95)
    throw new DepartementException() ;
      //l'instruction throw permet de lever une exception de
      type DepartementException.
  return (byte) dept ;
    //cette instruction est exécutée si aucune exception n'a
    été levée
}
```

VI.4. Comment capturer et traiter une exception

VI.4.1. Utilisation d'un gestionnaire d'exception

Pour capturer et traiter une exception de type CCC_Exception levée par une instruction throw, il faut :

- inclure dans un bloc **try** les instructions pour lesquelles on désire capturer une telle exception ;
- associer à ce bloc la définition du **gestionnaire d'exception** qui permettra de capturer et de traiter les exceptions de type CCC_Exception. La définition d'un gestionnaire est précédée d'un en-tête introduit par le mot clé **catch** .

```
try {
   …        //instructions susceptibles de lever des exceptions de type CCC_Exception
}
catch(CCC_Exception e) { … }
   //Le bloc catch est un gestionnaire d'exception associé au bloc try. Il capture les exceptions de type CCC_Exception et les traite en provoquant l'exécution des instructions qu'il contient.
```

Exemple

```
public static void main(String args[]) {
   try {
      Personne p = new Personne("Dupond", 100) ;
   }
      //le gestionnaire d'exception affiche un message d'erreur et interrompt l'exécution du programme
   catch(DepartementException e) {
      System.out.println("n° de département incorrect") ;
      System.exit(-1) ; }
}
```

VI.4.2. Gestion de plusieurs exceptions

A un bloc try peuvent être associés plusieurs gestionnaires d'exceptions, chaque gestionnaire permettant de capturer et de traiter des exceptions d'un type donné.

```
try {
   …
}
catch(CCC_Exception1 e) { … }
catch(CCC_Exception2 e) { … }
catch(CCC_Exception3 e) { … }
```

L'exécution du bloc try est interrompue dès qu'une exception est levée et il y a branchement au gestionnaire d'exception correspondant. Ainsi seule la première exception levée est capturée et traitée.

VI.4.3. Choix du gestionnaire d'exception

Lorsqu'une exception est levée (par une instruction throw) dans un bloc try, l'exécution du bloc try est interrompue et il y a branchement au gestionnaire d'exception qui permet de capturer et de traiter les exceptions du type de l'objet fourni à l'instruction throw.
La recherche du "bon" gestionnaire a lieu dans l'ordre où apparaissent les gestionnaires d'exception associés au bloc try. On sélectionne le premier qui permet de capturer et de traiter soit des exceptions du type exact de l'objet, soit d'un type ascendant.
Exemple

```
public class CCC_Exception extends Exception
  { … }
public class CCC_Exception1 extends CCC_Exception
  { … }
public class CCC_Exception2 extends CCC_Exception
  { … }

try {
  …
}
catch(CCC_Exception e) { … }
  //le gestionnaire d'exception capture et traite les exceptions de type CCC_Exception, CCC_Exception1 et CCC_Exception2.
```

VI.4.4. Poursuite de l'exécution du programme

Un gestionnaire d'exception ne met pas nécessairement fin à l'exécution du programme. Dans ce cas, l'exécution du programme se poursuit simplement avec les instructions suivant le bloc try (plus précisément, le dernier bloc catch associé à ce bloc try).

Exemple

```
public static void main(String args[]) {
   try {
      Personne p1 = new Personne("Durand", 75) ;
      Personne p2 = new Personne("Dupond", 100) ;
   }
      //le gestionnaire d'exception affiche un message d'erreur,
      mais n'interrompt pas l'exécution du programme
   catch(DepartementException e) {
      System.out.println("n° de département incorrect") ;
   }
   … /* suite de l'exécution du programme : attention les
      variables de classe p1 et p2 ne sont pas connues ici. */
}
```

VI.4.5. Le bloc finally

Nous avons vu que la levée d'une exception provoque un branchement inconditionnel au gestionnaire d'exception correspondant. Il n'est pas possible de revenir à la suite de l'instruction ayant provoqué l'exception. Cependant, Java permet d'introduire, à la suite d'un bloc try, un bloc particulier d'instructions qui sera exécuté :

- soit après la fin "naturelle" du bloc try, si aucune exception n'a été levée ;
- soit après le gestionnaire d'exception (à condition que ce dernier n'ait pas provoqué l'arrêt de l'exécution).

Ce bloc est introduit par le mot clé finally et doit obligatoirement être placé après le dernier gestionnaire d'exception.

Exemple

```
public static void main(String args[]) {
   String nom ;
   int dept ;
   …     //ouverture d'un fichier pour obtenir les valeurs des
         variables nom et dept
   try {
      Personne p = new Personne(nom, dept) ;
   }
   catch(DepartementException e) {
      System.out.println("n° de département incorrect") ;
   }
   finally {
      … //fermeture du fichier
   }
}
```

VI.4.6. Transmission d'informations au gestionnaire d'exception

On peut transmettre une information au gestionnaire d'exception :

- par le biais de l'objet fourni à l'instruction throw ;
- par l'intermédiaire du constructeur de la classe standard Exception.

a. Par l'objet fourni à l'instruction throw

Comme nous l'avons vu le type de l'objet fourni à l'instruction throw sert à choisir le bon gestionnaire d'exception. Mais cet objet est aussi récupéré par le gestionnaire d'exception sous la forme d'un argument, de sorte qu'il peut être utilisé pour transmettre une information.

Exemple

```
class DepartementException extends Exception {
   private int errDept ;
   public DepartementException(int err) {
      super(); errDept = err ; }
   public int geterrDept() {
      return errDept ; }
}
```

Dans la classe Personne, la méthode validerDepartement devient :

```
private byte validerDepartement(int dept) throws
DepartementException {
   if (dept < 1 || dept > 95)
      throw new DepartementException(dept) ;
   return (byte) dept ;
}

public class MonProgException {
   public static void main(String args[]) {
      try {
         Personne p = new Personne("Dupond", 100) ;
      }
      catch(DepartementException e) {
         System.out.println("n° de département incorrect :
         " + e.geterrDept()) ;
         System.exit(-1) ;
      }
   }
}
```

b. Par le constructeur de la classe Exception

La classe Exception dispose d'un constructeur à un argument de type String dont on peut récupérer la valeur à l'aide de la méthode getMessage.

Exemple

```
class DepartementException extends Exception {
  //ce constructeur est superflu
  public DepartementException(String message) {
    super(message) ; }
}
```

Dans la classe Personne, la méthode validerDepartement devient :

```
private byte validerDepartement(int dept) throws
DepartementException {
  if (dept < 1 || dept > 95)
    throw new DepartementException("n° de
    département incorrect : " + dept);
  return (byte) dept ;
}

public class MonProgException {
  public static void main(String args[]) {
    try {
      Personne p = new Personne("Dupond", 100) ;
    }
    catch(DepartementException e) {
      System.out.println(e.getMessage()) ;
      System.exit(-1) ;
    }
  }
}
```

VI.5. De la détection à la capture

VI.5.1. Le cheminement des exceptions

Lorsqu'une méthode f lève une exception (par une instruction throw), son exécution est interrompue et il y a branchement au gestionnaire d'exception qui permettra de capturer et de traiter l'exception.
La recherche d'un gestionnaire d'exception commence dans l'éventuel bloc try contenant l'instruction throw correspondante. Si l'on n'en trouve pas ou si aucun bloc try n'est prévu à ce niveau, on poursuit la recherche dans un éventuel bloc try associé à l'instruction d'appel de la méthode f dans une méthode appelante, et ainsi de suite.

Ainsi, une méthode susceptible de lever une exception peut :
- soit capturer et traiter l'exception ;
- soit **propager** l'exception (c'est à dire la retransmettre à sa propre méthode appelante). Le fil des appels, d'appelé en appelant, est alors remonté jusqu'à trouver une méthode qui s'est déclarée candidate pour le traitement de l'exception.

💥 *Le gestionnaire d'exception est rarement trouvé dans la méthode qui a levé l'exception puisque l'un des objectifs fondamentaux du traitement des exceptions est précisément de séparer détection et traitement.*

VI.5.2. Le mot clé throws

Toute méthode f susceptible de lever une exception de type CCC_Exception qu'elle ne traite pas localement doit avoir dans son en-tête le mot clé throws suivi du type de l'exception. Cette indication est obligatoire et a un double objectif :

- elle a une valeur documentaire : tout appel de la méthode f est susceptible de lever une exception de type CCC_Exception ;
- elle précise au compilateur que toute méthode qui appellera la méthode f devra elle aussi se préoccuper de l'exception de type CCC_Exception soit en la traitant, soit en la propageant.

Cette indication concerne donc les exceptions que la méthode peut lever directement par l'instruction throw, mais aussi toutes celles que peuvent lever (sans les traiter) toutes les méthodes qu'elle appelle.

Exemple

//L'instruction throws DepartementException *indique que la méthode* validerDepartement *est susceptible de lever une exception de type* DepartementException. *Cette indication est obligatoire dans la mesure où l'exception n'est pas traitée par la méthode elle-même.*

```
private byte validerDepartement(int dept) throws DepartementException {
  if (dept < 1 || dept > 95)
    throw new DepartementException() ;
  return (byte) dept ; }
  //la méthode setDepartement appelle la méthode validerDepartement qui est susceptible de lever une exception de type DepartementException.
public void setDepartement (int dept) throws DepartementException {
  departement = validerDepartement(dept) ; }
```

VI.5.3. Redéclenchement d'une exception

Dans un gestionnaire d'exception, il est possible de demander que l'exception traitée soit retransmise à un niveau englobant, comme si elle n'avait pas été traitée. Il suffit pour cela de la relancer en appelant à nouveau l'instruction throw.

Exemple

```
class Personne {
   private String nom ;
   private byte departement ;

   private byte validerDepartement(int dept) throws
   DepartementException {
      if (dept < 1 || dept > 95)
         throw new DepartementException() ;
      return (byte) dept ;
   }
   public void setDepartement (int dept) throws
   DepartementException {
      try {
         departement = validerDepartement(dept) ;
      }
      catch(DepartementException e)
      {
         System.out.println("erreur dans la méthode
         setDepartement") ;
         throw e ;
      }
   }
   public Personne (String nom, int dept) throws
   DepartementException {
      this.nom = nom.toUpperCase() ;
      if(dept != 0) setDepartement(dept) ;
   }
}
```

```
public class MonProgException {
   public static void main(String args[]) {
      try {
         Personne p = new Personne("Dupond", 100) ;
      }
      catch(DepartementException e) {
         System.out.println("n° de département incorrect") ;
         System.exit(-1) ; }
   }
}
```

```
erreur dans la méthode setDepartement
n° de département incorrect
```

TD12. Exceptions

La définition proposée pour la classe Module est donnée ci-dessous.

```
public class Module {
        private String nomModule;
        private int nbPlaces;                   // nb max. possible
d'inscrits
        private ArrayList listeEleves;          // répertorie les
inscrits

        private int valideNbPlaces(int leNbPlaces) throws
        ModuleExceptions{
            if (leNbPlaces < 4) {
                System.out.println("Sortie du programme:
            nombre de places du module incorrect!");
                System.exit(1);
          }
            return leNbPlaces;
        }
        public Module(String leNom, int leNbPlaces) {
          nomModule = leNom.toUpperCase();
          nbPlaces = valideNbPlaces(leNbPlaces);
          listeEleves = new ArrayList();
        }
        public boolean inscrire(Eleve nouvelInscrit) {
          if (listeEleves.size() == nbPlaces) {   // si plus de
places
            System.out.println("Impossible d'inscrire l'élève
"+
     nouvelInscrit + " : il n'y a plus de place dans le
module");
            return false;
            }


          // ici, on est sûr qu'il reste au moins une place
```

```
        Iterator iter = listeEleves.iterator();
        while (iter.hasNext()) {
           Eleve prochainInscrit = (Eleve) iter.next();
           if (prochainInscrit.equals(nouvelInscrit)) {
                 System.out.println("L'eleve  "+  nouvelInscrit
+ "
                 était déjà inscrit !");
                 return false; }
        }
        //ici, il y a de la place et l'élève n'est pas encore inscrit
        listeEleves.add(nouvelInscrit);
        return true;
      }
      public boolean annulerInscription(Eleve dejaInscrit) {
          return listeEleves.remove(dejaInscrit); }
      public String toString() {
        String description = getClass().getName() + "("
     + nomModule;
        description += ", places: " + nbPlaces + ", inscrits:
";
        if (listeEleves.size() == 0) description += "0)";
        else {
          description += listeEleves.size();
          Iterator iter = listeEleves.iterator();
             while (iter.hasNext()) {
           description += ", " + ((Eleve) iter.next());
          description += ')';
             }
        }
        return description;
      }
}
```

Définissez une classe d'exceptions ModuleException qui permet de gérer les problèmes de place pour les inscriptions dans un module.

```
public class ModuleException extends Exception {
    public ModuleException(String message) {
        super(message);
    }
}
```

Le comportement de la classe ModuleException devra être le suivant :

1. si un module est créé avec un nombre de places invalide (c'est à dire inférieur à 4), l'exécution s'arrête après affichage d'un message d'erreur ;
2. si on tente d'inscrire un nouvel élève dans un module alors que celui-ci est déjà plein, un message d'avertissement s'affiche mais l'exécution se poursuit avec l'ancienne composition du module.
3. si on tente d'inscrire dans un module un nouvel élève qui s'y trouvait déjà, un message d'avertissement s'affiche mais l'exécution se poursuit avec l'ancienne composition du module.

Créez un projet TD11. Importez les fichiers Eleve, Module, ModuleException et Test du répertoire Java_td_exception. Modifiez la méthode valideNbPlaces et le constructeur de la classe Module pour gérer le premier type d'exception. Modifiez la méthode inscrire de la classe Module et la méthode main de la classe Test pour gérer les deux autres types d'exception.

VII. L'interface graphique

VII.1. Introduction

Il existe deux principaux types de **composants** susceptibles d'intervenir dans une interface graphique :

les **conteneurs** qui sont destinés à contenir d'autres composants, comme par exemple les fenêtres ;

les **composants atomiques** qui sont des composants qui ne peuvent pas en contenir d'autres, comme par exemple les boutons.

Les classes suivies d'un astérisque (*) sont abstraites.

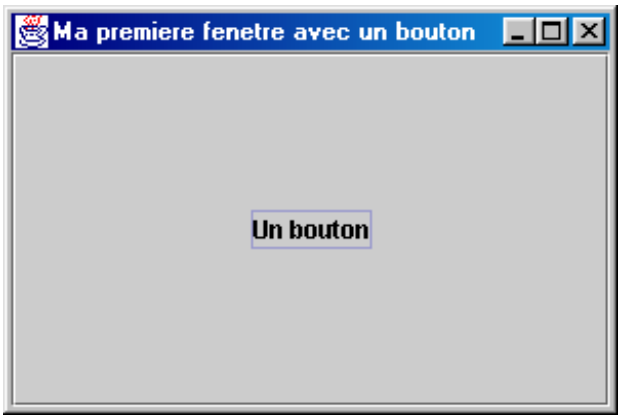
```
java.lang.Object
 |
 +--java.awt.Component*                          composants
     |
     +--java.awt.Container*                      conteneurs
         |
         +--java.awt.Panel
             |
             +--java.awt.Applet
                 |
                 +--javax.swing.JApplet
         |
         +--java.awt.Window
             |
             +-- javax.swing.JWindow
             |
             +--java.awt.Frame
                 |
                 +--javax.swing.JFrame           fenêtre graphique
             |
             +--java.awt.Dialog
                 |
                 +--javax.swing.JDialog          boîte de dialogue
```

```
java.lang.Object
 |
 +--java.awt.Component*                          composants
     |
     +--java.awt.Container*                      conteneurs
         |
         +--javax.swing.JComponent*              composants atomiques
             |
             +--javax.swing.JPanel               panneau utiliser pour dessiner
             |
             +--javax.swing.AbstractButton
                 |
                 +--javax.swing.JButton          bouton
                 |
                 +--javax.swing.JToggleButton
                     |
                     +--javax.swing.JCheckBox    case à cocher
                     |
                     +--javax.swing.JRadioButton bouton radio
                 |
                 +--javax.swing.JMenuItem        option d'un menu
                     |
                     +--javax.swing.JCheckBoxMenuItem      option d'un menu
                     |
                     +--javax.swing.JRadioButtonMenuItem   option d'un menu
                     |
                     +--javax.swing.JMenu        menu ou option d'un menu
             |
             +--javax.swing.JLabel               étiquette
             |
             +--javax.swing.text.JTextComponent
                 |
                 +--javax.swing.JTextField       champ de texte
             |
             +--javax.swing.text.JList           boîte de liste
             |
             +--javax.swing.text.JcomboBox       boîte de liste combinée
             |
             +--javax.swing.JScrollPane          panneau de défilement
             |
             +--javax.swing.JMenuBar             barre de menu
             |
             +--javax.swing.JPopUpMenu           menu surgissant
             |
             +--javax.swing.JToolBar             barre d'outils
```

- *Afin d'éviter d'avoir à s'interroger sans cesse sur la répartition dans les paquetages des différentes classes utilisées dans les interfaces graphiques, nous importerons systématiquement toutes les classes des paquetages* java.awt *et* javax.swing.

VII.2. Un premier exemple

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

class MaFenetre extends JFrame {
  private JButton MonBouton ;
  public MaFenetre () {
    super() ;
      //appel du constructeur par défaut de la classe JFrame
      qui peut être omis
    setTitle("Ma premiere fenetre avec un bouton") ;
      //initialisation du titre de la fenêtre
    setBounds(10,40,300,200) ;
      //le coin supérieur gauche de la fenêtre est placé au pixel
      de coordonnées 10, 40 et ses dimensions seront de 300 *
      200 pixels
    MonBouton = new JButton("Un bouton") ;
      //création d'un bouton de référence MonBouton portant
      l'étiquette Un bouton
    getContentPane().add(MonBouton) ;
      //Pour ajouter un bouton à une fenêtre, il faut incorporer
      le bouton dans le contenu de la fenêtre. La méthode
      getContentPane de la classe JFrame fournit une
      référence à ce contenu, de type Container.
      //La méthode add de la classe Container permet
      d'ajouter le bouton au contenu de la fenêtre.
  }
}

public class MonProg {
  public static void main(String args[]) {
    JFrame fen = new MaFenetre() ;
    fen.setVisible(true) ;
      //rend visible la fenêtre de référence fen
  }
}
```



Contrairement à une fenêtre, un bouton est visible par défaut.

Une fenêtre de type JFrame peut être retaillée, déplacée et réduite à une icône. **Attention**, la fermeture d'une fenêtre de type JFrame ne met pas fin au programme, mais rend simplement la fenêtre invisible (comme si on appelait la méthode setVisible(false)). Pour mettre fin au programme, il faut fermer la fenêtre console.

VII.3. Les fenêtres graphiques : la classe JFrame

La classe JFrame du paquetage javax.swing permet de créer des **fenêtres graphiques**.

La classe JFrame dispose de deux constructeurs : un constructeur sans argument et un constructeur avec un argument correspondant au titre de la fenêtre.

Remarque

Les instructions

```
JFrame fen = new JFrame() ;
fen.setTitle("Une fenêtre") ;
```

et l'instruction

```
JFrame fen = new JFrame("Une fenêtre") ;
```

sont équivalentes.

Une fenêtre est un conteneur destiné à contenir d'autres composants.

La méthode getContentPane de la classe JFrame fournit une référence, de type Container, au contenu de la fenêtre.

Les méthodes add et remove de la classe Container permettent respectivement d'ajouter et de supprimer un composant d'une fenêtre.

```
Component add(Component compo)
void remove(Component compo)
```

VII.4. Des méthodes utiles de la classe Component

La classe Component est une classe abstraite dérivée de la classe Object qui encapsule les composants d'une interface graphique.

Les méthodes suivantes de la classe Component permettent de gérer l'aspect d'un composant et donc en particulier d'une fenêtre.

Montrer et cacher un composant

```
void setVisible(boolean b)
```

Exemple :

```
compo.setVisible(true) rend visible le composant compo
```

Activer et désactiver un composant

```
void setEnabled(boolean b)
```

Exemple :

```
compo.setEnabled(true) active le composant compo
```

Connaître l'état (activé ou non) d'un composant

```
boolean isEnabled()
```

Exemple :

```
compo.isEnabled() retourne true si le composant compo est activé
```

Modifier la couleur de fond d'un composant

```
void setBackground(Color c)
```

Modifier la couleur du premier plan d'un composant

```
void setForeground(Color c)
```

Modifier la taille d'un composant

```
void setSize(int largeur, int hauteur)
```

Modifier la position et la taille d'un composant

```
void setBounds(int x, int y, int largeur, int hauteur)
```

Gérer les dimensions d'un composant

```
Dimension getSize()
void setSize(Dimension dim)
```

//La classe Dimension *du paquetage* java.awt *contient deux champs publics : un champ* height *(hauteur) et un champ* width *(largeur).*

VII.5. Les composants atomiques

La classe JComponent est une classe abstraite dérivée de la classe Container qui encapsule les composants atomiques d'une interface graphique. Les principaux composants atomiques offerts par Java sont les boutons, les cases à cocher, les boutons radio, les étiquettes, les champs de texte, les boîtes de liste et les boîtes de liste combinée.

```
java.lang.Object
 |
 +--java.awt.Component*
     |
     +--java.awt.Container*
         |
         +--javax.swing.JComponent*
             |
             +--javax.swing.AbstractButton
                 |
                 +--javax.swing.JButton                  bouton
                 |
                 +--javax.swing.JToggleButton
                     |
                     +--javax.swing.JCheckBox            case à cocher
                     |
                     +--javax.swing.JRadioButton         bouton radio
             |
             +--javax.swing.JLabel                       étiquette
             |
             +--javax.swing.text.JTextComponent
                 |
                 +--javax.swing.JTextField               champ de texte
             |
             +--javax.swing.text.JList                   boîte de liste
             |
             +--javax.swing.text.JcomboBox               boîte de liste combinée
             |
             +--javax.swing.JScrollPane                  panneau de défilement
```

La méthode setPreferredSize de la classe JComponent permet d'imposer une taille à un composant. Elle prend en argument un objet de type Dimension.

Exemple

```
JButton bouton = new JButton("Un bouton") ;
bouton.setPreferredSize(new Dimension(10, 20)) ;
  //bouton de largeur 10 et de hauteur 20
```

VII.5.1. Les cases à cocher

La case à cocher de type **JCheckBox** permet à l'utilisateur d'effectuer un choix de type oui/non.

Exemple

```
class MaFenetre extends JFrame {
   private JCheckBox MaCase;
   public MaFenetre () {
      super("Une fenetre avec une case") ;
      setBounds(10,40,300,200) ;
      MaCase = new JCheckBox("Une case") ;
         //création d'une case à cocher de référence MaCase
         portant l'étiquette Une case
      getContentPane().add(MaCase) ;
   }
}
```

Par défaut, une case à cocher est construite dans l'état non coché (*false*). La méthode isSelected de la classe AbstractButton permet de connaître l'état (*true* ou *false*) d'une case à cocher à un moment donné. La méthode setSelected de la classe AbstractButton permet de modifier l'état d'une case à cocher.

Exemple

```
MaCase.setSelected(true) ;
   //coche la case de référence MaCase
```

VII.5.2. Les boutons radio

Le bouton radio de type **JRadioButton** permet à l'utilisateur d'effectuer un choix de type oui/non. Mais sa vocation est de faire partie d'un groupe de boutons dans lequel une seule option peut être sélectionnée à la fois.

Exemple

```
class MaFenetre extends JFrame {
  private JRadioButton bRouge;
  private JRadioButton bVert;
  public MaFenetre () {
    super("Une fenetre avec des boutons radio") ;
    setBounds(10,40,300,200) ;
    bRouge = new JRadioButton("Rouge") ;
      //création d'un bouton radio de référence bRouge
      portant l'étiquette Rouge
    bVert = new JRadioButton("Vert") ;
      //création d'un bouton radio de référence bVert
      portant l'étiquette Vert
    ButtonGroup groupe = new ButtonGroup() ;
    groupe.add(bRouge) ; groupe.add(bVert) ;
      //Un objet de type ButtonGroup (classe du
      paquetage javax.swing, dérivée de la classe Object)
      sert uniquement à assurer la désactivation
      automatique d'un bouton lorsqu'un bouton du groupe
      est activé. Un bouton radio qui n'est pas associé à un
      groupe, exception faite de son aspect, se comporte
      exactement comme une case à cocher.
    Container contenu = getContentPane() ;
    contenu.setLayout(new FlowLayout()) ;
      //un objet de type FlowLayout est un gestionnaire de
      mise en forme qui dispose les composants les uns à la
      suite des autres
    contenu.add(bRouge) ;
```

```
        contenu.add(bVert) ;
          //Ajout de chaque bouton radio dans la fenêtre. Un
          objet de type ButtonGroup n'est pas un composant et
          ne peut pas être ajouté à un conteneur.
    }
  }
```

Par défaut, un bouton radio est construit dans l'état non sélectionné (*false*) (utilisation des méthodes isSelected et setSelected de la classe AbstractButton).

VII.5.3. Les étiquettes

Une étiquette de type **JLabel** permet d'afficher dans un conteneur un texte (d'une seule ligne) non modifiable, mais que le programme peut faire évoluer.

Exemple

```
class MaFenetre extends JFrame {
   private JLabel MonTexte;
   public MaFenetre () {
      super("Une fenetre avec une etiquette") ;
      setBounds(10,40,300,200) ;
      MonTexte = new JLabel ("texte initial") ;
         //création d'une étiquette de référence MonTexte
         contenant le texte texte initial
      getContentPane().add(MonTexte) ;
      MonTexte.setText("nouveau texte") ;
         //modification du texte de l'étiquette de référence
         MonTexte
   }
}
```

VII.5.4. Les champs de texte

Un champ de texte (ou boîte de saisie) de type **JTextField** est une zone rectangulaire dans laquelle l'utilisateur peut entrer ou modifier un texte (d'une seule ligne).

Exemple

```
class MaFenetre extends JFrame {
   private JTextField MonChamp1 ;
   private JTextField MonChamp2 ;
   public MaFenetre () {
      super("Une fenetre avec deux champs de texte") ;
      setBounds(10,40,300,200) ;
      MonChamp1 = new JtextField(20) ;
        //champ de taille 20 (la taille est exprimée en nombre
        de caractères standard affichés dans le champ)
      Monchamp2 = new JtextField("texte initial", 10) ;
        //champ de taille 10 contenant au départ le texte texte
        initial
      getContentPane().add(MonChamp1) ;
      getContentPane().add(MonChamp2) ;
   }
}
```

Aucun texte n'est associé à un tel composant pour l'identifier. On pourra utiliser un objet de type JLabel qu'on prendra soin de disposer convenablement.

La méthode getText permet de connaître à tout moment le contenu d'un champ de texte.

Exemple

```
String ch= Monchamp2.getText() ;
```

VII.5.5. Les boîtes de liste

La boîte de liste de type **JList** permet de choisir une ou plusieurs valeurs dans une liste prédéfinie. Initialement, aucune valeur n'est sélectionnée dans la liste.

Exemple

```
String[] couleurs = {"rouge", "bleu", "gris", "vert", "jaune", "noir"};
JList MaListe = new Jlist(couleurs) ;
MaListe.setSelectedIndex(2) ;
   //sélection préalable de l'élément de rang 2
```

Il existe trois sortes de boîtes de liste, caractérisées par un paramètre de type :

Valeur du paramètre de type	Type de sélection correspondante
SINGLE_SELECTION	sélection d'une seule valeur
SINGLE_INTERVAL_SELECTION	sélection d'une seule plage de valeurs (contiguës)
MULTIPLE_INTERVAL_SELECTION	sélection d'un nombre quelconque de plages de valeurs

Par défaut, le type d'une boîte de liste est MULTIPLE_INTERVAL_SELECTION.

La méthode setSelectionMode permet de modifier le type d'une boîte de liste.

Exemple

```
MaListe.setSelectionMode(SINGLE_SELECTION);
```

Pour une boîte de liste à sélection simple, la méthode getSelectedValue fournit la valeur sélectionnée. Son résultat est de type Object.

Exemple

```
String ch = (String) MaListe.getSelectedValue();
  //cast obligatoire
```

Pour les autres types de boîte de liste, la méthode getSelectedValue ne fournit que la première des valeurs sélectionnées. Pour obtenir toutes les valeurs, il faut utiliser la méthode getSelectedValues qui fournit un tableau d'objets.

Exemple

```
Object[] valeurs = MaListe.getSelectedValues();
for (int i=0 ;i<valeurs.length ;i++)
  System.out.println((String) valeurs[i]) ;
    //cast obligatoire
```

Par défaut une boîte de liste ne possède pas de **barre de défilement**. On peut lui en ajouter une en l'introduisant dans un panneau de défilement de type **JScrollPane** (classe dérivée de la classe JComponent). Dans ce cas, on n'ajoute pas au conteneur la boîte de liste, mais le paneau de défilement.

Exemple

```
JScrollPane defil = new JScrollPane(MaListe) ;
  //la liste de référence MaListe affiche 8 valeurs. Si elle en
  contient moins, la barre de défilement n'apparaît pas.
MaListe.setVisibleRowCount(3) ;
  //seules 4 valeurs de la liste sont visibles à la fois
getContentPane().add(defil) ;
  //ajoute le panneau de défilement au contenu de la fenêtre
```

VII.5.6. Les boîtes de liste combinée

La boîte de liste combinée (boîte combo) de type **JComboBox** associe un champs de texte et une boîte de liste à sélection simple. Tant que le composant n'est pas sélectionné, seul le champ de texte s'affiche:

Lorsque l'utilisateur sélectionne le champ de texte, la boîte de liste s'affiche :

L'utilisateur peut choisir une valeur dans la boîte de liste qui s'affiche alors dans le champ de texte. Initialement, aucune valeur n'est sélectionnée dans la liste.

Exemple

```
String[] couleurs = {"rouge", "bleu", "gris", "vert", "jaune", "noir"};
JComboBox MaCombo = new JComboBox(couleurs) ;
MaCombo.setSelectedIndex(2) ;
   //sélection préalable de l'élément de rang 2
```

La boîte combo est dotée d'une barre de défilement dès que son nombre de valeurs est supérieur à 8. On peut modifier le nombre de valeurs visibles avec la méthode setMaximumRowCount.

Exemple

```
MaCombo.setMaximumRowCount(4) ;
   //au maximum 4 valeurs sont affichées
```

Par défaut, le champ de texte associé à une boîte combo n'est pas éditable, ce qui signifie qu'il sert seulement à présenter la sélection courante de la liste. Mais il peut être rendu éditable par la méthode setEditable. L'utilisateur peut alors y entrer soit une valeur de la liste (en la sélectionnant), soit une valeur de son choix (en la saisissant).

Exemple

```
MaCombo.setEditable(true) ;
```

La méthode getSelectedItem fournit la valeur sélectionnée. Son résultat est de type Object.

Exemple

```
String ch = (String) MaCombo.getSelectedItem();
```

La méthode getSelectedIndex fournit le rang de la valeur sélectionnée. Elle fournit la valeur -1, si l'utilisateur a entré une nouvelle valeur (c'est à dire une valeur qui n'a pas été sélectionnée dans la liste).

La boîte combo dispose de méthodes appropriées à sa modification.

Exemple

```
MaCombo.addItem("orange") ;
   //ajoute la valeur orange à la fin de la liste combo
MaCombo.addItemAt("orange", 2) ;
   //ajoute la valeur orange à la positon 2 de la liste combo
MaCombo.removeItem("gris") ;
   //supprime la valeur gris de la liste combo
```

VII.6. Les menus et les barres d'outils

Une fenêtre de type JFrame est composée de composants atomiques, mais aussi de composants qui lui sont propres comme les menus et les barres d'outils.

```
java.lang.Object
 |
 +--java.awt.Component*
     |
     +--java.awt.Container*
         |
         +--javax.swing.JComponent*
             |
             +--javax.swing.AbstractButton
                 |
                 +--javax.swing.JMenuItem                    option d'un menu
                     |
                     +--javax.swing.JCheckBoxMenuItem        option d'un menu
                     |
                     +--javax.swing.JRadioButtonMenuItem     option d'un menu
                     |
                     +--javax.swing.JMenu                menu ou option d'un menu
             |
             +--javax.swing.JMenuBar                         barre de menu
             |
             +--javax.swing.JPopUpMenu                       menu surgissant
             |
             +--javax.swing.JToolBar                         barre d'outils
```

VII.6.1. Les menus déroulants

Les menus déroulants usuels font intervenir trois sortes d'objets:

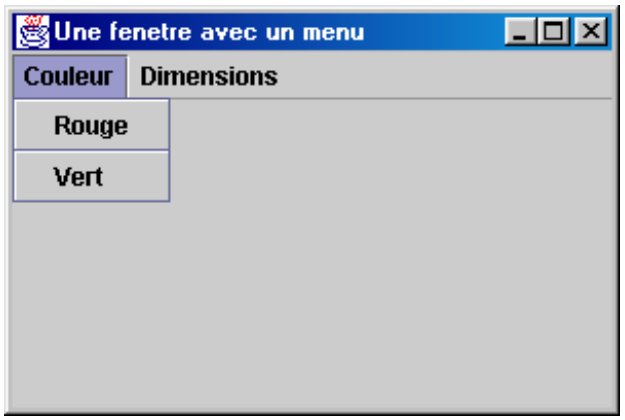
- un objet **barre de menu** de type JMenuBar ;
- différents objets **menu**, de type JMenu, visibles dans la barre de menu ;
- pour chaque menu, les différentes **options**, de type JMenuItem, qui le constituent.

La méthode setEnabled de la classe Component permet d'activer ou de désactiver un menu ou une option.

a. Un exemple

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

class MaFenetre extends JFrame {
   private JMenuBar barreMenus ;
   private JMenu couleur, dimensions ;
   private JMenuItem rouge, vert, hauteur, largeur ;
   public MaFenetre () {
      super("Une fenetre avec un menu") ;
      setSize(300, 200) ;
      //création d'une barre de menu
      barreMenus = new JMenuBar() ;
      setJMenuBar(barreMenus) ;
         //ajout de la barre de menu dans la fenêtre
      //création d'un menu Couleur et de ses options Rouge et
      Vert
      couleur = new JMenu("Couleur") ;
      barreMenus.add(couleur) ;
      rouge = new JMenuItem("Rouge") ;
      couleur.add(rouge) ;
      couleur.addSeparator() ; ;
         //ajout d'une barre séparatrice avant l'option suivante
      vert = new JMenuItem("Vert") ;
      couleur.add(vert) ;
      //création d'un menu Dimensions et de ses options
      Hauteur et Largeur
      dimensions = new JMenu("Dimensions") ;
      barreMenus.add(dimensions) ;
      hauteur = new JMenuItem("Hauteur") ;
      dimensions.add(hauteur) ;
      dimensions.addSeparator() ;
      largeur = new JMenuItem("Largeur") ;
      dimensions.add(largeur) ;
   }
}
```

```
public class MonMenu {
   public static void main(String args[]) {
      JFrame fen = new MaFenetre() ;
      fen.setVisible(true) ;
   }
}
```

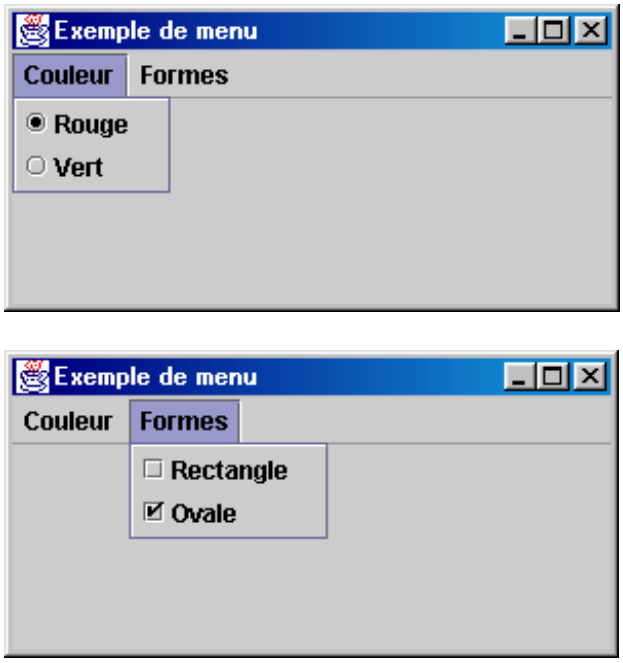


b. Les différentes sortes d'options

Les options de type JMenuItem sont les options les plus courantes dans un menu, mais il existe deux autres types d'options : des options cases à cocher de type JCheckBoxMenuItem et des options boutons radio de type JRadioButtonMenuItem.

Exemple

```
JMenuBar barreMenus = new JMenuBar() ;
setJMenuBar(barreMenus) ;
//création d'un menu Couleur et de son groupe de deux options Rouge et Vert
JMenu couleur = new JMenu("Couleur") ;
barreMenus.add(couleur) ;
JRadioButtonMenuItem rouge = new JRadioButtonMenuItem("Rouge") ;
JRadioButtonMenuItem vert = new JRadioButtonMenuItem("Vert") ;
couleur.add(rouge) ; couleur.add(vert) ;
ButtonGroup groupe = new ButtonGroup() ;
groupe.add(rouge) ; groupe.add(vert) ;
  //les options boutons radio sont placées dans un groupe de type ButtonGroup afin d'assurer l'unicité de la sélection à l'intérieur du groupe (cf paragraphe VII.5.2.)
//création d'un menu Formes et de ses cases à cocher Rectangle et Ovale
JMenu formes = new JMenu("Formes") ;
barreMenus.add(formes) ;
JCheckBoxMenuItem rectangle = new JCheckBoxMenuItem("Rectangle") ;
JCheckBoxMenuItem ovale = new JCheckBoxMenuItem("Ovale") ;
formes.add(rectangle) ; formes.add(ovale) ;
```



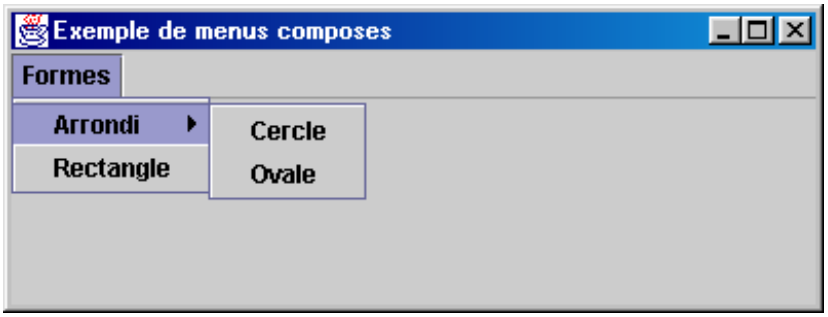
La méthode isSelected de la classe AbstractButton permet de savoir si une option est sélectionnée. Elle retourne *true* si l'option est sélectionnée, *false* sinon.

c. Composition des options

Jusqu'à présent, un menu était formé d'une simple liste d'options. En fait, une option peut à son tour faire apparaître une liste de sous-options. Pour ce faire, il suffit d'utiliser dans un menu une option de type JMenu (et non plus de type JMenuItem). Cette démarche peut être répétée autant de fois que voulu.

Exemple

```
JMenuBar barreMenus = new JMenuBar() ;
setJMenuBar(barreMenus) ;
//création d'un menu Formes composé d'une option Arrondi
composée elle même de deux options Cercle et Ovale, et,
d'une option Rectangle
JMenu formes = new JMenu("Formes") ;
barreMenus.add(formes) ;
JMenu arrondi = new JMenu("Arrondi") ;
formes.add(arrondi) ;
JMenuItem cercle = new JMenuItem("Cercle") ;
arrondi.add(cercle) ;
JMenuItem ovale = new JMenuItem("Ovale") ;
arrondi.add(ovale) ;
JMenuItem rectangle = new JMenuItem("Rectangle") ;
formes.add(rectangle) ;
```



VII.6.2. Les menus surgissants

Les menus usuels, que nous venons d'étudier, sont des menus rattachés à une barre de menu et donc affichés en permanence dans une fenêtre Java. Java propose aussi des **menus surgissants** de type JPopupMenu qui sont des menus (sans nom) dont la liste d'options apparaît suite à une certaine action de l'utilisateur.

Exemple

```
//création d'un menu surgissant comportant deux options Rouge et Vert
JPopupMenu couleur = new JPopupMenu() ;
rouge = new JMenuItem("Rouge") ;
couleur.add(rouge) ;
vert = new JMenuItem("Vert") ;
couleur.add(vert) ;
```

Un menu surgissant doit être affiché explicitement par le programme, en utilisant la méthode show de la classe JPopupMenu. Cette méthode a deux arguments : le composant concerné (en général, il s'agira de la fenêtre) et les coordonnées auxquelles on souhaite faire apparaître le menu (coordonnées du coin supérieur gauche). Le menu sera affiché jusqu'à ce que l'utilisateur sélectionne une option ou ferme le menu en cliquant à côté.

Exemple

```
JFrame fen = new JFrame() ;
fen.setVisible(true) ;
couleur.show(fen, x, y) ;
   //affiche le menu aux coordonnées x et y
```

VII.6.3. Les barres d'outils

Une barre d'outils de type JToolBar est un ensemble de boutons regrouper linéairement sur un des bords de la fenêtre. En général, ces boutons comportent plutôt des icônes que des libellés.

Exemple

```
class MaFenetre extends JFrame {
   JToolBar barreOutils ;
   JButton boutonRouge, boutonVert ;
   public MaFenetre () {
      super("Une fenetre avec une barre d'outils") ;
      setSize(300, 200) ;
      //création d'une barre d'outils composée de deux boutons: un bouton Rouge et un bouton Vert
      barreOutils = new JToolBar() ;
      boutonRouge = new JButton("Rouge") ;
      barreOutils.add(boutonRouge) ;
      boutonVert = new JButton("Vert") ;
      barreOutils.add(boutonVert) ;
      getContentPane().add(barreOutils) ; }
}
```

Par défaut, une barre d'outils est **flottante**, c'est à dire qu'on peut la déplacer d'un bord à un autre de la fenêtre, ou à l'intérieur de la fenêtre. On peut interdire à une barre d'outils de flotter grâce à la méthode setFloatable.

Exemple

```
barreOutils.setFloatable(false) ;
```

Un bouton peut être créé avec une icône de type ImageIcon (classe du paquetage javax.swing dérivée de la classe Object) au lieu d'un texte.

Exemple

```
JToolBar barreOutils = new JToolBar() ;
ImageIcon iconeVert = new ImageIcon("vert.gif") ;
  //création d'une icône à partir d'un fichier "vert.gif"
  contenant un dessin d'un carré de couleur vert
JButton boutonVert = new JButton(iconeVert) ;
barreOutils.add(boutonVert) ;
```

VII.6.4. Les bulles d'aide

Une bulle d'aide est un petit rectangle (nommé "tooltip" en anglais) contenant un bref texte explicatif qui apparaît lorsque la souris est laissée un instant sur un composant (boutons, menus, …). Java permet d'obtenir un tel affichage pour n'importe quel composant grâce à la méthode setToolTipText de la classe JComponent.

Exemple

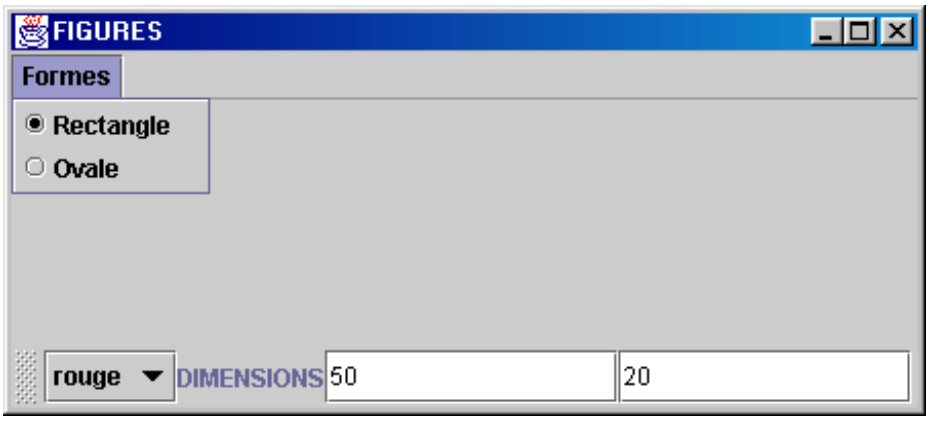
```
barreMenus = new JMenuBar() ;
setJMenuBar(barreMenus) ;
//création d'un menu Couleur et de ses options Rouge et Vert
couleur = new JMenu("Couleur") ;
barreMenus.add(couleur) ;
rouge = new JMenuItem("Rouge") ;
rouge.setToolTipText("fond rouge") ;
couleur.add(rouge) ;
couleur.addSeparator() ;
vert = new JMenuItem("Vert") ;
vert.setToolTipText("fond vert") ;
couleur.add(vert) ;
```

TD13. Une interface graphique

L'objectif des deux TD qui vont suivre est de permettre à l'utilisateur de choisir les formes qu'il souhaite dessiner dans une fenêtre, leurs dimensions et la couleur de fond. Les formes proposées se limiteront à l'ovale et au rectangle.

Construisez une fenêtre qui contient :

- une barre de menu composée d'un menu "Formes" qui permet de saisir la forme à dessiner parmi deux options boutons radio Rectangle et Ovale. On ne pourra choisir qu'une option à la fois ;
- une barre d'outils composée :
  - d'une boîte combo qui permet de choisir la couleur de fond du dessin parmi quatre couleurs (rouge, vert, jaune, bleu) ;
  - de deux champs de texte qui permettent de saisir les dimensions de la forme (largeur et hauteur) à dessiner ;



Afin que la barre d'outils barreOutils se situe en bas de la fenêtre principale, vous utiliserez l'instruction suivante :

```
getContentPane().add(barreOutils,"South") ;
```

Créez un nouveau projet qui contient une classe pour la fenêtre principale et une classe pour le programme principal.

VII.7. Les boîtes de dialogue

La boîte de dialogue est un conteneur. Elle permet de regrouper n'importe quels composants dans une sorte de fenêtre qu'on fait apparaître ou disparaître.

Java propose un certain nombre de boîtes de dialogue standard obtenues à l'aide de méthodes de classe de la classe JOptionPane : boîtes de message, boîtes de confirmation, boîtes de saisie et boîtes d'options. La classe JDialog permet de construire des boîtes de dialogue personnalisées.

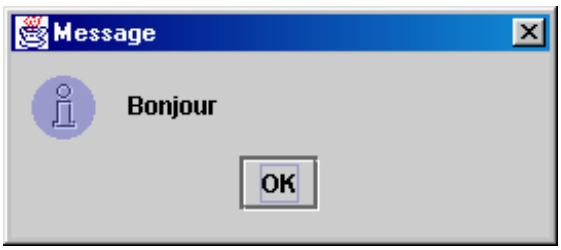
```
java.lang.Object
  |
  +--java.awt.Component*
        |
        +--java.awt.Container*
              |
              +--java.awt.Window
                    |
                    +--java.awt.Dialog
                          |
                          +--javax.swing.JDialog          boîte de dialogue personnalisée
              |
              +--javax.swing.JComponent*
                    |
                    +--javax.swing.JOptionPane          boîte de dialogue standard
```

VII.7.1. Les boîtes de message

Une boîte de message fournit à l'utilisateur un message qui reste affiché tant que l'utilisateur n'agit pas sur le bouton OK. Elle est construite à l'aide de la méthode de classe showMessageDialog de la classe JOptionPane.

Exemple

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

class MaFenetre extends JFrame {
   public MaFenetre () {
      super("Une fenetre") ; setSize(300, 200) ; }
}

public class BoiteMess {
   public static void main(String args[]) {
      JFrame fen = new MaFenetre() ;
      fen.setVisible(true) ;
      JOptionPane.showMessageDialog(fen, "Bonjour") ;
        //le premier argument de la méthode
        showMessageDialog correspond à la fenêtre parent
        de la boîte de message, c'est à dire la fenêtre dans
        laquelle la boîte de message va s'afficher. Cet
        argument peut prendre la valeur null.
   }
}
```



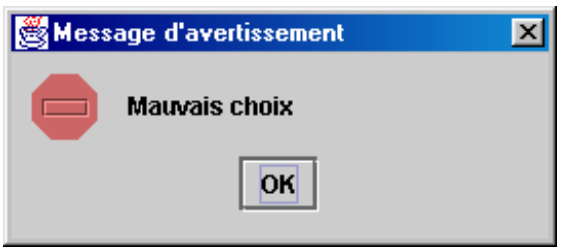
Dans l'exemple précédent, nous n'avons défini que le contenu du message. Il existe une variante de la méthode showMessageDialog qui permet aussi de choisir le titre de la

boîte et le type d'icône parmi la liste suivante (les paramètres sont des constantes entières de la classe JOptionPane).

Paramètre	Type d'icône
JOptionPane.ERROR_MESSAGE	Erreur
JOptionPane.INFORMATION_MESSAGE	Information
JOptionPane.WARNING_MESSAGE	Avertissement
JOptionPane.QUESTION_MESSAGE	Question
JOptionPane.PLAIN_MESSAGE	Aucune icône

Exemple

```
JOptionPane.showMessageDialog(fen, "Mauvais choix", "Message d'avertissement", JOptionPane.ERROR_MESSAGE) ;
```

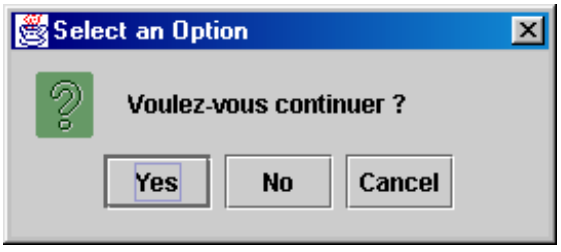


VII.7.2. Les boîtes de confirmation

Une boîte de confirmation offre à l'utilisateur un choix de type oui/non. Elle est construite à l'aide de la méthode de classe showConfirmDialog de la classe JOptionPane.

Exemple

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

class MaFenetre extends JFrame {
   public MaFenetre () {
      super("Une fenetre") ; setSize(300, 200) ; }
}

public class BoiteConf {
   public static void main(String args[]) {
      JFrame fen = new MaFenetre() ;
      fen.setVisible(true) ;
      JOptionPane.showConfirmDialog(fen, "Voulez-vous
      continuer ?") ;
   }
}
```

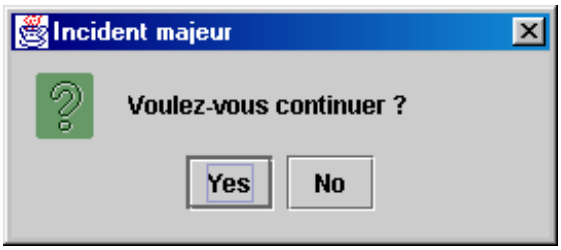


Dans l'exemple précédent, nous n'avons défini que la question posée à l'utilisateur. Il existe une variante de la méthode showConfirmDialog qui permet aussi de choisir le titre de la boîte et la nature des boutons qui s'y trouvent parmi la liste suivante (les paramètres sont des constantes entières de la classe JOptionPane).

Paramètre	Type de boîte
JOptionPane.DEFAULT_OPTION (-1)	Erreur
JOptionPane.YES_NO_OPTION (0)	boutons YES et NO
JOptionPane.YES_NO_CANCEL_OPTION (1)	boutons YES, NO et CANCEL
JOptionPane.OK_CANCEL_OPTION (2)	boutons OK et CANCEL

Exemple

```
JOptionPane.showConfirmDialog(fen, "Voulez-vous continuer ?", "Incident majeur", JOptionPane.YES_NO_OPTION) ;
```



La valeur de retour de la méthode showConfirmDialog précise l'action effectuée par l'utilisateur sous la forme d'un entier ayant comme valeur l'une des constantes suivantes de la classe JOptionPane :

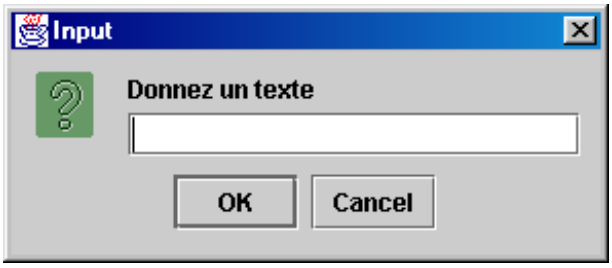
YES_OPTION (0),
OK_OPTION (0),
NO_OPTION (1),
CANCEL_OPTION (2),
CLOSED_OPTION (-1).

VII.7.3. Les boîtes de saisie

Une boîte de saisie permet à l'utilisateur de fournir une information sous la forme d'une chaîne de caractères. Elle est construite à l'aide de la méthode de classe showInputDialog de la classe JOptionPane.

Exemple

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

class MaFenetre extends JFrame {
   public MaFenetre () {
      super("Une fenetre") ; setSize(300, 200) ; }
}

public class BoiteSaisie {
   public static void main(String args[]) {
       JFrame fen = new MaFenetre() ;
       fen.setVisible(true) ;
       JOptionPane.showInputDialog (fen, "Donnez un
       texte") ;
   }
}
```

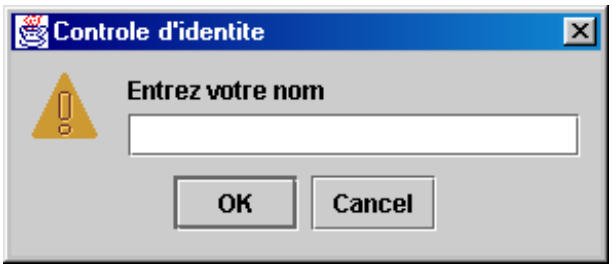


La valeur de retour de la méthode showInputDialog est soit un objet de type String contenant le texte fournit par l'utilisateur, soit la valeur *null* si l'utilisateur n'a pas confirmé sa saisie par le bouton OK.

Dans l'exemple précédent, nous n'avons défini que la question posée à l'utilisateur. Il existe une variante de la méthode showInputDialog qui permet aussi de choisir le titre de la boîte et le type d'icône (suivant les valeurs fournies au paragraphe VII.7.1.).

Exemple

```
JOptionPane.showInputDialog(fen, "Entrez votre nom",
"Controle d'identite",
JOptionPane.WARNING_MESSAGE) ;
```

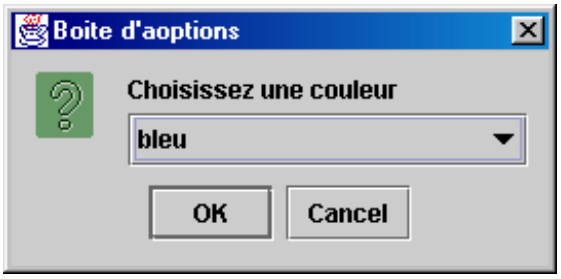


VII.7.4. Les boîtes d'options

Une boîte d'options permet à l'utilisateur de choisir une valeur parmi une liste de valeurs, par l'intermédiaire d'une boîte combo. Elle est construite à l'aide de la méthode de classe showInputDialog de la classe JOptionPane.

Exemple

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

class MaFenetre extends JFrame {
   public MaFenetre () {
      super("Une fenetre") ; setSize(300, 200) ; }
}

public class BoiteOptions {
   public static void main(String args[]) {
      JFrame fen = new MaFenetre() ;
      fen.setVisible(true) ;
      String[] couleurs = {"rouge", "bleu", "gris", "vert",
      "jaune", "noir"};
      JOptionPane.showInputDialog (fen, "Choisissez
      une couleur",
      "Boite d'options", //titre de la boîte
      JOptionPane.QUESTION_MESSAGE,
        //type d'icône suivant les valeurs du paragraphe
        VII.7.1.
      null,
        //icône supplémentaire (ici aucune)
      couleurs,
        //liste de valeurs représentée dans la boîte combo
      couleurs[1]) ;
        //valeur sélectionnée par défaut
   }
}
```



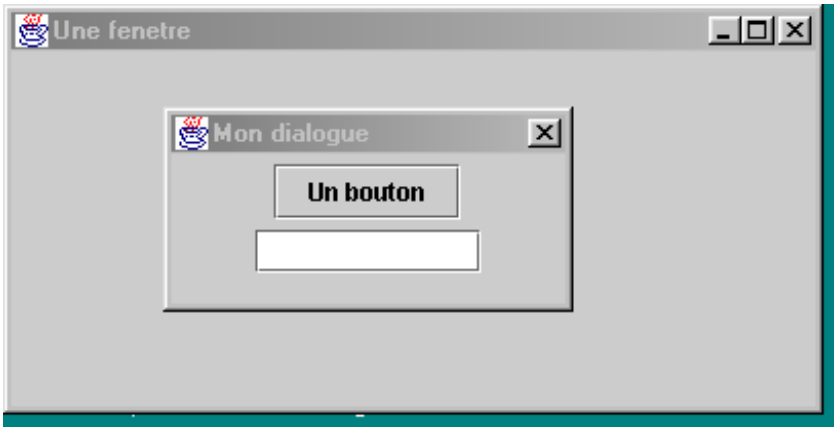
La valeur de retour de la méthode showInputDialog est soit un objet de type Object contenant la valeur sélectionnée par l'utilisateur, soit la valeur *null* si l'utilisateur n'a pas confirmé sa saisie par le bouton OK.

VII.7.5. Les boîtes de dialogue personnalisées

La classe JDialog permet de créer ses propres boîtes de dialogue.

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

class MaFenetre extends JFrame {
   public MaFenetre () {
      super("Une fenetre") ;
      setSize(300, 200) ; }
}

class MonDialog extends JDialog {
   private JButton MonBouton ;
   private JTextField MonChamp ;
   public MonDialog (JFrame fen) {
      super(fen,          //fenêtre parent
      "Mon dialogue",     //titre
      true) ;     //boîte modale : l'utilisateur ne peut agir que
                  sur les composants de la boîte de dialogue et
                  ceci tant qu'il n'a pas mis fin au dialogue
      setSize(200, 100) ;
        //il est nécessaire d'attribuer une taille à une boîte de
        dialogue avant de l'afficher
      MonBouton = new JButton("Un bouton") ;
      MonChamp = new JTextField(10) ;
      Container contenu = getContentPane() ;
      contenu.setLayout(new FlowLayout()) ;
          //un objet de type FlowLayout est un gestionnaire de
          mise en forme qui dispose les composants les uns à la
          suite des autres
      contenu.add(MonBouton); contenu.add(MonChamp) ;
        //ajout d'un bouton et d'un champ de texte dans la boîte
        de dialogue
   }
}
```

```
public class MonProgBoiteDialogPers {
   public static void main(String args[]) {
      JFrame fen = new MaFenetre() ;
      fen.setVisible(true) ;
      JDialog bd = new MonDialog(fen) ;
      bd.setVisible(true) ;
         // affiche la boîte de dialogue de référence bd
   }
}
```



VII.8. Les gestionnaires de mise en forme

Pour chaque conteneur (fenêtre, boîte de dialogue, …), Java permet de choisir un gestionnaire de mise en forme (en anglais "Layout manager") responsable de la disposition des composants.

Les gestionnaires de mise en forme proposés par Java sont les gestionnaires BorderLayout, FlowLayout, CardLayout, GridLayout, BoxLayout et GridBagLayout. Ce sont tous des classes du paquetage java.awt dérivées de la classe Object et qui implémentent l'interface LayoutManager.

La méthode setLayout de la classe Container permet d'associer un gestionnaire de mise en forme à un conteneur. Le gestionnaire BorderLayout est le gestionnaire par défaut des fenêtres et des boîtes de dialogue.

Exemple

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

class MaFenetre extends JFrame {
   public MaFenetre () {
      super("Une fenetre") ;
      setSize(300, 200) ;
      getContentPane().setLayout(new FlowLayout()) ;
         //changement de gestionnaire de mise en forme
   }
}
public class MonProgLayout {
   public static void main(String args[]) {
      JFrame fen = new MaFenetre() ;
      fen.setVisible(true) ; }
}
```

VII.8.1. Le gestionnaire BorderLayout

Le gestionnaire de mise en forme BorderLayout permet de placer chaque composant dans une zone géographique.

L'emplacement d'un composant est choisi en fournissant en argument de la méthode add de la classe Container l'une des constantes entières suivantes (on peut utiliser indifféremment le nom de la constante ou sa valeur) :

Constante symbolique	Valeur
BorderLayout.NORTH	"North"
BorderLayout.SOUTH	"South"
BorderLayout.EAST	"East"
BorderLayout.WEST	"West"
BorderLayout.CENTER	"Center"

Si aucune valeur n'est précisée à la méthode add, le composant est placé au centre.

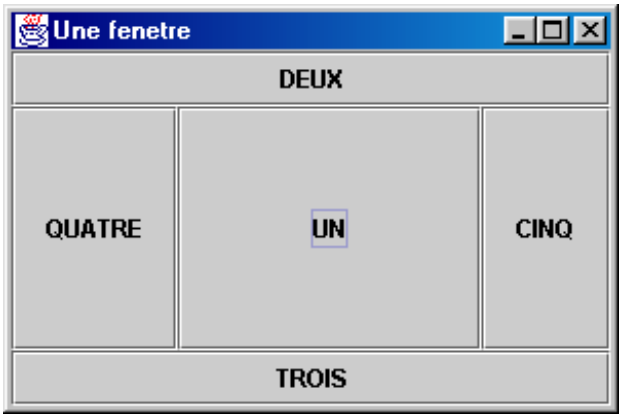
La classe BorderLayout dispose de deux constructeurs :

```
public BorderLayout() ;
public BorderLayout(int hgap, int vgap) ;
  //hgap et vgap définissent respectivement l'espace horizontal et l'espace vertical (en nombre de pixels) entre les composants d'un conteneur. Par défaut, les composants sont espacés de 5 pixels.
```

Exemple

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

class MaFenetre extends JFrame {
  public MaFenetre () {
    super("Une fenetre") ; setSize(300, 200) ;
    Container contenu = getContentPane() ;
    contenu.setLayout(new BorderLayout()) ;   //inutile
    contenu.add(new JButton("UN")) ;
      //bouton placé au centre par défaut
```

```
        contenu.add(new JButton("DEUX"), "North") ;
        contenu.add(new JButton("TROIS"), "South") ;
        contenu.add(new JButton("QUATRE"), "West") ;
        contenu.add(new JButton("CINQ"), "East") ;
    }
}

public class MonProgBLayout {
    public static void main(String args[]) {
        JFrame fen = new MaFenetre() ;
        fen.setVisible(true) ;
    }
}
```



Le gestionnaire de mise en forme BorderLayout *ne tient pas compte de la taille souhaitée des composants, qui peut être imposée par la méthode* setPreferredSize *de la classe* JComponent *(cf. paragraphe VII.5.).*

VII.8.2. Le gestionnaire FlowLayout

Le gestionnaire de mise en forme FlowLayout permet de disposer les composants les uns à la suite des autres, de gauche à droite.

La classe FlowLayout dispose de trois constructeurs :

```
public FlowLayout() ;
public FlowLayout(int align) ;
  //align est un paramètre d'alignement d'une ligne de composants par rapport aux bords verticaux de la fenêtre. Ce paramètre peut prendre ses valeurs parmi les constantes entières suivantes (on peut utiliser indifféremment le nom de la constante ou sa valeur) : FlowLayout.LEFT("Left"), FlowLayout.RIGHT("Right") ou FlowLayout.CENTER("Center"). Par défaut les composants sont alignés à gauche.
public FlowLayout(int align, int hgap, int vgap) ;
  //hgap et vgap définissent les espaces entre les composants.
```

Exemple

```
class MaFenetre extends JFrame {
  public MaFenetre () {
    super("Une fenetre") ; setSize(300, 200) ;
    Container contenu = getContentPane() ;
    contenu.setLayout(new FlowLayout()) ;
      //changement de gestionnaire de mise en forme
    contenu.add(new JButton("UN")) ;
    contenu.add(new JButton("DEUX")) ;
  }
}
```

- *Le gestionnaire de mise en forme* FlowLayout *tient compte, dans la mesure du possible, de la taille souhaitée des composants, qui peut être imposée par la méthode* setPreferredSize *de la classe* JComponent.

VII.8.3. Le gestionnaire CardLayout

Le gestionnaire de mise en forme CardLayout permet de disposer les composants suivant une pile, de telle façon que seul le composant supérieur soit visible à un moment donné.

Exemple

```
class MaFenetre extends JFrame {
  public MaFenetre () {
    super("Une fenetre") ; setSize(300, 200) ;
    Container contenu = getContentPane() ;
    CardLayout pile = new CardLayout(30, 20) ;
      //les arguments précisent les retraits entre le
      composant et le conteneur : 30 pixels de part et d'autre
      et 20 pixels en haut et en bas
    contenu.setLayout(pile) ;
      //changement de gestionnaire de mise en forme
    contenu.add(new JButton("UN"), "bouton1") ;
      //la chaîne "bouton1" permet d'identifier le composant
      au sein du conteneur
    contenu.add(new JButton("DEUX"), "bouton2") ;
    contenu.add(new JButton("TROIS"), "bouton3") ;
  } }
```

Par défaut, le composant visible est le premier ajouté au conteneur (dans l'exemple précédent, c'est le bouton UN). On peut faire apparaître un autre composant de la pile de l'une des façons suivantes :

```
pile.next(contenu) ;       //affiche le composant suivant
pile.previous(contenu) ; //affiche le composant précédent
pile.first(contenu) ;      //affiche le premier composant
pile.last(contenu) ;       //affiche le dernier composant
pile.show(contenu,"bouton3") ;
  //affiche le composant identifié par la chaîne "bouton3"
```

VII.8.4. Le gestionnaire GridLayout

Le gestionnaire de mise en forme GridLayout permet de disposer les composants les uns à la suite des autres sur une grille régulière, chaque composant occupant une cellule de la grille.

La classe GridLayout dispose de deux constructeurs :

```
public GridLayout(int rows, int cols) ;
  //rows et cols définissent respectivement le nombre de lignes et de colonnes de la grille.
public GridLayout(int rows, int cols, int hgap, int vgap) ;
  //hgap et vgap définissent les espaces entre les composants.
```

Les dernières cases d'une grille peuvent rester vides. Toutefois, si plus d'une ligne de la grille est vide, le gestionnaire réorganisera la grille, de façon à éviter une perte de place.

Exemple

```
class MaFenetre extends JFrame {
  public MaFenetre () {
    super("Une fenetre") ; setSize(300, 200) ;
    Container contenu = getContentPane() ;
    contenu.setLayout(new GridLayout(2, 2)) ;
      //changement de gestionnaire de mise en forme
    contenu.add(new JButton("UN")) ;
    contenu.add(new JButton("DEUX")) ;
    contenu.add(new JButton("TROIS")) ;
    contenu.add(new JButton("QUATRE")) ;
  }
}
```

Le gestionnaire de mise en forme BoxLayout permet de disposer des composants suivant une même ligne ou une même colonne, mais avec plus de souplesse que le gestionnaire GridLayout.

Le gestionnaire de mise en forme GridBagLayout, comme le gestionnaire GridLayout, permet de disposer les composants suivant une grille, mais ceux-ci peuvent occuper plusieurs cellules ; en outre, la taille des cellules peut être modifiée au cours de l'exécution.

VII.8.5. Un programme sans gestionnaire de mise en forme

Il est possible de n'associer aucun gestionnaire de mise en forme à un conteneur. Les composants sont alors ajoutés au conteneur à l'aide de la méthode setBounds de la classe Component.

Exemple

```
class MaFenetre extends JFrame {
   public MaFenetre () {
      super("Une fenetre") ; setSize(300, 200) ;
      Container contenu = getContentPane() ;
      contenu.setLayout(null) ;
         //changement de gestionnaire de mise en forme
      JButton bouton1 = new JButton("UN") ;
      contenu.add(bouton1) ;
      bouton1.setBounds(40, 40, 80, 30) ;
         //le coin supérieur gauche du bouton est placé au pixel
         de coordonnées 40, 40 par rapport au coin supérieur
         gauche de la fenêtre et les dimensions du bouton sont
         de 80 * 30 pixels
   }
}
```

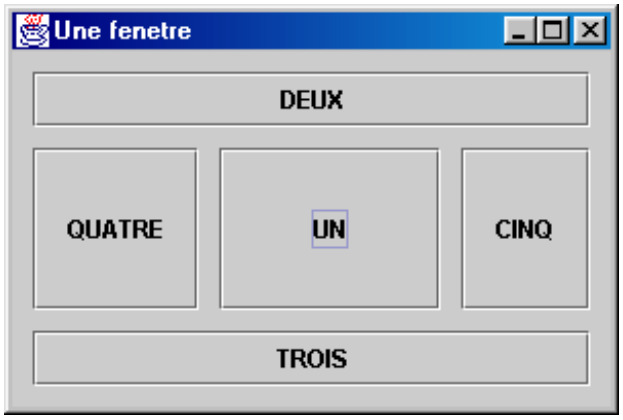
VII.8.6. Une classe Insets pour gérer les marges

La méthode getInsets de la classe Container permet de définir les quatre marges (haut, gauche, bas et droite) d'un conteneur. Elle retourne un objet de type Insets (classe du paquetage java.awt dérivée de la classe Object) défini par quatre champs publics de type entier initialisés par le constructeur suivant :

public Insets(int top, int left, int bottom, int right)

Exemple

```
class MaFenetre extends JFrame {
  public MaFenetre () {
    super("Une fenetre") ; setSize(300, 200) ;
    Container contenu = getContentPane() ;
    contenu.setLayout(new BorderLayout(10, 10)) ;
    contenu.add(new JButton("UN")) ;
    contenu.add(new JButton("DEUX"), "North") ;
    contenu.add(new JButton("TROIS"), "South") ;
    contenu.add(new JButton("QUATRE"), "West") ;
    contenu.add(new JButton("CINQ"), "East") ;
  }

  //redéfinition de la méthode getInsets afin de définir de
  nouvelles marges pour la fenêtre
  public Insets getInsets() {
    Insets normal = super.getInsets() ;
      //récupération des marges par défaut de la fenêtre
    return new Insets(normal.top+10, normal.left+10,
    normal.bottom+10, normal.right+10) ;
      //création d'un nouvel objet de type Insets pour
      modifier les marges de la fenêtre
  }
}
```


Une fenetre
DEUX
QUATRE
UN
CINQ
TROIS

VII.9. Dessinons avec Java

Java permet de dessiner sur n'importe quel composant grâce à des méthodes de dessin. Cependant, en utilisant directement ces méthodes, on obtient le dessin attendu mais il disparaîtra en totalité ou en partie dès que le conteneur du composant aura besoin d'être réaffiché (par exemple en cas de modification de la taille du conteneur, de déplacement, de restauration après une réduction en icône…). Pour obtenir la **permanence d'un dessin**, il est nécessaire de placer les instructions du dessin dans la méthode paintComponent du composant concerné. Cette méthode est automatiquement appelée par Java chaque fois que le composant est dessiné ou redessiné.

💥 *Ce problème de permanence ne se pose pas pour les composants d'un conteneur qui sont créés en même temps que le conteneur et restent affichés en permanence.*

Pour dessiner, on utilise, en général, un conteneur particulier appelé **panneau** de type JPanel.

```
java.lang.Object
 |
 +--java.awt.Component*
     |
     +--java.awt.Container*
          |
          +--javax.swing.JComponent*
               |
               +--javax.swing.JPanel                panneau utiliser pour dessiner
```

VII.9.1. Création d'un panneau

Un panneau de type JPanel est un composant intermédiaire qui peut être contenu dans un conteneur et qui peut contenir d'autres composants.

Un panneau est une sorte de "sous-fenêtre", sans titre ni bordure, qui doit obligatoirement être associé par la méthode add (de la classe Container) à un autre conteneur, généralement une fenêtre.

Exemple

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

class MaFenetre extends JFrame {
   private JPanel panneau ;
   public MaFenetre () {
      super("Une fenetre avec un panneau jaune") ;
      setSize(300, 200) ;
      panneau = new JPanel();
      panneau.setBackground(Color.yellow) ;
         //Color.yellow est une constante de la classe Color
         (classe du paquetage java.awt dérivée de la classe
         Object) correspondant à la couleur jaune
      getContentPane().add(panneau) ;
         //le panneau de couleur jaune occupe toute la fenêtre
   }
}

public class MonProgPanneau {
   public static void main(String args[]) {
      JFrame fen = new MaFenetre() ;
      fen.setVisible(true) ; }
}
```

Le gestionnaire de mise en forme FlowLayout est le gestionnaire par défaut des panneaux.

VII.9.2. Dessin dans un panneau

Pour obtenir un dessin permanent dans un composant, il faut redéfinir sa méthode paintComponent (méthode de la classe JComponent), qui sera appelée chaque fois que le composant aura besoin d'être redessiné. Cette méthode a l'en-tête suivant :

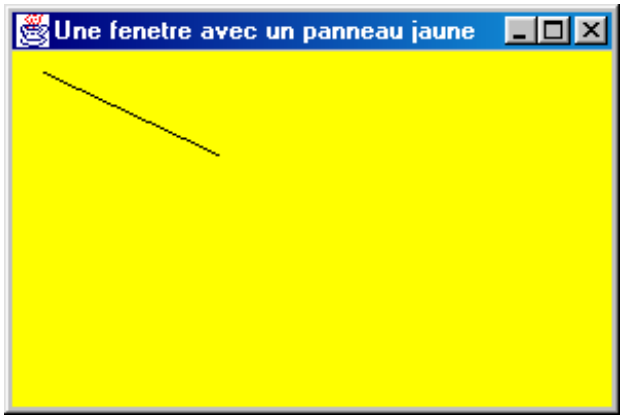
```
void paintComponent (Graphics g)
```

Son unique argument g de type Graphics est le **contexte graphique** du composant qui a appelé la méthode. Il sert d'intermédiaire entre les demandes de dessin et leur réalisation effective sur le composant.

Exemple

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

class MaFenetre extends JFrame {
   private Panneau pan ;
   public MaFenetre () {
      super("Une fenetre avec un panneau jaune") ;
      setSize(300, 200) ;
      pan = new Panneau();
      pan.setBackground(Color.yellow) ;
      getContentPane().add(pan) ;
   }
}

class Panneau extends JPanel {
   public void paintComponent(Graphics g) {
      super.paintComponent(g) ;
         //appel explicite de la méthode paintComponent de la
         classe de base JPanel , qui réalise le dessin du
         panneau.
      g.drawLine(15, 10, 100, 50) ;
         //méthode de la classe Graphics qui trace un trait du
         point de coordonnées (15, 10) au point de coordonnées
         (100, 50)
   }
}
```

```
public class PremDes {
   public static void main(String args[]) {
      JFrame fen = new MaFenetre() ;
      fen.setVisible(true) ;
   }
}
```



La méthode repaint (de la classe Component) permet d'appeler la méthode paintComponent.

Exemple

```
Panneau pan ;
pan.repaint() ;
   //appel de la méthode PaintComponent de l'objet pan de type Panneau
```

VII.9.3. La classe Graphics

La classe Graphics est une classe abstraite du paquetage java.awt dérivée de la classe Object. Cette classe dispose de nombreuses méthodes pour dessiner sur un composant et gère des paramètres courants tels que la couleur de fond, la couleur de trait, le style de trait, la police de caractères, la taille des caractères…

La méthode paintComponent fournit automatiquement en argument le contexte graphique du composant qui l'a appelée.

a. Paramétrer la couleur du contexte graphique

En Java, une couleur est représenté par un objet de type Color (classe du paquetage java.awt dérivée de la classe Object). Les constantes prédéfinies de la classe Color sont : Color.black, Color.white, Color.blue, Color.cyan, Color.darkGray, Color.gray, Color.lightGray, Color.green, Color.magenta, Color.orange, Color.pink, Color.red, Color.yellow.

Par défaut, dans la méthode paintComponent d'un composant, la couleur courante du contexte graphique correspondant est la couleur d'avant plan du composant (éventuellement modifiée par la méthode setForeground de la classe Component du composant). Dans la méthode paintComponent d'un composant, on peut :

récupérer la couleur du contexte graphique correspondant

```
public abstract Color getColor()
```

modifier la couleur du contexte graphique correspondant

```
public abstract setColor(Color c)
```

Les méthodes getColor et setColor sont des méthodes de la classe Graphics.

b. Paramétrer la police du contexte graphique

D'une manière générale, à un instant donné, un composant dispose d'une **police** courante de type Font (classe du paquetage java.awt dérivée de la classe Object) qui se définit par :

- un nom de famille de police (SansSerif, Serif, Monospaced, Dialog, DialogInput) ;
- un style : style normal (Font.PLAIN), style gras (Font.BOLD), style italique (Font.ITALIC), et style gras et italique (Font.BOLD+Font.ITALIC) ;
- une taille exprimée en points typographiques et non pas en pixels.

Exemple

```
Font f = new Font("Serif", Font.BOLD, 18) ;
```

Dans la méthode paintComponent d'un composant, on peut :

- récupérer la police du contexte graphique correspondant
  ```
  public abstract Font getFont()
  ```
- modifier la police du contexte graphique correspondant
  ```
  public abstract setFont(Font f)
  ```

Les méthodes getFont et setFont sont des méthodes de la classe Graphics.

c. Méthodes de dessin de la classe Graphics

public abstract void drawLine(int x1, int y1, int x2, int y2)
*//trace un trait du point de coordonnées (*x1*,* y1*) au point de coordonnées (*x2*,* y2*)*

public void drawRect(int x, int y, int largeur, int hauteur)
//dessine un rectangle de largeur largeur *et de hauteur* hauteur *au point de coordonnées (*x, y*)*

public void drawOval(int x, int y, int largeur, int hauteur)
//dessine un ovale de largeur largeur *et de hauteur* hauteur *au point de coordonnées (*x, y*)*

public abstract void drawstring(String str, int x, int y)
//dessine le texte str *au point de coordonnées (*x, y*)*

💥 *Toutes ces méthodes tracent des lignes en utilisant la couleur du contexte graphique correspondant.*

VII.9.4. Affichage d'images

Java sait traiter deux formats de stockage d'images : le format GIF (Graphical Interface Format) avec 256 couleurs disponibles et le format JPEG (Joint Photographic Expert Group) avec plus de 16 millions de couleurs disponibles.

Le chargement d'une image se fait en trois étapes.

- Tout d'abord, le constructeur de la classe ImageIcon (classe du paquetage javax.swing dérivée de la classe Object) permet de charger une image.
- Ensuite, la méthode getImage permet d'obtenir un objet de type Image à partir d'un objet de type ImageIcon. La méthode getImage de la classe Toolkit (classe du paquetage java.awt dérivée de la classe Object) permet de charger une image depuis un fichier local. La méthode getImage de la classe Applet permet de charger une image depuis un site distant. L'argument de cette méthode est un objet de type URL (classe du paquetage java.net dérivée de la classe Object), qui permet de représenter une adresse dans le réseau Internet (URL pour Uniform Reference Location).
- Enfin, la méthode drawImage de la classe Graphics permet d'afficher une image de type Image (classe du paquetage java.awt dérivée de la classe Object).

La méthode getImage n'attend pas que le chargement de l'image soit effectué pour rendre la main au programme. Il se pose alors le problème de l'affichage de l'image.
Afin de ne pas voir afficher qu'une partie de l'image, Java fournit en quatrième argument de la méthode drawImage de la classe Graphics la référence a un objet particulier appelé **observateur**. Cet objet implémente l'interface ImageObserver comportant une méthode imageUpdate qui est appelée chaque fois qu'une nouvelle portion de l'image est disponible.

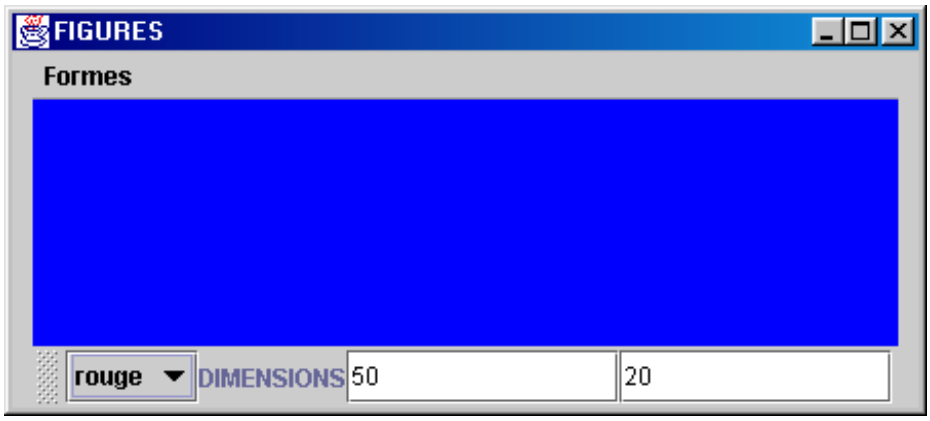
Tous les composants implémentent l'interface ImageObserver et fournissent une méthode imageUpdate qui appelle, par défaut, la méthode repaint. Ainsi, pour régler le problème de l'affichage de l'image, il suffira de fournir this en quatrième argument de la méthode drawImage.

Exemple

```
class MaFenetre extends JFrame {
   private Panneau pan ;
   public MaFenetre () {
      super("Une fenetre avec une image") ;
      setSize(300, 200) ;
      pan = new Panneau();
      getContentPane().add(pan) ; }
}

class Panneau extends JPanel {
   private ImageIcon rouge;
   public Panneau() {
      rouge = new ImageIcon("rouge.gif") ;
         //chargement d'une image dans l'objet de référence
         rouge, à partir du fichier "rouge.gif"
   }
   public void paintComponent(Graphics g) {
      super.paintComponent(g) ;
      Image imRouge = rouge.getImage();
         //la méthode getImage retourne une référence à un
         objet de type Image à partir d'un objet de type
         ImageIcon
      g.drawImage(imRouge, 15, 10, this) ;
         //affiche l'image imRouge au point de coordonnées
         (15, 10)
   }
}
```

TD14. Une interface graphique

Complétez la fenêtre du TD1 en lui ajoutant une zone de dessin.



Créez un nouveau projet qui contient une classe pour la fenêtre principale, une classe pour la zone de dessin et une classe pour le programme principal.

VIII. La gestion des évènements

VIII.1. Introduction

Un clic souris, la frappe d'une touche au clavier ou le changement de la taille d'une fenêtre sont des exemples d'événements.

💥 *Java classe les événements en deux niveaux : les événements de bas niveau (par exemple, un clic dans une fenêtre) et les événements de haut niveau (par exemple, une action sur un bouton qui peut provenir d'un clic souris ou d'une frappe au clavier).*

En Java, les événements n'ont pas une valeur physique, mais logique. Un événement dépend du composant qui l'a généré. On appelle **source** d'un événement l'objet qui l'a généré.

Exemple

L'événement émis suite à un clic souris dans une fenêtre est de type MouseEvent. .

L'événement émis suite à un clic souris sur un bouton est de type ActionEvent. .. .

Tout événement qui peut se produire dans une interface graphique est de type XXXEvent, classe du paquetage java.awt.event ou du paquetage javax.swing.event dérivée de la classe EventObject (classe du paquetage java.util dérivée de la classe Object). ..

💥 *Afin d'éviter d'avoir à s'interroger sur la répartition dans les paquetages des différentes classes utilisées dans les interfaces graphiques, nous importerons systématiquement toutes les classes des paquetages* java.awt, java.awt.event, javax.swing *et* javax.swing.event.

VIII.2. Traiter un événement

Un composant ne traite pas forcément lui même les événements qu'il génère. Il délègue ce traitement à des objets particuliers appelés **écouteurs** (un composant peut être son propre écouteur).

En fonction des événements qu'ils traitent, un écouteur doit implémenter une interface particulière, dérivée de l'interface EventListener, qui correspond à une catégorie d'événements. Pour traiter un événement de type XXXEvent, un écouteur doit implémenter l'interface XXXListener.

Exemple

L'interface MouseListener correspond à une catégorie d'**événements souris** de type MouseEvent. Elle comporte cinq méthodes correspondant chacune à un événement souris particulier..

```
public interface MouseListener extends EventListener {
  public void mousePressed(MouseEvent e) ;
    //appelé lorsqu'un bouton de la souris est pressé sur un composant
    //l'argument e de type MouseEvent correspond à l'objet événement généré
  public void mouseReleased(MouseEvent e) ;
    //appelé lorsqu'un bouton de la souris est relâché sur un composant
  public void mouseClicked(MouseEvent e) ;
    //appelé lors d'un clic souris sur un composant (la souris n'a pas été déplacée entre l'appui et le relâchement du bouton)
  public void mouseEntered(MouseEvent e) ;
    //appelé lorsque la souris passe de l'extérieur à l'intérieur d'un composant
  public void mouseExited(MouseEvent e) ; }
    //appelé lorsque la souris sort d'un composant (la souris passe de l'intérieur à l'extérieur du composant)
```

VIII.3. Intercepter un événement

Lorsqu'un composant veut intercepter un événement de type XXXEvent, il doit le préciser dans son constructeur en appelant la méthode addXXXListener(XXXListener objetEcouteur), où l'argument objetEcouteur correspond à l'objet écouteur chargé de traiter l'événement.

Pour savoir quels événements sont susceptibles d'être générés par un composant donné, il faut rechercher toutes les méthodes de la forme addXXXListener définies dans la classe du composant et dans ses classes ascendantes.

Exemple

La classe Component définit notamment les méthodes suivantes :

```
public void addFocusListener(FocusListener l) ;
  //prise et perte du focus (à un moment donné, un seul
  composant est sélectionné, on dit qu'il a le focus)
public void addKeyListener(KeyListener l) ;
  //événements clavier
public void addMouseListener(MouseListener l) ;
   //événements souris
public void addMouseMotionListener
(MouseMotionListener l) ;
  //événements liés au déplacement de la souris
```

VIII.4. Un premier exemple

VIII.4.1. Première version

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

class MaFenetre extends JFrame {
   public MaFenetre () {
      super("Une fenetre qui traite les clics souris") ;
      setSize(300, 200) ;
      addMouseListener(new EcouteurSouris());
         //la fenêtre fait appel à un écouteur d'événements souris
         pour traiter les clics souris
   }
}

   //L'écouteur d'événements souris doit implémenter l'interface
   MouseListener qui correspond à une catégorie
   d'événements souris.
class EcouteurSouris implements MouseListener {
      //redéfinition de la méthode appelée lors d'un clic souris
   public void mouseClicked(MouseEvent e) {
      System.out.println("clic dans la fenetre"); }
      //la redéfinition des autres méthodes est "vide"
   public void mousePressed(MouseEvent e) { }
   public void mouseReleased(MouseEvent e) { }
   public void mouseEntered(MouseEvent e) { }
   public void mouseExited(MouseEvent e) { }
}

public class MonProgEvtClic1 {
   public static void main(String args[]) {
      JFrame fen = new MaFenetre() ;
      fen.setVisible(true) ; }
}
```

VIII.4.2. Deuxième version

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

class MaFenetre extends JFrame implements MouseListener {
  public MaFenetre () {
    super("Une fenetre qui traite les clics souris") ;
    setSize(300, 200) ;
    addMouseListener(this);
      //la fenêtre est son propre écouteur d'événements souris
  }
    //L'argument e de type MouseEvent correspond à l'objet
    événement généré dans la fenêtre lors d'un clic souris. On
    peut utiliser les informations qui lui sont associées.
  public void mouseClicked(MouseEvent e) {
    int x = e.getX() ;
    int y = e.getY() ;
      //coordonnées du curseur de la souris au moment du clic
    System.out.println("clic dans la fenetre au point de
    coordonnees " + x + ", " + y);
  }
  public void mousePressed(MouseEvent e) { }
  public void mouseReleased(MouseEvent e) { }
  public void mouseEntered(MouseEvent e) { }
  public void mouseExited(MouseEvent e) { }
}

public class MonProgEvtClic2 {
  public static void main(String args[]) {
    JFrame fen = new MaFenetre() ;
    fen.setVisible(true) ;
  }
}
```

VIII.5. La notion d'adaptateur

Pour chaque interface XXXListener **possédant plusieurs méthodes**, Java fournit une classe particulière XXXAdapter, appelée **adaptateur**, qui implémente toutes les méthodes de l'interface avec un corps vide.

Pour définir un écouteur d'événements de type XXXEvent, il suffit alors de dériver l'écouteur de la classe XXXAdapter et de redéfinir uniquement les méthodes voulues.

Exemple

```
class MaFenetre extends JFrame {
    public MaFenetre () {
      super("Une fenetre qui traite les clics souris") ;
      setSize(300, 200) ;
      addMouseListener(new EcouteurSouris());
   }
}

class EcouteurSouris extends MouseAdapter {
     //redéfinition uniquement de la méthode appelée lors
     d'un clic souris
   public void mouseClicked(MouseEvent e) {
      System.out.println("clic dans la fenetre") ;
   }
}
```

VIII.6. Récapitulons

Pour traiter un événement de type inconnu (par exemple la fermeture d'une fenêtre) généré par une source, les étapes à suivre sont :

Rechercher les méthodes de la forme addXXXListener définies dans la classe de la source ou une de ses classes ascendantes.

Identifier l'interface XXXListener qui convient en regardant ses méthodes ; ce qui permet d'identifier le type XXXEvent de l'événement à traiter.

Exemple

L'interface WindowListener définit la méthode windowClosing appelée lors de la fermeture d'une fenêtre (la méthode addWindowListener est une méthode de la classe Window, classe ascendante de la classe JFrame).

Définir un écouteur pour traiter l'événement.

- L'objet source est son propre écouteur, il doit implémenter l'interface XXXListener adéquate.
- L'écouteur est une classe indépendante qui implémente l'interface XXXListener adéquate.
- L'écouteur est une classe indépendante qui dérive de la classe XXXAdapter associée à l'interface XXXListener adéquate.

Implémenter la ou les méthodes de l'interface XXXListener qui nous intéressent. Les informations contenues dans l'événement passé en paramètre de ces méthodes pourront être exploitées.

Exemple

```
class MaFenetre extends JFrame {
  public MaFenetre () {
    super("Une fenetre qui gere sa fermeture") ;
```

```
        setSize(300, 200) ;
        addWindowListener(new EcouteurFermer());
    }
}

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

VIII.7. Un exemple avec des boutons

Un bouton ne peut déclencher qu'un seul événement de type ActionEvent. L'interface ActionListener ne comporte qu'une seule méthode actionPerformed.

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

class MaFenetre extends JFrame implements ActionListener {
  private JButton MonBouton1, MonBouton2 ;
  public MaFenetre () {
    super("Une fenetre avec deux boutons") ;
    setSize(300, 200) ;
    Container contenu = getContentPane() ;
    contenu.setLayout(new FloxLayout()) ;
    MonBouton1 = new JButton("Bouton 1") ;
    contenu.add(MonBouton1) ;
    MonBouton2 = new JButton("Bouton 2") ;
    contenu.add(MonBouton2) ;
    //un même événement peut être traité par plusieurs écouteurs : deux écouteurs sont associés à l'action de l'utilisateur sur le bouton MonBouton1
    MonBouton1.addActionListener(this);
      //gère l'action de l'utilisateur sur le bouton MonBouton1
    MonBouton1.addActionListener(new EcouteurFermer());
      //gère la fermeture de la fenêtre lors d'une action de l'utilisateur sur le bouton MonBouton1
    MonBouton2.addActionListener(this);
      //gère l'action de l'utilisateur sur le bouton MonBouton2
  }


  public void actionPerformed(ActionEvent e) {
```

```
        //utilisation de la méthode getSource de la classe
        EventObject qui fournit une référence de type Object sur
        la source de l'événement concerné
        if(e.getSource() == MonBouton1)
          //conversion implicite du type JButton en un type
          ascendant Object
          System.out.println("action sur le bouton 1") ;
        if(e.getSource() == MonBouton2)
          System.out.println("action sur le bouton 2") ;
    }
}

class EcouteurFermer implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }
}

public class MonProgEvtBouton1 {
    public static void main(String args[]) {
        JFrame fen = new MaFenetre() ;
        fen.setVisible(true) ; }
}
```

La méthode getSource de la classe EventObject permet d'identifier la source d'un événement. Elle s'applique à tous les événements générés par n'importe quel composant.

La méthode getActionCommand de la classe ActionEvent permet d'obtenir la **chaîne de commande** associée à la source d'un événement. Les composants qui disposent d'une chaîne de commande sont les boutons, les cases à cocher, les boutons radio et les options menu. Par défaut, la chaîne de commande associée à un bouton est son étiquette.

Exemple

La méthode actionPerformed de la classe MaFenetre peut s'écrire :

```
public void actionPerformed(ActionEvent e) {
   String nom = e.getActionCommand() ;
   System.out.println("action sur le " + nom) ;}
```

La méthode setActionCommand de la classe JButton permet d'associer une autre chaîne de commande à un bouton.

Exemple

```
MonBouton1.setActionCommand ("premier bouton").
```

VIII.8. Un exemple de création dynamique de boutons

Jusqu'à présent, les composants d'un conteneur étaient créés en même temps que le conteneur et restaient affichés en permanence. Cependant, on peut, à tout moment, ajouter un nouveau composant à un conteneur (grâce à la méthode add de la classe Container) ou supprimer un composant d'un conteneur (grâce à la méthode remove de la classe Container). Si l'une de ses opérations est effectuée après l'affichage du conteneur, il faut forcer le gestionnaire de mise en forme à recalculer les positions des composants du conteneur (1) soit en appelant la méthode validate (de la classe Component) du conteneur, (2) soit en appelant la méthode revalidate (de la classe JComponent) des composants du conteneur.

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

class MaFenetre extends JFrame {
  private JButton MonBouton ;
  public MaFenetre () {
    super("Une fenetre avec des boutons dynamiques") ;
    setSize(300, 200) ;
    Container contenu = getContentPane() ;
    contenu.setLayout(new FlowLayout()) ;
    MonBouton = new JButton("Creation de boutons") ;
    contenu.add(MonBouton) ;
    MonBouton.addActionListener(new EcouteurBouton
    (contenu)) ;
  }
```

Un exemple de création dynamique de boutons

```
class EcouteurBouton implements ActionListener {
   private Container contenu ;
   public EcouteurBouton (Container contenu) {
      this.contenu = contenu ; }
   public void actionPerformed(ActionEvent e) {
      JButton NvBouton = new JButton("Bouton") ;
      contenu.add(NvBouton) ;
      contenu.validate() ;
         //recalcul les positions des composants du conteneur
   }
}

public class MonProgEvtBouton2 {
   public static void main(String args[]) {
      JFrame fen = new MaFenetre() ;
      fen.setVisible(true) ;
   }
}
```

VIII.9. Les classes internes et anonymes

Les classes internes et les classes anonymes sont souvent utilisées dans la gestion des événements des interfaces graphiques.

VIII.9.1. Les classes internes

Une **classe** est dite **interne** lorsque sa définition est située à l'intérieure d'une autre classe (ou d'une méthode).

Exemple

```
class MaFenetre extends JFrame {
   public MaFenetre () {
      super("Une fenetre qui gere sa fermeture") ;
      setSize(300, 200) ;
      addWindowListener(new EcouteurFermer());
   }
   //classe interne à la classe MaFenetre
   class EcouteurFermer extends WindowAdapter {
      public void windowClosing(WindowEvent e) {
         System.exit(0); }
   }      //fin classe interne
}
```

Un objet d'une classe interne est toujours associé, au moment de son instanciation, à un objet d'une classe externe dont on dit qu'il lui a donné naissance.

Un objet d'une classe interne a toujours accès aux champs et méthodes (même privés) de l'objet externe lui ayant donné naissance.

Un objet d'une classe externe a toujours accès aux champs et méthodes (même privés) de l'objet interne auquel il a donné naissance.

Exemple

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

class Cercle {
   private Centre c ;
   private double r ;
   class Centre {          //classe interne à la classe Cercle
      private int x, y ;
      public Centre(int x, int y) {
         this.x = x ; this.y = y ; }
      public void affiche() {
         System.out.println("( " + x + ", " + y + ")") ; }
   }      //fin classe interne
   public Cercle(int x, int y, double r) {
      c = new Centre(x, y) ;
         //création d'un objet de type Centre, associé à l'objet
         de type Cercle lui ayant donné naissance (celui qui a
         appelé le constructeur)
      this.r = r ; }
   public void affiche() {
      System.out.println("Cercle de rayon " + r + " et de
      centre ") ;
      c.affiche() ; }// méthode affiche de la classe Centre
   public void deplace(int dx, int dy) {
      c.x += dx ; c.y += dy ; }
         //accès aux champs privés x et y de la classe Centre
}

public class MonProgCercle {
   public static void main(String args[]) {
      Cercle c1 = new Cercle(1, 3, 2.5) ; c1.affiche() ;
      c1.deplace(4, -2) ; c1.affiche() ; }
}
```

Cercle de rayon 2.5 et de centre (1, 3)
Cercle de rayon 2.5 et de centre (5, 1)

VIII.9.2. Les classes anonymes

Une **classe anonyme** est une classe sans nom. Elle peut dériver d'une autre classe.

Exemple

```
class MaFenetre extends JFrame {
  public MaFenetre () {
    super("Une fenetre qui gere sa fermeture") ;
    setSize(300, 200) ;
    addWindowListener(new WindowAdapter() {
        //classe anonyme dérivant de la classe WindowAdapter
      public void windowClosing(WindowEvent e) {
        System.exit(0); }
    }) ;      //fin classe anonyme
  }
}
```

Une classe anonyme peut également implémenter une interface.

Exemple

```
class MaFenetre extends JFrame {
  private JButton UnBouton ;
  public MaFenetre () {
    super("Une fenetre avec un bouton") ;
    setSize(300, 200) ;
    UnBouton = new JButton("Un bouton") ;
    getContentPane().add(UnBouton) ;
    UnBouton.addActionListener(new ActionListener(){
        //classe anonyme implémentant l'interface ActionListener
      public void actionPerformed(ActionEvent e) {
        System.exit(0); }
    } ) ;      //fin classe anonyme
  }
}
}
```

TD15. La gestion des évènements

Gestion des événements d'une fenêtre

L'objectif de ce TD est de gérer les événements générés par chaque composant de la fenêtre du TD14, afin de prendre en compte les actions de l'utilisateur. Il vous faut donc gérer les événements générés par:

- les deux options boutons radio Rectangle et Ovale ;
- la boîte combo qui permet de choisir la couleur de fond du dessin ;
- les deux champs de texte qui permettent de saisir les dimensions de la forme (largeur et hauteur) à dessiner.

De plus, vous aurez à gérer la fermeture de la fenêtre principale avec la case système.

Créez un nouveau projet qui contient une classe pour la fenêtre principale, une classe pour la zone de dessin, une (ou des) classe(s) pour gérer les événements et une classe pour le programme principal.

Pour vous aider

Chaque action sur une option bouton radio *r* d'un groupe provoque :

- un événement Action et un événement Item pour *r* ;
- un événement Item pour l'option précédemment sélectionnée dans le groupe, si celle-ci existe et si elle diffère de *r*.

Une boîte combo génère des événements Item à chaque modification de la sélection.

Un champ de texte génère :

- un événement Action provoqué par l'appui de l'utilisateur sur la touche de validation (le champ de texte étant sélectionné) ;

un événement *perte de focus*, appartenant à la catégorie Focus, au moment où le champ de texte perd le focus, c'est à dire lorsque l'utilisateur sélectionne un autre composant (soit par la souris, soit par le clavier).

Bibliographie

Claude Delannoy. *Programmer en Java*. Eyrolles, 2000.

Cay S. Horstmann et Gary Cornell. *Au cœur de Java 2. Volume II : Fonctions avancées*. CampusPress France, 2000.

Gilles Clavel, Nicolas Mirouze, Emmanuel Pichon et Mohamed Soukal. *Java, la synthèse*. InterEditions, 1997.

Laura Lemay et Charles L. Perkins. *Le programmeur Java*. Simon & Schuster Macmillan (France), 1996.

http ://www.java.sun.com

Index

Index

Index

Index

N

Index