Table des matières

0.1 Préambule

Ce cours est destinée à des personnes connaissant déjà les bases de la programmation ainsi que les concepts objet. Par ailleurs, il est loin d'être exhaustif, mais a vocation à le devenir progressivement. Donc merci de signaler les manques constatés les plus importants. Pour tout complément, se référer au site web suivant :

`http://java.sun.com/products/jdk/1.1/docs/`

Chapitre 1

Les bases

1.1 introduction

1.1.1 Les concepts clefs

Ce que Java doit être :

- orienté objet : effet de mode ; facilite la programmation *propre*.
- propriétaire : pour éviter la prolifération de versions différentes ;
- universel ;
- adapté aux réseaux, et notamment à internet.

Conséquences : Java est un langage de programmation :

- **propriétaire, mais à licence libre** : c'est le compromis idéal pour avoir le contrôle d'un langage sans limiter sa diffusion.
- **à la syntaxe proche de celle de c++** : ainsi, tous les gens habitués à c++ sont sensés être rapidement opérationnels en Java ;
- **intégrant la notion de machine virtuelle** : un source Java est compilé en un pseudo-assembleur, appelé *byte code*, qui sera interprété sur la machine où il sera exécuté. Quel que soit la machine, le byte code est le même. Donc sans diffuser le source, on a un code exécutable sur toute machine sur laquelle Java est installé. Inconvénients : c'est très lent, et il faut que Java soit installé sur la machine cible.
- **assorti d'une importante bibliothèque standard** : ainsi, on est quasi assuré que tout ce qui est nécessaire à l'exécution d'un programme sera présent sur la machine cible. De plus, si l'on change d'environnement de programmation, on garde les mêmes bibliothèques.

- **doté de nombreuses fonctionnalités réseau (sockets, rmi, servlets, applets, etc.)** : il devient ainsi facile de télécharger des applications Java ou bien de faire communiquer des programmes Java s'exécutant sur des machines différentes comme s'il s'agissait de la même machine, voire du même processus.
- **assorti d'un mécanisme de vérification de code pour la sécurité** : on peut ainsi télécharger "sans risque" des applications via internet.

1.1.2 l'aspect orienté-objet de Java

Les plus

Comme tout bon langage orienté objet, il n'y a pas de pointeurs en Java, ce qui ne signifie toutefois pas que la notion de pointeur est absente, vue qu'elle est implantée au moyen des *références*.

Parmi les concepts objets intéressants de Java, on trouve bien évidemment, les notions de classe, d'héritage, de classe et méthode abstraites, de surcharge, de redéfinition, etc. Plus rarement présent dans les autres langages, on trouve aussi le concept d'*interface*. Une interface est un ensemble nommé de profils de méthode.

Les moins

L'héritage multiple n'existe pas en Java. Ce manque est partiellement pallié par la présence des interfaces.

Encore plus étonnant, il n'y a pas non plus de généricité dans Java. Mais là encore, les interfaces viendront à notre secours.

Contrairement à Eiffel, mais comme la plupart des autres langages, Java ne dispose pas de clauses telles que précondition, postcondition et invariant, permettant une lisibilité et une vérification du code plus aisée.

1.1.3 Un premier programme ?

Pour ne pas déroger à la règle, voici le classique "Hello World" en Java :

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

Figure 1.1: "Hello World" en Java

Conclusion : cela paraît incompréhensible et horriblement lourd pour écrire une simple phrase à l'écran. Effectivement, on fait ici appel à plusieurs concepts qui ne sont pas parmi les plus simples, notamment en Java (entrées/sorties,

méthodes de classe, nommages par exemple). Par conséquent, on oublie tout, et on part sur de nouvelles bases.

1.2 Les classes

Une classe *Toto* en Java est obligatoirement définie dans un fichier nommé *Toto.Java*. Par convention, les noms de classes commencent tous par une majuscule. La structure de base d'un tel fichier est la suivante (on verra par la suite que d'autres choses peuvent y figurer) :

[package *nompackage*]
[public] class *nomclasse* {
{*champ* || *méthode*}*
}

Un *package* est un nom donné à un ensemble logique de classes. Il existe un package par défaut contenant toutes les classes non déclarées comme membre d'un package particulier.

Une classe définie comme *public* est visible de toutes les autres classes. Sinon, seules les classes de son package peuvent y faire référence.

A partir d'ici, on peut comprendre la première ligne de l'exemple 1.1.

En Java, le corps d'une méthode est donné juste après son profil dans la déclaration même de la classe. Dans le cas des méthodes abstraites (ou virtuelles), par contre, aucun corps de méthode n'est donné.

1.2.1 Les variables

On distingue trois types de variables en Java :

- les variables d'instance ;
- les variables de classe ;
- les variables locales.

Le rôle de ces variables est similaire à celui qu'elles ont dans les autres langages orientés objets :

- Les variables d'instance correspondent à des caractéristiques propres à chaque objet de la classe. Elles sont déclarées comme des champs.
- Les variables de classe ont une valeur identique pour tous les objets de la classe : leur valeur est propre à la classe et non à l'objet. Elles sont déclarées comme des champs, mais avec le mot clef *static* en plus.
- les variables locales permettent de stocker temporairement des valeurs au sein d'une méthode. Elles sont déclarées dans le corps de la méthode.

N.B. : dans une large mesure, les paramètres des méthodes sont assimilables à des variables locales.

1.2.2 Déclaration d'une variable d'instance

Une variables d'instance est définie par sa visiblité, son type, son nom, éventuellement par une valeur par défaut, ainsi que par certaines caractéristiques particulières. Un exemple est le suivant :

```
private int abscisse = 0;
```

1.2.3 Visibilité

Il existe 4 types de visibilité :

- *private* : la variable n'est visible qu'à l'intérieur de la classe dans laquelle elle est déclarée ;
- *par défaut* : visible uniquement dans le package de la classe ;
- *protected* : visible dans le package et dans les sous-classes de la classe dans laquelle elle est déclarée ;
- *public* : visible partout.

En règle général, une variable d'instance ne devrait jamais être publique afin de respecter la notion d'encapsulation : les classes faisant référence à une autre ne doivent connaître que le profil et le rôle de ses méthodes, et non leur implantation. Les avantages sont sont les suivants :

- on peut changer la représentation interne de la classe sans que toutes les autres classes y faisant référence soit à refaire ;
- on peut contrôler les modifications apportées aux variables d'instance d'une classe.

1.2.4 Type d'une variable

Il existe trois genres de types en Java : les types de base, les classes et les tableaux. Suivant qu'il s'agit de l'un ou de l'autre, les variables joueront un rôle fort différent. Les types de base sont les suivants :

- boolean : vaut true ou false, false par défaut ;
- char : caractère stocké sur 2 octets pour l'aspect internationnal du langage ;
- byte : entier sur un octet ;
- short : entier sur deux octets ;
- int : entier sur quatre octets ;
- long : entier sur huit octets ;

- float : flottant sur octets ;
- double : flottant sur huit octets.

N.B. : tous les types numériques sont signés. De plus, les types *float* et *double* peuvent prendre des valeurs spéciales telles que NEGATIVE_INFINITY, POSITIVE_INFINITY et NaN (Not a Number).

Une variable dont le type est un des types de base représente directement la valeur que l'on souhaite lui voir contenir. Une variable dont le type est une classe contient par contre **nécessairement**[1] une référence à une zone mémoire qui, elle, contient l'objet proprement dit. Exemple :

Soit a et b deux variables de type *int*, et c et d deux variables de type *Toto*, Toto étant une classe contenant un champ x. Après exécution des instructions suivantes :

a = 3;
b = a;
b = 4;

a vaut 3 et b vaut 4.

Mais si l'on exécute la séquence suivante :

c.x = 3;
d = c;
d.x = 4;

Alors d.x vaut 4, mais c'est aussi le cas de c.x. En effet, le sens de la deuxième action est : "maintenant, d référence la même zone mémoire que c".

Les tableaux, bien que n'étant pas des objets, sont cependant gérés comme des variables dont le type serait une classe : une variable de type tableau est un référence sur la zone mémoire contenant le tableau, et non le tableau lui-même.

Déclaration des variables de classe

Les variables de classe se déclarent comme les variables d'instance, à savoir dans la définition d'une classe, mais on rajoute le mot clef `static` dans leur déclaration. Exemple :

```
public static int var;
```

Les variables locales

Le profil de leur déclaration est similaire à celui des variables d'instance. La seule différence est dans le lieu de leur déclaration, qui se situe n'importe où dans le corps d'une méthode.

N.B. : contrairement à c++, on ne peut utiliser le mot clef `static` pour caractériser des variables locales *rémanentes*.

[1] Contrairement à c++

1.2.5 Vie et mort des objets

La seule manière de créer un objet d'une classe donnée est d'appeler un *constructeur* de cette classe. Un constructeur est une méthode portant un nom strictement identique à la classe à laquelle il appartient. Par défaut, toute classe contient un constructeur ne prenant aucun paramètre et ne faisant rien. L'appel à un constructeur se fait au moyen de l'instruction `new`.

Exemple :

```
public class Toto {
    private int x = 0;
    public Toto(int v) {
       x = v;}
}
 Toto a = null;
 a = new Toto(); //a.x vaut 0
 Toto b = new Toto(15); //b.x vaut 15
```

Remarque : pour préciser clairement qu'une variable ne référence aucun objet, il est conseillé de l'initialiser à la valeur `null`.

N.B. : dans le cas d'une sous-classe, le constructeur de la super-classe est appelé soit manuellement, soit automatiquement (voir la partie concernant l'héritage). C'est ce qu'on appelle le chaînage des constructeurs.

Les objets sont détruits automatiquement par le *ramasse-miette*. Celui-ci attend bien évidemment qu'un objet ne soit plus référencé avant de le détruire. Si une méthode `finalize()` est présente dans la classe de l'objet, alors elle est appelée lorsque sa destruction a lieu. Pour faire accélérer la libération de l'espace pris par un objet devenu inutile, il peut être intéressant d'affecter la valeur `null` à la dernière variable le référençant.

N.B. : contrairement aux constructeurs, les *finaliseurs* ne sont pas chaînés automatiquement.

1.2.6 Les méthodes d'instance

Une méthode d'instance est une méthode (fonction) qui sera toujours appliquée à un objet particulier (une instance) de la classe dans laquelle elle est définie. Toute méthode est déclarée en même temps que sa définition est donnée. Un profil d'une méthode est en général de la forme :

visibilité typeRetour nomMéthode Paramètres

La liste de paramètres peut être vide. Contrairement à ce que le langage C autorise, par contre, une méthode Java prend toujours un nombre fixe de paramètres. Il est toutefois possible de définir au sein d'une même classe plusieurs méthodes de même nom mais avec des paramètres différents : c'est ce qu'on appelle la **surcharge**.

Dans le profil d'une méthode, un paramètre est déclaré sous la forme *Type Nom*.

1.2.7 Les méthodes de classe

Tout comme on peut définir des variables de classe, il est aussi possible de définir des méthodes de classe au moyen du mot clef `static`. Une méthode de classe ne peut bien évidemment pas référencer des variables d'instance de la classe dans laquelle elle est définie, puisqu'elle n'est pas rattachée à un objet. Elle peut par contre lire et modifier les variables de classe, même si ce n'est pas obligatoire. Il existe une méthode de classe particulière, la méthode `main` : celle-ci est celle qui est exécutée lorsqu'un programme Java commence. La deuxième ligne de l'exemple de la figure 1.1 s'éclaircit peu à peu...

1.2.8 Utiliser les variables et méthodes d'instance et de classe

Soit vi une variable d'instance de la classe C et vc une variable de classe de cette même classe C. Alors dans le corps des méthodes de C, on pourra faire référence à ces variables en utilisant directement leurs noms, à savoir vi et vc. Exception : si une variable locale s'appelle aussi vi, alors vi référence cette variable locale. On pourra alors faire référence à la variable d'instance vi en utilisant $this.vi$. Dans les autres classes, si ces variables sont visibles, il faudra préciser à quoi elles se réfèrent. Ainsi, pour une variable d'instance, il faut indiquer l'objet dont la valeur de cette variable nous intéresse. Par exemple, si o est un tel objet, on fera $o.vi$. La variable de classe n'étant quant à elle attachée à aucune objet, mais à la classe, on l'utilisera ainsi : $C.vc$. L'appel au méthodes d'instance s'effectue de la même manière que pour les variables d'instance. De même, les méthodes de classe s'utilisent en préfixant leur nom par le nom de la classe à laquelle elles appartiennent.

1.2.9 Des variables particulières : les constantes

En Java, toute variable non locale est membre d'une classe, soit sous la forme d'une variable d'instance, soit sous la forme d'une variable de classe. Par conséquent, une constante sera définie comme une variable de classe. De plus, pour préciser que sa valeur ne peut pas être changée, on rajoutera à sa déclaration le mot clef `final`.

Exemple :

```
public static final int ZERO = 0;
```

1.3 Les chaînes de caractères

Contrairement au langage C (et dans une large mesure à C++), les chaînes de caractères ne sont pas gérées comme un tableau de caractères, mais comme des objets d'une classe particulière : la classe `java.lang.String`. Cette classe à une particularité : une fois une chaîne de caractères créée, elle ne peut pas être

modifiée. Si l'on souhaite modifier une chaîne de caractères, soit on en crée une autre, soit on passe par la classe `java.lang.StringBuffer`.

Pour les deux classes, la position des caractères est numérotée à partir de 0.

1.3.1 La classe `String`

Les créateurs de cette classe sont multiples. De plus, de nombreuses méthodes renvoient directement un nouvel objet de type `String`. Voici quelques-uns des constructeurs de la classe :

- `String()` : crée une chaîne vide ;
- `String(String)` : crée une chaîne identique à celle passée en paramètre ;
- `String(StringBuffer)` : pour la conversion de la classe `StringBuffer` à la classe `String` ;
- `String(char [])` et `String(char [], int, int)` : pour convertir un tableau ou un sous-tableau de caractères en chaîne ;
- `String(byte [])` et `String(byte [], int, int)` pour convertir un tableau d'octets en chaîne de caractères en utilisant le codage des caractères par défaut associé à la plateforme.

Passons maintenant aux méthodes essentielles. Comme un objet de type `String` ne peut pas être modifié, une méthode telle que `concat()` renvoie en fait une nouvelle chaîne :

- `equals(Object)` : pour comparer deux chaînes ;
- `equalsIgnoreCase(String)` : pour comparer deux chaînes, en ignorant les différences minuscules/majuscules ;
- `compareTo(String)` : renvoie -1, 0 ou 1 suivant l'ordre lexicographique des deux chaînes à comparer ;
- `indexOf(String)` et `indexOf(String, int)` : renvoie l'index de la première occurence d'une sous-chaîne, en commençant éventuellement à une position donnée.
- `lastIndexOf(String)` et `lastIndexOf(String, int)` : renvoie l'index de la dernière occurence d'une sous-chaîne, en commençant éventuellement à une position donnée.
- `length()` : la longueur de la chaîne ;
- `replace(char,char)` : renvoie une nouvelle chaîne en changeant toutes les occurences d'un caractère par un autre ;
- `startsWith(String)` et `endsWith(String)` : pour savoir si une chaîne commence par ou finit par une certaine chaîne ;

- `substring(int, int)` : pour extraire une sous-chaîne comprise entre le premier paramètre et le caractère précédant le deuxième paramètre ;
- `concat(String)` : pour concaténer deux chaînes ; Il est aussi possible d'utiliser l'opérateur `+` ;
- `toLowerCase()` et `toUpperCase()` : pour convertir en minuscules ou majuscules ;
- `trim()` : pour supprimer les espaces en tête et en queue de chaîne ;
- `valueOf(`*type*`)` : pour convertir une donnée d'un des types de base en sa représentation sous la forme d'une chaîne de caractères.

1.3.2 La classe `StringBuffer`

Les objets de type `StringBuffer` sont créés avec de la place pour 16 caractères de plus que leur valeur initiale. Au fur et à mesure des besoins, de l'espace mémoire leur est dynamiquement réalloué.

- `StringBuffer()` : crée une chaîne vide ;
- `StringBuffer(String)` : initialisation à partir d'une chaîne ;
- `StringBuffer(int)` : réservation d'une certaine place dès le départ (à utiliser principalement pour des raisons d'efficacité).

Les méthodes essentielles de la classe en question sont :

- `append(`*type*`)` : pour un ajout en fin de chaîne d'un objet de type *type*;
- `insert(int,` *type*`)` : pour l'insertion en milieu de chaîne ;
- `length()` : pour connaître la longueur de la chaîne ;
- `reverse()` : pour renverser l'ordre des caractères ;
- `setCharAt(int, char)` : pour modifier un caractère ;
- `capacity()` : pour connaître la taille du buffer réservé pour la chaîne ;
- `ensureCapacity(int)` : si la taille du buffer est inférieure au paramètre, l'augmente pour que ce ne soit plus le cas ;
- `toString()` : pour convertir en chaîne de caractères de type `String`.

1.3.3 Conversion d'une chaîne en un type de base

A chaque type de base est associée une classe encapsulant une variable de ce type. Par exemple, il existe une classe `java.lang.Integer` encapsulant une variable de type `int`. Ces classes regroupent notamment des méthodes de classe associées au type en question. On trouve notamment dans chacune de ces classes une méthode `parse`*Type*`(String)`. Cette méthode permet de convertir la chaîne de caractères en paramètre dans le type en question.

Exemple :

```
String chaine = "12";
int entier;
entier = Integer.parseInt(chaine);
entier *= 2;
```

1.4 Utilisation des tableaux

Un tableau peut être déclaré selon l'une des deux syntaxes suivantes :

```
Toto monTableau[];
Toto[] monAutreTableau
```

Bien que n'étant pas un objet à proprement parlé, un tableau s'initialise comme un objet (par `new`) et sa place est libérée automatiquement par le ramasse-miette lorsqu'il n'est plus référencé. Exemple :

```
monTableau = Toto[10];
```

La ligne précédente permet de créer un tableau de 10 cases (numérotées de 0 à 9), chacune de celles-ci étant une référence à un objet de type Toto. Lors de l'initialisation d'un tableau, les cases sont remplies par la valeur par défaut du type des éléments (pour des objets, il s'agit de la valeur `null`). Il est aussi possible de créer un tableau en donnant directement son contenu :

```
int[] monTroisiemeTableau = {2, 3, 5, 7};
```

Le nombre d'éléments d'un tableau T peut être connu grâce au champ *length* du tableau, champ utilisable exclusivement en lecture de la manière suivante : $T.length$. De plus, les accès à un tableau sont toujours contrôlés (en cas d'accès hors des bornes, une exception - voir le chapitre en question - `ArrayIndexOutOfBoundsException` est générée).

1.4.1 Les tableaux à plusieurs dimensions

Un tableau à n dimensions ($n > 1$) est en fait un tableau de tableaux à $n - 1$ dimensions. Lorsque l'on crée un tel tableau, la taille de la première dimension ainsi que des m suivantes (m étant quelconque mais devant vérifier $0 \leq m \leq n - 1$), doivent être précisées. Il n'est pas possible de préciser la taille de la dimension j ($j > 1$) si celle de la dimension $j - 1$ n'a pas été donnée, comme le montre les exemples suiva5nts :

```
int[][][] monTroisiemeTableau = new int[10][][];//valide
int[][][] monTroisiemeTableau = new int[10][5][];//valide
int[][][] monTroisiemeTableau = new int[4][5][7];//valide
```

```
int[][][] monTroisiemeTableau = new int[10][][7];//erreur
int[][][] monTroisiemeTableau = new int[][][];//erreur
```

De même qu'un tableau à une dimension, un tableau à plusieurs dimensions peut être créé en donnant directement son contenu comme dans l'exemple suivant :

```
int[][] doub = {{1,2},{2,4},{3,6}};
```

N.B. : comme un tableau à plusieurs dimensions est en fait un tableau de tableaux, chacun des sous-tableaux peut avoir une taille variable. Et donc l'exemple suivant est lui aussi valide :

```
int[][] biz = {{1,2},{2,4,6},{3,6,9,12}};
```

On comprend maintenant mieux le profil de la méthode `main` de l'exemple Hello World : la méthode main prend en paramètre un tableau de chaînes de caractères, chaque case du tableau référençant un des arguments passés sur la ligne de commande.

1.5 Les opérateurs en Java

On retrouve quasiment les mêmes opérateurs qu'en c/c++, à savoir :

- incrémentation et décrémentation : ++, --
  - post-incrémentation : $i++$
  - pré-incrémentation : $i--$
- les opérateurs classiques : +, -, *, /
- modulo : %
- concaténation de chaînes de caractères : +
- décalages à gauche, à droite avec signe, à droite avec 0 : <<, >>, >>>
- compatibilité de classe : `instanceof`
- comparaisons (égalité, différence) : ==, !=
- complément logique : !
- "et", "ou", "ou exclusif" bit-à-bit (sur entiers et booléens) : &, ||, ^
- "et", "ou" conditionnels (sur booléens) : &&, ||||
- si-alors-sinon : `?  :`
- l'affectation : =

Cas particulier du *et* et du *ou* logiques : il existe deux opérateurs pour le *et* logique : & et &&. Le premier correspond au vrai *et* logique : chacun des opérandes est évalué avant de conclure sur la valeur logique du résultat. Le deuxième opérateur est en fait un *et* séquentiel : dès qu'un des opérandes est évalué à *faux*, on s'arrête (on sait alors que le résultat sera forcément *faux*). La différence entre le || et le ||| est similaire : le deuxième arrête l'évaluation dès qu'un des opérandes est évalué à *vrai* (le résultat sera alors forcément *vrai*).

Tout comme en c/c++, certains de ces opérateurs on des "raccourcis d'affectations". Ainsi, `a += 2;` est équivalent à `a = a + 2`. Les raccourcis légaux sont les suivants :

+=, -=, *=, /=, %=, <<=, >>=, >>>=, &=, ^=, || =

1.6 Les instructions

1.6.1 Si - Alors - Sinon

Ce bloc conditionnel peut être présent selon deux versions :

```
if (condition)
    bloc-o
if (condition)
    bloc-oui
else
    bloc-n
```

un *bloc* est constitué soit d'une instruction, soit d'une séquence d'instructions entre accolades. Dans les deux cas, si `condition` est évalué à *vrai*, `bloc-oui` est exécuté, puis on continue après la fin du bloc conditionnel. Si par contre `condition` est évalué à *faux*, dans le premier cas, on continue après le bloc conditionnel, tandis que dans le deuxième cas, on exécute d'abord `bloc-non` avant de passer à la suite.

N.B. : Java ne lève pas l'ambiguité du conflit "shift-reduce" que présentait la grammaire du c.

1.6.2 Opérateur de choix

Comme en c, il existe l'opérateur `switch`, qui permet de proposer différents choix selon la valeur d'une variable :

```
switch(i) {
    case 1 : bloc-1 ;
    case 2 : bloc-1-2; break
    case 4 : bloc-4;break
    default : bloc-autre}
```

L'exemple précédent s'interprète comme suit :

- si i vaut 1, on exécute `bloc-1` puis `bloc-1-2` avant de sortir ;

- si i vaut 2, on exécute juste `bloc-1-2` ;
- si i vaut 4, on exécute `bloc-4` ;
- dans tous les autres cas, on exécute `bloc-autre`.

1.6.3 L'instruction while

Cette instruction se présente sous la forme suivante :

```
while (condition) bloc-boucle
```

Et elle s'interprète comme ceci : si `condition` est vrai, alors exécuter `bloc-boucle` puis recommencer. Sinon, poursuivre après l'instruction `while` sans exécuter `bloc-boucle`.

1.6.4 L'instruction do

1.6.5 La boucle for

1.6.6 break et continue

Ces deux instructions permettent de sortir prématurément d'un bloc de boucle ou de test. Dans le cas du `break`, on quitte le bloc, tandis que dans le cas du `continue`, on passe à l'itération suivante. Ces instructions se réfèrent toujours à la boucle la plus interne. Si on veut sortir de plusieurs niveaux d'imbrication, il faut utilser les formes étiquetées de ces instructions, comme le montre l'exemple, qui affiche les nombres compris entre 2 et 100 qui sont des carrés parfaits[2] :

[2] le programme donné n'est évidemment pas le meilleur pour une telle fonction

```
int max = 100;
int x = 2;
boucle1 :  for (x = 0 ; x <= max ; x++)
   {
   if (x % 2 == 0) continue ;
   for (int i = 0 ; i < x ; i++)
      if (i * i > x) continue boucle1
      else if (i * i == x)
         {System.out.println(x); break;}
   }
```

Chapitre 2

Les concepts objets

2.1 L'héritage

2.1.1 Syntaxe

Nous avions déjà vu, dans le chapitre précédent, comment définir des classes *ex nihilo* en Java. Mais Java étant un langage objet, il est bien évidemment possible de construire des classes par héritage. Ceci se fait au moyen du mot clef `extends`. Toute classe Java que l'utilisateur définit hérite d'une autre classe. Si cela n'est pas fait explicitement, alors il s'agira de la classe `Object`, dont toute classe hérite directement ou indirectement. Exemple :

```
public class GereX {
   public int x = 0;
   public void setx(int v){
      x = v;
   }
   public int getx(){
      return x;
   }
}
public class GereY extends GereX{
   public int y = 0;
   public void sety(int v){
      y = v;
   }
   public int gety(){
      return y;
   }
}
```

Ainsi, la classe `GereY` hérite de la classe `GereX`. La séquence d'instruction suivante est alors tout à fait valide :

```
GereX a = new GereX();
```

```
GereX b;
GereY c = new GereY();
int d;
a.setX(10);
d = a.getX();
c.setY(20);
c.setX(15);
b = c;
b.setX(7);
d += c.getX()+c.getY();
```

Question : que vaut alors `d` ?

Rappel : il n'est évidemment pas possible de faire, à la suite du code suivant :

```
c = b;
```

Si une telle opération était à faire, il faudrait procéder ainsi :

```
c = (b instanceof gereY)? (gereY) b :  null;
```

En Java, contrairement à c++, il n'y a pas d'héritage multiple. Autrement dit, une classe ne peut hériter que d'une seule autre classe.

2.1.2 Création et Destruction d'objets

Comme cela avait été mentionné dans la partie consacrée à la vie des objets, les constructeurs de l'arbre d'héritage sont automatiquement chaînés si cela n'est pas fait explicitement (au moyen du mot-clef `super`), mais ce n'est pas le cas des finaliseurs.

Ainsi, supposons que l'on rajoute à la classe GereX précédente les constructeurs suivants :

```
GereX() {   x = 5;
}
GereX(int v) {
setx(v);
}
```

Et que l'on rajoute à la classe GereY les constructeurs :

```
GereY() {
y = 10;
}
GereY(int v) {
super(v);
sety(v);
}
```

Si l'on exécute les instructions suivantes :

```
GereY b,c;
b = newGereY();
c = newGereY(4);
int e,f,g,h;;
```

```
e = b.getX();f=b.getY();
g = c.getX();h=c.getY();
```

on aura e = 5, f = 10, g = 4, h = 4.

Remarque : pour que le chaînage explicite des constructeurs soit détecté, il est impératif que l'appel au constructeur de la super-classe soit la première instruction du constructeur de la sous-classe.

2.1.3 Redéfinition de méthodes et liaison dynamique

Ces concepts sont implantés en Java de manière similaire à ce que l'on trouve par exemple en c++. Ainsi, si on rajoute à la classe GereY la méthode suivante (N.B. : noter l'utilisation de `super` pour faire référence à une méthode de la super-classe redéfinie dans la classe courante) :

```
setx(int v) {
super.setx(2*v);
sety(v);
}
```

Et que l'on exécute le code suivant :

```
GereX a,b;
GereY c;
a = newGereX();
c = newGereY();
b = c;
a.setx(4);
b.setx(3);
int e,f;
e = a.getx();
f = b.getx();
```

Alors e vaudra 4 et f vaudra... 6.

2.2 Classe et méthode abstraites (ou virtuelle)

Une méthode abstraite est une méthode dont la définition est reportée à ultérieurement dans l'arbre d'héritage. Toute classe comportant au moins une méthode abstraite est une classe abstraite. Il n'est pas possible de créer directement des instances d'une telle classe. Seules des instances des sous-classes donnant une définition *concrète* de toutes les méthodes abstraites de cette classe pourront être créées.

Exemple de classe abstraite :

```
public abstract class Clab {
   private int x, int y;
   public bool sym() {
      return (x = y);
   }
   public abstract calc();
```

```
}
```

2.3 Classe finale

Une classe finale est une classe qui ne peut avoir de sous-classe. Elle se définit au moyen du mot clef `final` ainsi :

```
public final class Clab {
   private int x, y;
   public bool sym() {
      return (x = y);
   }
}
```

N.B. : il va de soi qu'une classe finale abstraite n'a aucun sens.

2.4 Les classes internes

2.5 Les interfaces

2.5.1 Définition

Une interface est un ensemble nommé de profils de méthodes. On peut donc voir une interface comme une classe abstraite ne comportant que des méthodes abstraites (et donc pas de variables).

Exemple de définition d'une interface :

```
public interface Comp {
   public bool egal(Object o2);
   public bool leq(Object o2);
   public bool lt(Object o2);
```

```
}
```

Comme pour les classes, il est possible de faire hériter une interface d'une autre interface. Par contre, contrairement aux classes, l'héritage multiple d'interfaces est autorisé.

2.5.2 Implantation

Une interface peut être *implantée* par une classe. Cela impose que toutes les méthodes de l'interface soient définies dans la classe en question. Exemple :

```
public class Entier {
   public int val;
   public bool egal(Object o2) {
      if (o2 instanceOf Entier)
      else)
         return (Entier o2).val == o1.val)
   public bool leq(Object o2) {
   public bool lt(Object o2) {
}
```

Chapitre 3

Quelques aspects particuliers

3.1 Les exceptions

3.1.1 Généralités

Les exceptions permettent de séparer le code associé à la gestion des erreurs du code *effectif*. Une exception se traduit en Java par un objet d'une classe particulière héritant de la classe `Exception`. Il existe des exceptions déjà présentes dans le langage, mais il est possible de créer ses propres exceptions. Lorsqu'on fait appel à une méthode pouvant générer une exception, on doit soit traiter l'exception, soit la passer à la méthode appelante. Dans ce dernier cas, il faut signaler que l'on est susceptible de renvoyer le type d'exception en question.

Il y a des exceptions à ce dernier point : les exceptions qui sont des sous-classes de `Error` et `RuntimeException`, qui n'ont pas besoin d'être déclarées lorsqu'elles ne sont pas gérées. C'est le cas notamment de l'exception `ArrayIndexOutOfBoundsException`, levée lorsqu'il y a un accès à un tableau hors de ses bornes.

3.1.2 Traiter des exceptions

Voici un exemple de traitement des exceptions : si on passe 4 arguments (ou plus) à la ligne de commande, aucune exception n'est levée. Dans le cas contraire, l'exception levée dans le bloc `try` de la méthode `somme()` est prise en compte par l'instruction `catch`.

```
import java.lang.*;
public class TestException {
   public int[] monTableau;
   public int somme()
   {
      int i;
      int s = 0;
      try
```

```
        {
           for (i = 0 ; i < 4;i++)
           {
              s += monTableau[i];
           }
        }
        catch (ArrayIndexOutOfBoundsException e)
        {
           System.out.println("Erreur d'accès au tableau");
        }
        return s;
     }
     public static void main(String[] args)
     {
        int i,s;
        TestException t = new TestException();
        t.monTableau = new int[args.length];
        for (i = 0 ; i < args.length; i++)
        {
           t.monTableau[i] = Integer.parseInt(args[i]);
        }
        s = t.somme();
        System.out.println(s);
     }
  }
```

3.1.3 Lever une exception

Pour lever (déclencher) une exception, on utilise la commande throws suivi d'une référence à un objet appartenant à une sous-classe de la classe `Exception`.

Exemple :

```
throw new ArithmeticException("probleme de math");
```

3.1.4 Transmettre une exception

Nous avions vu qu'une méthode qui ne traitait pas une exception qu'elle pouvait générer directement ou indirectement (c'est-à-dire qui pouvait être générée par une des méthodes qu'elle appelait) devait obligatoirement indiquer qu'elle était susceptible de renvoyer ce type d'exception (sauf... "exception"). Ceci ce fait grâce au mot-clef `throws` dans le profil de la méthode. Exemple :

```
public maMethode() throws ArithmeticException { ...}
```

3.2 Les threads

3.3 Les RMI

3.4 Les composants (JavaBeans)

Chapitre 4

Les entrées-sorties

En Java, les entrées, comme les sorties, sont traitées sous la forme de flux.

4.1 Les entrées-sorties standards

Elles sont implantées au moyen de variables de classe de la classe `java.lang.System` :

- `in` : entrée standard, de type `java.io.InputStream` ;
- `out` : sortie standard, de type `java.io.PrintStream` ;
- `err` : sortie erreur standard, de type `java.io.PrintStream` ;

Les entrées-sorties standards peuvent être modifiées au moyen des méthodes suivantes de la classe `System` :

- `setIn(InputStream)` ;
- `setOut(PrintStream)` ;
- `setErr(PrintStream)` ;

4.1.1 La classe `PrintStream`

Cette classe existe principalement par compatibilité avec les versions précédentes du langage, mais son usage est à éviter. Il est plutôt conseillé d'utiliser la nouvelle classe `PrintWriter`.

Elle hérite de la classe `FilterOutputStream` qui elle-même hérite de `OutputStream`.

On peut distinguer quatre noms de méthodes :

- `print()` : écrit le paramètre. La méthode est définie pour tous les types de base, le type `String`, le type `char []` et aussi le type `Object` ;

- `println()` : idem que `print()`, mais avec un retour à la ligne après ;
- `flush()` : pour forcer l'écriture du tampon ;
- `close()` : pour fermer le canal.

4.1.2 La classe `PrintWriter`

Il s'agit d'une classe héritant de `Writer` et définissant notamment les méthodes décrites pour la classe `PrintStream`. On trouve cependant quelques méthodes supplémentaires :

- `PrintWriter(OutputStream)`, `PrintWriter(Writer)`, `PrintWriter(OutputStream, boolean)`, `PrintWriter(Writer, boolean)` : le booléen sert à indiquer s'il y a vidage automatique du tampon en fin de ligne (par défaut, ce n'est pas le cas).
- `write(char[], int,int)` : pour écrire un sous-tableau de caractères ;
- `write(String, int,int)` : pour écrire une sous-chaîne de caractères ;

4.1.3 La classe `InputStream`

Il s'agit d'une classe abstraite dont héritent les différentes classes permettant de gérer les entrées. Elle définit notamment les méthodes suivantes :

- `close()` : pour fermer le canal ;
- `read()` : pour lire l'octet suivant ;
- `read(byte[])` : lit n octets, où n est la taille du tableau passé en paramètre ;
- `read(byte[], int, int)` : le premier entier est l'indice à partir duquel remplir le tableau, le deuxième est le nombre d'octets à lire ;
- `skip(long)` : pour sauter des caractères ;
- `mark(int)` et `reset()` : marquer une position pour pouvoir y revenir plus tard. Le paramètre de `mark()` précise le nombre maximum de caractères pendant la lecture desquels la marque demeure valable.

4.1.4 La classe `InputStreamReader`

Cette classe hérite de la classe `java.io.Reader`. Elle permet d'interpréter un flux comme un flux de caractères au lieu d'un flux d'octets. Elle implante notamment les méthodes suivantes :

- `InputStreamReader(InputStream)`, `InputStreamReader(InputStream, String)` : constructeurs. Le deuxième paramètre permet de préciser quel type de codage est à utiliser pour convertir les octets en caractères (le premier constructeur utiliser le codage par défaut de la plateforme).
- `close()` pour fermer le canal ;
- `getEncoding()` pour connaître le nom du codage utilisé ;
- `read()` pour lire un caractère ;
- `read(char[], int, int)` pour lire un certain nombre de caractères (troisième argument) à insérer dans un tableau (premier argument) à partir d'un indice donné (deuxième argument) ;
- `ready()` : pour savoir s'il y a des caractères à lire.

4.1.5 La classe `BufferedReader`

Cette classe permet de lire plus aisément un flux d'entrée en mode caractères, notamment grâce à la méthode `readLine()`.

Les méthodes disponibles sont les suivantes :

- `close()` : pour fermer le canal ;
- `mark(int)` et `reset()` : pour marquer une position et éventuellement y revenir (si on n'a pas lu plus de caractères que spécifié lors de l'appel à `mark()`) ;
- `read()` pour lire un caractère ;
- `read(char[], int, int)` pour lire un certain nombre de caractères ;
- `readLine()` : pour lire une ligne de texte ;
- `ready()` : pour savoir si l'on peut lire des caractères ;
- `skip(long)` : pour sauter un certain nombre de caractères.

Application : utilisation de l'entée standard

Compte tenu des classes présentées ci-desssus, voici comment lire la réponse à une question en utilisant les entrées-sorties standards (le mode texte).

```
import java.io.*;
import java.lang.*;

public class MaQuestion {
   public static void main(String[] args) {
      System.out.println("Entrer un nombre entier :");
      BufferedReader in = new BufferedReader(
```

```
                        new InputStreamReader(System.in));
        String entree = null;
        try
        {
           entree = in.readLine();
        }
        catch (IOException ex)
        {
           System.exit(1);
        }
        int val = Integer.parseInt(entree);
        int carre = val * val;
        System.out.println("le carre de "
                        + val + " est " + carre);
           }
}
```

4.2 Les fichiers

Tout fichier est d'abord représenté par une instance de la classe `File`. C'est pourquoi nous allons d'abord commencer par étudier cette classe. Ensuite, si l'on veut lire dans un fichier, on utilisera la classe `FileInputStream`, alors qu'on passera par `FileOutputStream` pour écrire dans un fichier.

4.2.1 La classe `File`

Les créateurs

Les trois créateurs de cette classe sont les suivants :

- `File(String)` : on passe le chemin d'accès complet au fichier (relatif ou absolu) ;
- `File(String, String)` : on passe le chemin d'accès au répertoire contenant le fichier ainsi que le nom du fichier (le séparateur est automatiquement inséré par le système) ;
- `File(File, String)` : on passe le répertoire contenant le fichier et le nom de ce dernier. Si la référence au répertoire est à `null`, on travaille avec le répertoire courant.

Les opérations sur les fichiers

- `renameTo(File)` pour renommer un fichier
- `mkdir()`, `mkdirs()` : pour créer un répertoire. Au besoin, le deuxième crée les répertoires intermédiaires.

- `delete()` pour effacer un fichier

Tests sur un fichier

- `exist()` : pour savoir si le fichier existe ;
- `isDirectory()` : pour savoir s'il s'agit d'un répertoire ;
- `isFile()` : pour savoir s'il s'agit d'un fichier standard ;
- `canRead()` : pour savoir si on a les droits en lecture ;
- `canWrite()` : pour savoir si on a les droits en écriture ;

Informations sur le fichier

- `length()` : longueur du fichier ;
- `lastModified()` : date de dernière modification.

Manipulations sur la hiérarchie

- `getAbsolutePath()` : pour obtenir le chemin d'accès absolu ;
- `getCanonicalPath()` : renvoie la forme canonique d'un chemin d'accès ;
- `getName()` : le nom du fichier ;
- `getParent()` : le chemin d'accès au répertoire contenant le fichier ;
- `isAbsolute()` : pour savoir s'il s'agit d'un chemin d'accès absolu.

Cas particulier des répertoires

Si l'objet de type `File()` est un répertoire, on peut lui appliquer deux méthodes :

- `list()` : renvoie la liste des fichiers du répertoire (sous la forme d'un tableau de chaînes de caractères) ;
- `list(FilenameFilter)` : la liste des fichiers du répertoire vérifiant le filtre passé en paramètre.

Remarque : l'interface `FilenameFilter` définit une seule et unique fonction, `accept(File, String)` qui prend en paramètre un répertoire et un nom de fichier, et renvoie un booléen pour indiquer si le fichier vérifie le filtre ou non.

4.2.2 Lecture depuis un fichier

La lecture dans un fichier se fait en créant un objet de type `FileInputStream`. Les créateurs les plus intéressants de cette classe sont les suivants :

- `FileInputStream(File)` ;
- `FileInputStream(String)`.

Les méthodes s'appliquant à un tel objet sont les suivantes :

- `close()` ferme le canal ;
- `read()` : lit un octet ;
- `read(byte[])` : lit un certain nombre d'octets ;
- `read(byte[]), int, int` : lit un certain nombre d'octets ;
- `skip(long)` : saute un certain nombre de caractères.

Bien sûr, si l'on souhaite lire ce fichier comme un fichier de caractères (classique pour un fichier texte), on passera par un objet de type `BufferedReader`, comme pour l'entrée standard. Il est aussi possible de passer par l'intermédiaire de la classe `FileReader` en lieu et place de `FileInputStream`, mais alors on ne dispose pas de la méthode `readLine()`.

4.2.3 Ecrire dans un fichier

Fichier d'octets

Pour ecrire dans un fichier, on passera *a priori* par un objet de type `FileOutputStream`. Les créateurs de la classe en question sont les suivants :

- `FileOutputStream(File)`
- `FileOutputStream(String)`
- `FileOutputStream(String, boolean)` où le booléen permet de préciser si l'on écrit à la fin du fichier ou bien si l'on écrase le contenu précédent. Par défaut, l'ancien contenu du fichier est effacé.

Les méthodes s'appliquant sur un objet de type `FileOutputStream` sont les suivantes :

- `close()`
- `write(byte [])`
- `write(byte [], int, int)`
- `write(int)` : pour écrire un... octet !

Fichier texte

Si l'on souhaite écrire dans un fichier ASCII, on utilisera la classe `FileWriter`. Les créateurs de cette classe sont :

- `FileWriter(File)` ;
- `FileWriter(String)` ;
- `FileWriter(String, boolean)`.

Cette classe héritant de `OutputStreamWriter`, on pourra écrire au moyen des méthodes suivantes :

- `write(int)` pour écrire un caractère ;
- `write(String, int, int)` pour écrire une sous-chaîne ;
- `write(char[], int, int)` pour écrire une partie d'un tableau.

4.3 Lecture et Ecriture de données

Ces opérations s'effectuent respectivement au moyen des classes `DataInputStream` et `DataOutputStream`.

Chapitre 5

Interfaces graphiques

5.1 Introduction

Java est un langage conçu principalement pour fonctionner avec des interfaces graphiques (GUI : *Graphical User Interface*). C'est pourquoi il est presque plus facile de gérer des entrées sorties via ces interfaces qu'en mode texte classique, et c'est aussi la raison qui fait que les classes de base du langage intègrent tout ce qui est nécessaire pour gérer de telles interfaces. Ici, nous présentons les classes du package `awt` des versions 1.1.* de Java. A partir de Java 2 (Java 1.2.*), le rôle de ces classes est mieux géré par `swing`.

La plupart des caractéristiques concernant les interfaces graphiques en Java sont valables que l'on développe une application standard ou bien une applet. Nous présenterons ici ce qui se passe dans le cas d'une application standard. Les différences pour les applets seront abordées dans le chapitre dédié aux applets.

5.2 Structure générale

Une interface graphique est constitué de *composants* disposés dans des *conteneurs* en respectant un certain *style de présentation*. Ces trois concepts se retrouvent presque directement en Java :

- **les composants** sont des objets dont la classe hérite de `java.awt.Component`. C'est notamment le cas des boutons (classe `Button`), des cases à cocher (classe `Checkbox`), des zones de texte (classes `TextArea` et `TextField`, etc.
- **les conteneurs** sont des classes héritant de `java.awt.Container`. C'est le cas de la classe `Frame`, par exemple, qui permet de créer la fenêtre principale d'une application. Mais c'est aussi le cas des classes `Window`, `Panel` (Pour des sous-zones dans une fenêtre), `Dialog` (Pour une simple boîte de dialogue) et `Applet`.

- **les styles de présentation** sont des classes qui implantent l'interface `java.awt.LayoutManager`. Ils permettent de disposer assez simplement les composants au sein d'un conteneur. Un style de présentation peut être attaché à un conteneur au moyen de la méthode `setLayout` définie dans la classe `Container`.

5.2.1 Un premier exemple

Le cahier des charges

On souhaite, dans notre application, afficher des boîtes de dialogue comportant une ligne d'information (variable suivant les circonstances) et deux boutons : *valider* et *annuler*.

La structure de la classe

Comme on veut une boîte de dialogue, Il faut commencer par définir une classe héritant de `java.awt.Dialog`. Cette classe devra définir les variables associant à notre boîte de dialogue ses composants. En l'occurence, ici, on veut une ligne de texte, donc nous aurons une variable d'instance de type `Label` et deux boutons, donc nous aurons deux variables d'instance de type `Button`. Le début de notre application commence donc comme ceci :

```
import java.awt.*;
public class MaBoite extends Dialog {
   private Label ligne;
   private Button valider, annuler;
```

Le constructeur va devoir non seulement appeler le constructeur de la classe mère (`Dialog`), mais aussi définir le style de présentation de notre boîte de dialogue ainsi que la place des composants en accord avec ce style.

Construction de l'objet

Le constructeur de la classe `Dialog` prend trois arguments :

- la fenêtre principale à laquelle est rattachée la boîte de dialogue : c'est donc un objet de type `Frame` ;
- le titre de la boîte de dialogue (optionnel) : de type `String` ;
- la *modalité* de la boîte de dialogue : si ce booléen est à vrai, la fenêtre est *modale*, c'est-à-dire que les autres fenêtres de l'application sont inactives tant qu'elle n'a pas été fermée.

Le code correspondant est alors le suivant :

```
public MaBoite(Frame appli, String message) {
   super(appli, false) //boîte non modale
   setLayout(new BorderLayout());
```

Disposition des composants

Il convient de commencer par choisir comment on va disposer les différents composants au sein de notre boîte de dialogue. Supposons que l'on veuille les disposer ainsi :

- la phrase centrée verticalement et horizontalement ;
- le bouton *Valider* en bas à gauche ;
- le bouton *Annuler* en bas à droite.

On peut revoir cette présentation selon le découpage suivant :

- la phrase centree verticalement et horizontalement ;
- une zone dans le bas. Cette zone contient :
  - un bouton *Valider* à gauche ;
  - un bouton *Annuler* à droite ;

Le découpage de plus haut niveau peut être réalisé au moyen du style de présentation `BorderLayout`. En effet, ce style découpe un conteneur en 5 zones : haut, bas, gauche, droite et centre appelées respectivement `North`, `South`, `West`, `East` et `Center`. Nous attacherons à la zone `Center` le message à afficher.

Le code correspondant est le suivant :

```
ligne = new Label(message);
add("Center", ligne);
```

Pour le bas, nous allons attacher une zone appelée `Panel` (il s'agit d'une sorte de sous-conteneur). Dans cette zone, nous positionnerons un bouton à gauche et un bouton à droite. Là encore, le style de présentation `BorderLayout` peut convenir :

```
Panel zone = new Panel();
zone.setLayout(new BorderLayout());
valider = new Button("Valider");
annuler = new Button("Annuler")
zone.add("West", valider);
zone.add("East", annuler);
add("South", zone);
```

Enfin, il reste à adapter la taille de la boîte de dialogue à son contenu, ce qui se fait grâce à la méthode `pack()` :

```
pack();
}
```

Le programme principal

Toute application utilisant une interface graphique possède une fenêtre principale de type `Frame`. Nous allons donc la créer, puis créer deux boîtes de dialogue rattachées à cette fenêtre. Une fois les boîtes de dialogue crées, il ne restera plus qu'à les afficher :

```
        public static void main(String[] args) {
           Frame f = new Frame("Test de Boîtes");
           f.setSize(100,50);
           f.show()
           MaBoite b1 = new MaBoite(f, "La première boîte");
           MaBoite b2 = new MaBoite(f, "La deuxièmee boîte");
           b1.show();
           b2.show();
        }
     }
```

Le programme Java ainsi créé affiche une fenêtre principale et deux boîtes de dialogue qui y sont rattachées. Nous verrons plus tard comment gérer les événements que peuvent générer ces boîtes de dialogue, notamment le clic sur un de leurs boutons.

5.3 Les composants

On distingue les composants suivants :

- les boutons (`Button`);
- les zones de dessin (`Canvas`) ;
- les cases à cocher (`Checkbox`) (qui peuvent devenir des boutons radio en utilisant la classe `CheckboxGroup`) ;
- les menus de choix (`Choice`) ;
- les étiquettes (`Label`) ;
- les listes d'articles (`List`) ;
- les ascenseurs (`Scrollbar`) ;
- les zones de saisie de texte (`TextArea` et `TextField`).

Par la suite, nous allons détailler le contenu des composants, mais sans toutesfois aborder l'aspect *événement*.

5.3.1 La classe `Component`

Il s'agit d'une classe abstraite dont héritent toutes les classes décrivant des composants. Elle comporte notamment les méthodes suivantes :

- `add(PopupMenu)` et `remove(MenuComponent)` : pour ajouter ou supprimer un menu contextuel à un composant ;
- `contains(int, int)` ou `contains(Point)` pour savoir si un point appartient au composant ;
- `getBackground` et `setBackground` pour connaître ou fixer la couleur de fond du composant ;
- `getForeground` et `setForeground` pour connaître ou fixer la couleur de premier plan du composant ;
- `getBounds` pour connaître les coordonnées du composant (renvoie en résultat un `Rectangle`) ;
- `getFont` et `setFont` pour connaître ou fixer la police de caractères ;
- `getCursor` et `setCursor` pour connaître ou fixer le curseur défini pour le composant ;
- `paint` pour dessiner le composant ;
- `setVisible(boolean)` pour afficher ou masquer le composant ;
- `setSize(int, int)` pour définir la taille du composant.

Pour connaître les autres méthodes, voir :

`http://java.sun.com/products/jdk/1.1/docs/api/java.awt.Component.html`

5.3.2 Les boutons

Voici quelques une des méthodes de la classe `Button` :

- `Button()` et `Button(String)` pour créer un bouton avec un éventuel nom ;
- `setLabel(String)` et `getLabel` pour fixer ou connaître le nom d'un bouton.

5.3.3 Les zones de dessin

La classe `Canvas` contient principalement deux méthodes : le constructeur `Canvas` et la méthode `paint(Graphics)`, en général à redéfinir.

5.3.4 Les cases à cocher

Les méthodes de la classe `Checkbox` ayant trait aux cases à cocher sont les suivantes :

- `Checkbox()`, `Checkbox(String)` et `Checkbox(String, boolean)` pour créer une case à cocher avec un éventuel nom associé et un éventuel état (l'état par défaut est *off*) ;
- `setLabel(String)` et `getLabel()` pour associer un nom à la case à cocher ;
- `setState(boolean)` et `getState` pour modifier ou connaître l'état d'une case à cocher.

5.3.5 Les boutons radio

Les boutons radio sont des cases à cocher un peu particulières : au sein d'un groupe de boutons radio, seul un bouton peut être dans l'état *on* à un instant donné. En Java, ils sont créés comme des cases à cocher associées à un groupe de type `CheckboxGroup`.

Les méthodes essentielles de la classe `CheckboxGroup` sont les suivantes :

- `CheckboxGroup()` : le constructeur ;
- `setSelectedCheckbox(Checkbox)` : pour préciser le bouton à *on* ;
- `getSelectedCheckbox()` : pour connaître le bouton à *on*.

Les méthodes de la classe `Checkbox` en rapport avec les boutons radio sont les suivantes :

- Checkbox(String, boolean, CheckboxGroup) et Checkbox(String, CheckboxGroup, boolean) : constructeurs associant un bouton radio à un groupe ;
- setCheckboxGroup(CheckboxGroup) pour associer un bouton radio à un groupe ;
- getCheckboxGroup() pour connaître le groupe auquel un bouton radio est associé.

5.3.6 Les menus de choix

Ces menus permettent de choisir entre plusieurs options prédéfinies. Les méthodes principales de la classe `Choice` sont les suivantes :

- `Choice()` : le constructeur ;
- `add(String)` et `addItem(String)` : pour ajouter une option au menu ;
- `insert(String, int)` pour ajouter une option au menu à la position donnée ;

- `getItem(int)` : pour connaître le nom de l'option située à la position passée en paramètre ;
- `getSelectedIndex()` et `getSelectedItem()` : pour connaître la position ou le nom de l'option sélectionnée ;
- `remove(int)` et `remove(String)` : pour retirer une option du menu ;
- `removeAll()` : Pour supprimer toutes les options d'un menu ;
- `select(int)` et `select(String)` : pour préciser quelle option est sélectionnée.

5.3.7 Les étiquettes

Les étiquettes permettent d'afficher une ligne de texte statique. Les principales méthodes de la classe `Label` sont les suivantes :

- `Label()`, `Label(Sring)`, `Label(String, int)` : permet de construire une étiquette avec un texte éventuel, en précisant si nécessaire l'alignement (LEFT, RIGHT, CENTER) ;
- `setText(String)` et `getText()` : pour fixer ou connaître le contenu d'une étiquette.

5.3.8 Les listes d'articles

Il s'agit de listes dans lesquelles on peut choisir un ou plusieurs articles. Elles sont implantées au moyen de la classe `List` dont voici les méthodes essentielles :

- `Liste()`, `List(int)` et `List(int, boolean)` : permet de créer une telle liste, avec éventuellement un nombre d'articles à afficher, et en précisant si nécessaire si l'on peut sélectionner plus d'un article de la liste. La sélection multiple est interdite par défaut. Seul un booléen spécifié à vrai permet de l'autoriser.
- `add(String)`, `additem(String)`, `add(String, int)` et `additem (String, int)` pour ajouter un article à une liste, en précisant éventuellement sa position ;
- `delitem(int)`, `remove(int)` et `remove(String)` : pour supprimer l'article situé à la position donnée en paramètre ou bien la première occurence d'un article portant un nom donné ;
- `removeAll()` : pour supprimer tous les articles d'une liste ;
- `deselect(int)` : pour déselectionner un article ;
- `getItem(int)` : pour connaître l'article situé à la position donnée ;

- `getItemCount()` : pour connaître le nombre d'articles dans la liste ;
- `getItems()` : pour connaître tous les articles de la liste ;
- `getRows()` : pour connaître le nombre de lignes visibles ;
- `getSelectedIndex()` et `getSelectedItem()` : pour connaître la position ou le nom de l'article sélectionné ;
- `getSelectedIndexes()` et `getSelectedItems()` : renvoie sous la forme d'un tableau d'entiers ou de chaînes de caractères les positions ou les noms des différents articles séletionnés ;
- `isIndexSelected(int)` : pour savoir si un article est sélectionné ;
- `isMultipleMode()` : pour savoir si on a droit aux sélections multiples ;
- `replaceItem(String, int)` : pour changer l'article à la position donné ;
- `select(int)` : pour sélectionner un article ;
- `setMultipleMode(boolean)` : pour autoriser ou interdire la sélection multiple ;

5.3.9 Les ascenseurs

La classe `Scrollbar` permet d'implanter des ascenseurs, aussi bien horizontaux que verticaux. On les gère au moyen des méthodes suivantes :

- `Scrollbar(int, int, int, int, int)` : crée un ascenseur. Le premier paramètre précise l'orientation : `VERTICAL` ou `HORIZONTAL`. Le deuxième paramètre est la position initiale de l'ascenseur, le troisième paramètre est la taille de la zone représentée par l'ascenseur, et les quatrième et cinquième paramètres sont les minimum et maximum que peut prendre l'ascenseur.
- `getOrientation()`, `getValue` `getBlockIncrement()`, `getMinimumValue()` et `getMaximumValue()` pour connaître certaines informations ;
- `setOrientation(int)`, `setValue` `setBlockIncrement()`, `setMinimumValue()` et `setMaximumValue()` pour modifier certaines informations ;
- `getUnitIncrement()` et `setUnitIncrement(int)` : pour l'incrément unitaire ;
- `getVisibleAmount()` et `setVisibleAmount(int)` : pour fixer et connaître la taille représentée par l'ascenseur : peut *a priori* être différente du `BlockIncrement`.

5.3.10 Les zones de texte

Il existe deux types de zones de texte : les `TextField` et les `TextArea`. Le premier type correspond à des zones d'une ligne, et le deuxième a des zones de plusieurs lignes. Ces deux classes possèdent toutefois des méthodes en commun, regroupées dans la classe abstraite `TextComponent` dont elles héritent toutes deux.

Les méthodes communes de la classe `TextComponent`

- `getCaretPosition()` et `setCaretPosition(int)` : retourne la position du curseur ou la fixe ;
- `getSelectedText()` : le texte sélectionné ;
- `getSelectionStart()` et `getSelectionEnd()` : début et fin de la zone sélectionnée ;
- `setSelectionStart()` et `setSelectionEnd()` : fixe le début et la fin de la zone sélectionnée ;
- `getText()` : le texte contenu dans le champ ;
- `setEditable(boolean)` et `isEditable()` : pour rendre le texte editable ou savoir s'il l'est ;
- `select(int, int)` : pour sélectionner le texte entre les positions indiquées ;
- `selectAll()` : sélectionne tout le texte ;
- `setText(String)` : pour modifier le texte.

Méthodes propres à la classe `TextField`

- `TextField`, `TextField(int)`, `TextField(String)`, `TextField(String, int)` : constructeur prenant éventuellement en paramètres un texte initial et une largeur exprimée en nombre de colonnes ;
- `getColumns()` et `setColumns(int)` : pour connaître ou fixer le nombre de colonnes ;
- `setEchoChar(char)` et `getEchoChar()` : pour fixer le caractère affiché chaque fois qu'un caractère est tapé (pour les champs *mot de passe*, par exemple) ou pour connaître ce caractère. `echoCharIsSet()` permet de savoir si un tel caractère est fixé.
- `getMinimumSize()`, `getMinimumSize(int)`, `getPreferredSize()` et `getPreferredSize(int)` : pour connaître les tailles minimum et idéales du champ, en précisant éventuellement un nombre de colonnes.

Méthodes propres à la classe `TextArea`

- `TextArea()`, `TextArea(int, int)`, `TextArea(String)`, `TextArea (String, int, int)` et `TextArea(String, int, int, int)` : Constructeur dont les paramètres optionnels sont : le nombre de lignes et de colonnes, le texte initial, et la présence ou non de barres de défilement. Ce dernier paramètre peut prendre l'une des valeurs suivantes :
  - `SCROLLBARS_BOTH`,
  - `SCROLLBARS_HORIZONTAL_ONLY`,
  - `SCROLLBARS_VERTICAL_ONLY`,
  - `SCROLLBARS_NONE`.
- `apend(String)` : pour ajouter du texte ;
- `insert(String, int)` : pour insérer du texte à une position donnée ;
- `replaceRange(String, int, int)` : pour remplacer une partie du texte ;
- `getMinimumSize()`, `getMinimumSize(int,int)`, `getPreferredSize()` et `getPreferredSize(int,int)` : pour connaître les tailles minimum et idéales du champ, en précisant éventuellement des nombres de lignes et de colonnes.
- `getColumns()` et `setColumns(int)` : pour connaître ou fixer le nombre de colonnes ;
- `getRows()` et `setRows(int)` : pour connaître ou fixer le nombre de colonnes.
- `getScrollbarVisibility()` : pour connaître les ascenseurs utilisés ;

5.4 Les conteneurs

Les différents conteneurs sont les suivants :

- `Panel` ;
- `Applet`, qui hérite de Panel ;
- `Window` ;
- `Frame`, qui hérite de `Window` ;
- `Dialog`, qui hérite de `Window` ;
- `FileDialog`, qui hérite de `Dialog` ;
- `ScrollPane`.

Ils héritent tous de la classe abstraite `Container` qui hérite elle-même de `Component`.

5.4.1 La classe `Container`

Un conteneur gère l'ensemble de ses composants sous la forme d'une liste dont l'ordre détermine dans certains cas la visibilité. Cette classe définit de nombreuses méthodes dont voici les principales :

- `add(Component)`, `add(Component, int)`, `add(Component, Object)` et `add (Component, Object, int)` : ajoute un composant, en précisant éventuellement une position dans la liste et des contraintes.
- `getComponent(int)` : pour connaître un composant à une position donnée ;
- `getComponentAt(int, int)` et getComponentAt(Point) : pour savoir au sein de quel composant est un point donné ;
- `getComponentCounts()` : renvoie le nombre de composants ;
- `getComponents()` : renvoie la liste des composants ;
- `getMinimumSize()`; `getMaximumSize()` et `getPreferredSize()` : pour connaître les tailles en question du conteneur ;
- `remove(int)`, `remove(Component)` : pour supprimer un composant dont on passe soit la référence soit la position ;
- `removeAll()` : supprime tous les composants ;
- `setLayout(LayoutManager)` et `getLayout()` : pour fixer ou connaître le gestionnaire de présentation ;
- `doLayout()` : poru forcer la disposition des composants ;
- `update` : met à jour le conteneur. Ne pas utiliser en général, mais faire appel à `validate()`;
- `validate` : met à jour le conteneur ;
- paint(Graphics) : dessine le conteneur ;
- paintComponents(Graphics) : dessine chacun ces composants du conteneur ;
- `isAncestorOf(Component)` : permet de savoir si un composant figure bien (directement ou indirectement) dans le conteneur.

5.4.2 La classe `Window`

Cette classe contient notamment les méthodes suivantes :

- `Window(Frame)` : constructeur créant une fenêtre invisible et prenant en paramètre la fenêtre principale de l'application ;
- `pack()` : donne à chaque composant sa taille idéale ;
- `show()` : affiche la fenêtre et la met au premier plan ;
- `isShowing()` : pour savoir si la fenêtre est affichée ;
- `toFront()` et `toBack()` : met une fenêtre au premier plan ou à l'arrière plan ;
- `dispose()` : permet de libérer les ressources associées à une fenêtre dont on n'a plus besoin ;
- `getToolkit()` : pour renvoyer l'objet de type `Toolkit` rattaché à la fenêtre ;
- `getFocusOwner()` : renvoie le composant actuellement actif.

5.4.3 La classe `Frame`

Cette classe permet de créer une fenêtre *principale* avec barre de menus et cadre (il peut y avoir plusieurs *frames* au sein d'une application, donc le terme *principale* n'est pas forcément bien adapté). Comme elle hérite de la classe `Window`, les méthodes définies dans cette dernière sont aussi applicables à la classe `Frame`.

Les méthodes essentielles de cette classe sont les suivantes :

- `Frame()` et `Frame(String)` : crée une fenêtre avec éventuellement un titre. Au départ, la fenêtre est invisible.
- `setIconImage(Image)` et `getIconImage()` : pour connaître ou fixer l'icone représentant la fenêtre ;
- `setMenuBar(MenuBar)` et `getMenuBar()` : pour fixer ou récupérer la barre de menus attachée à la fenêtre ;
- `remove(MenuComponent)` : pour supprimer la barre de menus ;
- `setResizable(boolean)` et `getResizable()` : pour préciser ou savoir si la taille d'une fenêtre peut être modifiée (à *vrai* par défaut) ;
- `getTitle()` et `setTitle(string)` : pour connaître ou modifier le titre de la fenêtre.

5.4.4 Les boîtes de dialogue

Elles sont gérées par la classe `Dialog`, qui hérite de `Window`. Une boîte de dialogue ne possède pas de barre de menus et est toujours rattachée à une fenêtre principale (`Frame`). On distingue les boîtes de dialogue *modales* (elles bloquent toute action dans la *Frame* dont elle dépend) des autres.

- `Dialog(Frame)`, `Dialog(Frame, boolean)`, `Dialog (Frame, String)`, `Dialog(Frame, String, Boolean)` : constructeur. On peut éventuellement préciser le titre de la boîte de dialogue et sa modalité (une boîte de dialogue est par défaut non modale).
- `setModal(boolean)` et `isModal()` : pour modifier ou connaître la modalité d'une boîte de dialogue ;
- `setResizable(boolean)` et `isResizable()` pour préciser ou savoir si la taille d'une boîte de dialogue peut être modifiée ;
- `setTitle(String)` et `getTitle()` pour modifier ou connaître le titre d'une boîte de dialogue ;
- `show()` : pour afficher une boîte de dialogue.

5.4.5 Boite de dialogue d'ouverture/sauvegarde de fichiers

Il s'agit d'une boîte de dialogue modale standard pour ouvrir ou sauvegarder un fichier. Les méthodes propres à la classe `FileDialog` sont les suivantes :

- `FileDialog(Frame)`, `FileDialog (Frame, String)`, `FileDialog (Frame, String, int)` : Constructeur prenant en paramètres éventuels le titre de la boîte de dialogue et son type : `LOAD` ou `SAVE` (par défaut, il s'agit de `LOAD`).
- `setDirectory(String)` et `getDirectory()` : pour fixer ou modifier le répertoire courant ;
- `setFile(String)` et `getFile()` : fixe ou renvoie le fichier sélectionné ;
- `setFilenameFilter(String)` et `getFilenameFilter()` : modifie ou renvoie le filtre à appliquer aux noms de fichiers.
- setMode(int) : pour modifier le type : `LOAD` ou `SAVE`.

5.4.6 La classe `Panel`

Cette classe permet de créer un conteneur invisible inclus comme composant d'un autre conteneur, facilitant ainsi certaines dispositions de composants qui, sans cela, seraient relativement compliquées. Le constructeur de cette classe, `Panel()`, ne prend aucun paramètre. Il existe une autre version, `Panel(LayoutManager)`, qui prend en paramètre le gestionnaire de présentation souhaité.

5.4.7 La classe `ScrollPane`

5.4.8 La classe `Applet`

Voir le chapitre consacré aux applets.

5.5 Les styles de présentation

5.5.1 Introduction

Les gestionnaires de présentation implantent tous l'interface `LayoutManager`, éventuellement indirectement via l'interface `LayoutManager2`, qui hérite de `LayoutManager`. Un gestionnaire de présentation s'utilise en étant rattaché à un conteneur. C'est lors de l'appel à la méthode `add` du conteneur en question que le gestionnaire va être appelé, et qu'il placera le nouveau composant de manière à respecter la contrainte passée en paramètre. Les gestionnaires de présentation sont au nombre de 5 :

- `BorderLayout` ;
- `FlowLayout` ;
- `GridLayout` ;
- `GridBagLayout` ;
- `CardLayout`.

Les méthodes présentes dans l'interface `LayoutManager` sont les suivantes :

- addLayoutComponent(String, Component) ;
- layoutContainer(Container) ;
- minimumLayoutSize(Container) ;
- preferredLayoutSize(Container);
- removeLayoutComponent(Component);

Cependant, l'appel à ces méthodes étant effectué dans les classes de conteneurs, nous ne nous attarderons pas dessus. Etudions plutôt les propriétés des différents gestionnaires de présentation.

En effet, pour utiliser ces gestionnaires, on commence à les attacher à un conteneur au moyen de la méthode `setLayout(LayoutManager)` du conteneur. C'est l'appel à la méthode `add(String, Component)` du conteneur qui appellera le gestionnaire de présentation, où la chaîne de caractères représente la contrainte que l'on passe au gestionnaire.

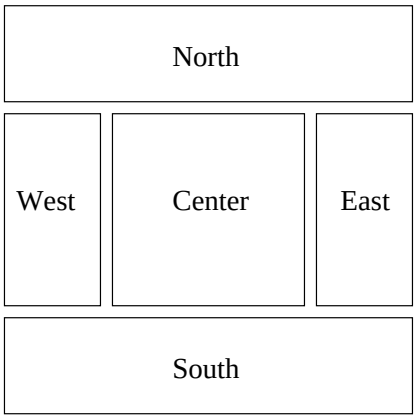


Figure 5.1: Le gestionnaire BorderLayout

5.5.2 `BorderLayout`

Le gestionnaire de présentation `BorderLayout` divise le conteneur auquel il est associé en 5 zones, comme le montre la figure 5.1.

Pour préciser où mettre un composant dans un conteneur auquel ce gestionnaire est rattaché, il suffit de passer comme contrainte le nom de la zone (voir figure) dans laquelle on compte mettre le composant.

Il existe deux versions du constructeur : `BorderLayout()` et `BorderLayout(int, int)` : les deux paramètres optionnels sont l'espacement entre deux zones, horizontalement et verticalement.

5.5.3 `FlowLayout`

Ce gestionnaire remplit le conteneur ligne à ligne : tant qu'il y a de la place sur une ligne, il y ajoute le nouveau composant. Lorsque ce n'est plus possible, il passe à une nouvelle ligne.

Le constructeur existe sous trois formes : `FlowLayout()`, `FlowLayout(int)`, `FlowLayout(int, int, int)`. Le premier paramètre optionnel peut valoir :

- `CENTER`,
- `LEFT`,
- `RIGHT`.

selon l'alignement que l'on souhaite pour les différents composants d'une ligne. Les deux derniers paramèetres désignent quant à eux les espacements horizontaux et verticaux entre composants.

5.5.4 GridLayout

Ce gestionnaire arrange les composants selon une grille. On trouve notamment les deux formes suivantes du constructeur : `GridLayout(int, int)` et `GridLayout(int, int, int, int)`. Les deux premiers paramètres désignent respectivement les nombres de lignes et de colonnes, les deux paramètres suivants, optionnels, correspondent aux intervalles, en horizontal et en vertical, entre composants adjacents.

5.5.5 GridBagLayout

Ce gestionnaire est le plus puissant des différents gestionnaires de présentation. Il permet de placer les différents composants sur une grille, mais chaque composant peut utiliser un nombre variable de lignes et de colonnes. Les caractéristiques de positionnement d'un composant sont spécifiées au moyen de la classe `GridBagConstraints`.

Un exemple

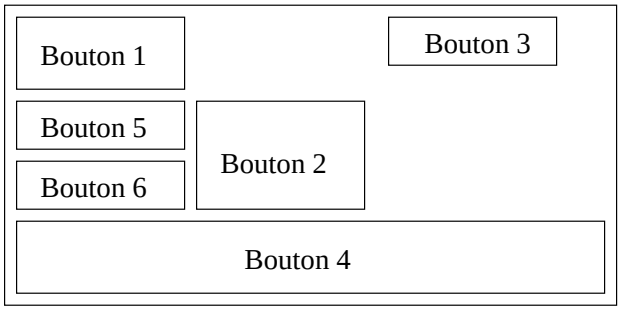
Voici un exemple de code utilisant le gestionnaire `GridBagLayout` :

```
import java.awt.*;
public class ExempleGridBag extends Frame {
   private Button b1, b2, b3, b4, b5, b6;
   public ExempleGridBag() {
      GridBagLayout gridbag = new GridBagLayout();
      setLayout(gridbag);
      GridBagConstraints c = new GridBagConstraints();
      //position du premier bouton
      c.gridx = 0; c.gridy = 0;
      //taille du premier bouton
      //c.gridwidth = 1;c.gridheight = 1;
      //Comportement lorsque la taille
      //de la frame est modifiee
      c.fill = GridBagConstraints.VERTICAL;
      c.weighty = 1.0;
      c.weightx = 0.5;
      b1 = new Button("Bouton 1");
      gridbag.setConstraints(b1, c);
      add(b1);
      //position du deuxieme bouton
      c.gridx = 1;
      c.gridy = 1;
      //taille du deuxieme bouton
      c.gridwidth = 2;
      c.gridheight = 2;
      //Comportement si la taille de la frame est modifiee
```

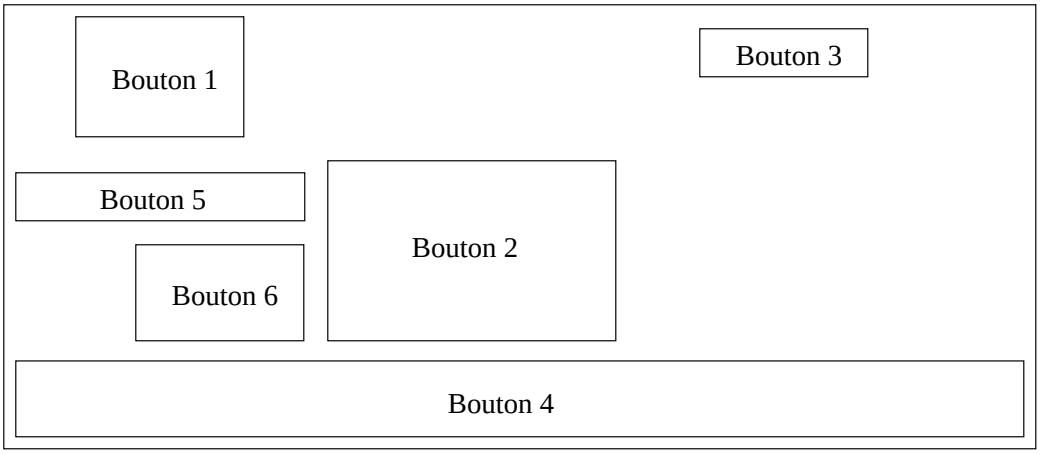
```
c.fill = GridBagConstraints.BOTH;
c.weightx = 1.0;
//c.heighty = 1.0;
b2 = new Button("Bouton 2");
gridbag.setConstraints(b2, c);
add(b2);
//position du troisieme bouton
c.gridx = 3;
c.gridy = 0;
//taille du troisieme bouton
c.gridwidth = 1;
c.gridheight = 1;
c.insets = new Insets(0,5,10,15);
//Comportement si la taille de la frame est modifiee
c.fill = GridBagConstraints.NONE;
//c.weightx = 1.0;
//c.weighty = 1.0;
b3 = new Button("Bouton 3");
gridbag.setConstraints(b3, c);
add(b3);
//quatrieme bouton
c.insets = new Insets(0,0,0,0);
c.gridx = 0;
c.gridy = 3;
c.gridwidth = 4;
//c.gridheight = 1;
c.fill = GridBagConstraints.HORIZONTAL;
c.ipady = 5;
c.weighty = 0.0;
b4 = new Button("Bouton 4");
gridbag.setConstraints(b4, c);
add(b4);
//cinquieme bouton
c.ipady = 0;
//c.gridx = 0;
c.gridy = 1;
c.gridwidth = 1;
//c.gridheight = 1;
c.weighty = 0.5;
b5 = new Button("Bouton 5");
gridbag.setConstraints(b5, c);
add(b5);
//sixieme bouton
//c.gridx = 0;
c.gridy = 2;
//c.gridwidth = 1;
```

```
        //c.gridheight = 1;
        c.fill = GridBagConstraints.VERTICAL;
        c.anchor = GridBagConstraints.EAST;
        c.weighty = 1.0;
        b6 = new Button("Bouton 6");
        gridbag.setConstraints(b6, c);
        add(b6);
    }
    public static void main(String[] args) {
        ExempleGridBag egb = new ExempleGridBag();
        egb.pack();
        egb.show();
    }
}
```

Le résultat produit par défaut a l'aspect suivant :



Si l'on agrandit à la main la taille de la fenêtre, elle prend alors l'aspect suivant :



Les contraintes possibles

Résumons les contraintes qu'il est possible de spécifier :

- `gridx`, `gridy` : position du composant dans la grille ;
- `gridwidth`, `gridheight` : largeur et hauteur du composant (en nombre de cases de la grille) ;
- `ipadx`, `ipady` : espace à rajouter à la taille par défaut du composant ;
- `insets` : espace à rajouter autour du composant ;
- `anchor` : si l'espace est plus grand que la taille du composant, précise où afficher le composant dans cet espace. Les valeurs possibles sont : `CENTER`, `NORTH`, `SOUTH`, `WEST`, `EAST`, `NORTHEAST`, `SOUTHEAST`, `SOUTHWEST`, `NORTHWEST`.
- `fill` : occupation de l'espace supplémentaire s'il y en a par le composant. Peut valoir `NONE`, `VERTICAL`, `HORIZONTAL`, `BOTH`.
- `weightx`, `wheighty` : poids pour indiquer comment répartir entre colonnes et entre lignes l'espace supplémentaire lorsque la taille du conteneur est modifiée.

5.5.6 `CardLayout`

Ce gestionnaire permet de gérer une liste de composants comme une pile de cartes : un seul composant est visible à la fois. Il possède notamment les méthodes suivantes :

- `first(Container)` : pour afficher le premier composant ;
- `last(Container)` : pour afficher le dernier composant ;
- `next(Container)` : pour afficher le composant suivant ;
- `previous(Container)` : pour afficher le composant précédent ;
- `addLayoutComponent(Component, Objet)` : ou le paramètre de type `Object` doit être une chaîne de caractères : permet d'associer un nom à un composant pour faciliter les accès directs ;
- `removeLayoutComponent(Component)` : pour supprimer un composant ;
- `show(Container, String)` : pour afficher un composant avec un nom donné.

5.6 Les menus

En Java, un menu (`Menu`) est constitué d'un ensemble d'*articles de menus* (`MenuItem`). De plus, un menu est en général intégré à une barre de menus (`MenuBar`).

Examinons ces différents éléments.

5.6.1 La barre de menus

Une barre de menus est en général toujours rattachée à une fenêtre de type `Frame`. Ce rattachement est effectué au moyen de la méthode `setMenuBar(MenuBar)` de la classe `Frame`.

Les méthodes les plus importantes de la classe `MenuBar` sont les suivantes :

- `MenuBar()` : pour créer une barre de menus ;
- `add(Menu)` : pour ajouter un menu ;
- `remove(int)` et `remove(MenuComponent)` : pour supprimer un menu ;
- `getMenu(int)` : pour récupérer une référence à un menu ;
- `getMenuCount()` : pour connaître le nombre de menus ;
- `getShortcutMenuItem(MenuShortcut)` : pour connaître l'article de menu auquel le raccourci spécifié est associé.
- `deleteShortcut(MenuShortcut)` : pour supprimer un raccourci clavier ;
- `shortcuts()` : pour avoir la liste des équivalents claviers sous la forme d'une *énumération* (`Enumeration`).

Le menu d'aide est un menu comme les autres, mis à part le fait que Java le gère différemment selon la plateforme d'exécution, afin de respecter le style d'interface en vigueur. Aussi, il est inséré différemment dans la barre de menus :

- `setHelpMenu(Menu)` : pour fixer le menu d'aide ;
- `getHelpMenu()` : pour connaître le menu d'aide ;

5.6.2 Les menus

La classe `Menu` hérite de la classe `MenuItem`. Un menu est une liste d'articles de menus. Voici une liste de méthodes de la classe `Menu` :

- `Menu(String)`, `Menu(String, boolean)` : crée un menu avec le nom spécifié. Si le booléen est spécifié et est à vrai, il s'agit d'un menu *tear-off*, c'est-à-dire qu'on peut le *sortir* de la barre de menu, de telle sorte qu'il reste affiché.

- `add(String)`, `add(MenuItem)` : pour ajouter un article au menu.
- `addSeparator()` : pour ajouter une ligne de séparation.
- `getItem(int)` : pour récupérer l'article à la position spécifiée ;
- `insert(MenuItem, int)`, `insert(String, int)` et `insertSeparator(int)` : pour faire une insertion dans un menu ;
- `getItemCount()` : pour connaître le nombre d'articles dans le menu ;
- `remove(int)`, `remove(MenuComponent)` : pour supprimer un article particulier ;
- `removeAll()` : pour supprimer tous les articles d'un menu.

5.6.3 Les articles de menus

Dans le cas où il s'agit d'un article simple (juste un nom), on peut insérer l'article en question directement au moyen de la méthode `add(String)` de la classe `Menu`. Mais il existe d'autres types d'articles de menus. Ainsi, un sous-menu est un article de menu particulier. Mais comme la classe `Menu` hérite de la classe `MenuItem`, il est possible d'ajouter un sous-menu au moyen de la méthode `add(MenuItem)` de la classe `Menu`.

La classe `MenuItem`, dont hérite forcément tous les articles de menu, gère les articles de menu simples, possède notamment les méthodes suivantes :

- `setShortcut(MenuShortcut)`, `getShortcut()` et `deleteShortcut()` : pour les raccourcis-claviers ;
- `setEnabled(boolean)` et `isEnabled()` : pour valider ou invalider l'article, ou bien savoir s'il est validé ;
- `getLabel()` et `setLabel(String)` : pour changer le nom de l'article ;
- `setActionCommand(String)` : si le nom à associer à l'événement généré par cet article est différent du nom de l'article (voir le chapitre concernant les événements) ;
- `getActionCommand()` : comme son nom l'indique.

Les articles à cocher

Il s'agit d'un article de menu particulier, dont la classe est `CheckboxMenuItem`, constitué d'un nom précédé d'une case à cocher, pouvant être cochée ou non. Les méthodes essentielles de la classe en question sont :

- `CheckboxMenuItem(String)` et `CheckboxMenuItem(String, boolean)` : les constructeurs ;
- `getState()`, `setState(boolean)` : pour connaître ou modifier l'état de la case à cocher.

5.6.4 Les raccourcis-claviers

Ils sont implantés au moyen de la classe `MenuShortcut`. La méthode essentielle de cette classe est son constructeur `MenuShortcut(int, boolean)`, qui prend en paramètre le code de la touche ainsi qu'un booléen spécifiant si la touche `SHIFT` doit être enfoncée ou pas.

5.6.5 Un exemple relativement complet

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

   public class MaFrameAMenus extends Frame
                           implements ActionListener {
      MenuItem quitter ;
      Label contenu;
      CheckboxMenuItem acocher;

      public MaFrameAMenus() {
         MenuBar mb = new MenuBar();
         Menu menufichier = new Menu("Fichier");
         Menu menuautre = new Menu("Autre", true);
         Menu sousmenu = new Menu("Suite");
         acocher = new CheckboxMenuItem("Sortie possible");

         quitter = new MenuItem("Quitter");
         quitter.setActionCommand("Quitter");
         // inutile en temps normal ;
         // mais necessaire quand on utilise les raccourcis


         quitter.addActionListener(this);
         MenuShortcut raccourci =
                           new MenuShortcut(KeyEvent.VK_Q);
         quitter.setShortcut(raccourci);

         sousmenu.add("Rien");
         sousmenu.add(acocher);
         sousmenu.add(quitter);

         menufichier.add("ouvrir");
         menufichier.add("sauver");
         menufichier.add("sauver sous...");
         menufichier.addSeparator();
         menufichier.add(sousmenu);
```

```
        mb.add(menufichier);

        menuautre.add("première option");
        menuautre.add("deuxième option");

        MenuItem deux = menuautre.getItem(1);
        deux.setEnabled(false);

        mb.add(menuautre);

        setMenuBar(mb);

        contenu = new Label("Un texte pour faire joli");

        add("Center", contenu);

    }

    public void actionPerformed(ActionEvent ae) {
        if (ae.getActionCommand().equals("Quitter") &
                            [acocher.getState() == true)
        {System.exit(0);}
    }

    public static void main(String[] args) {
        MaFrameAMenus mf = new MaFrameAMenus();
        mf.setTitle("Titre");
        mf.pack();
        mf.setVisible(true);
    }
}
```

5.6.6 Les menus contextuels

La classe `PopupMenu`, qui permet de créer des menus contextuels, est une sous classe de la classe `Menu`. Par conséquent, mis à part le constructeur (`PopupMenu(String)`), elle s'utilise comme la classe `Menu`. Pour associer un menu contextuel à un composant, il suffit d'utiliser la méthode `add(PopupMenu)` de la classe `Component`.

5.7 Les couleurs

On a vu que l'on pouvait modifier la couleur des composants grâce aux méthodes `setBackground(Color)` et `setForeground(Color)`. Mais comment fonctionne la classe `Color` ? En voici un descriptif.

5.7.1 Les constantes

Certaines couleurs courantes sont définies sous la forme de constantes. Ce sont les suivantes :

- `black` et `white` : noir et blanc ;
- `darkGray`, `gray` et `lightGray` : différents niveaux de gris ;
- `blue` : bleu
- `cyan` : cyan
- `green` : vert
- `magenta` : magenta
- `orange` : orange
- `pink` : rose
- `red` : rouge
- `yellow` : jaune

5.7.2 Les créateurs

On dispose de 3 créateurs :

- Color(int, int, int) : on passe les niveaux des 3 couleurs (Rouge, Vert, Bleu) sous la forme de 3 entiers compris entre 0 et 255.
- Color(float, float, float) : on passe les niveaux des 3 couleurs sous la forme d'un pourcentage du niveau de chacune des couleurs. Il s'agit dont de nombres entre 0 et 1.
- Color(int) : le niveau du rouge est exprimé par les bits 16 à 23, celui du vert par les bits 8 à 15 et celui du bleu par les bits 0 à 7.

5.7.3 D'autres méthodes

Voici quelques unes des autres méthodes de la classe `Color` :

- `brighter()` et `darker()` : pour éclaircir ou assombrir une couleur ;
- `equals(Object)` : pour comparer deux couleurs ;
- `getBlue()`, `getGreen()` et `getRed` : pour connaître le niveau de chacune des couleurs ;

5.8 Les polices de caractères

Les polices de caractères sont principalement gérées par la classe `Font` que l'on présente brièvement ici.

- Font(String, int, int) : constructeur. Le premier paramètre est le nom de la police, le deuxième est son style, et le troisième est sa taille. Les valeurs possibles pour le style sont les suivantes :

  - `PLAIN` : style normal ;
  - `BOLD` : gras ;
  - `ITALIC` : italique ;
  - `BOLD+ITALIC` : gras italique.

  La liste des polices disponibles peut être obtenue au moyen de la méthode `getFontList` de la classe `Toolkit` dont voici un exemple d'utilisation :

```
   import java.awt.*;
public class Polices extends Frame {
public static void main(String[] args) {
Polices mf = new Polices();
Toolkit tk;
tk = mf.getToolkit();
String[] fl;
fl = tk.getFontList();
int i;
for (i = 0 ; i < fl.length;i++)
{
System.out.println(fl[i]);
}
System.exit(0);
}
}
```

- `getName()` : renvoie le nom logique de la police ;
- `getStyle()` : renvoie le style de la police ;
- `getSyze()` : renvoie la taille de la police ;
- `isPlain()`, `isBold()`, `isItalic()` : pour savoir si le style de la police a certaines caractéristiques ;
- `equals(Object)` : pour comparer deux polices.

Chapitre 6

Les événements

6.1 Introduction

Bien que pouvant exister hors du contexte des interfaces graphiques, ils sont essentiels à la gestion des interfaces graphiques, et c'est là leur principale utilité.

Un événement au sens informatique est très proche de la notion d'événement au sens classique du terme : il indique que quelque chose vient de se passer. Afin de préciser certaines choses sur ce qui s'est passé, tout événement est caractérisé par un objet (dont la classe hérite de `java.util.EventObject`). Un événement a vocation à être pris en compte aussi tôt que possible, rompant ainsi avec la traditionnelle vision d'un programme s'exécutant séquentiellement.

Chaque composant d'une interface graphique peut générer différents types d'événements. Par exemple, le fait de cliquer sur un bouton génére un événement de typ `ActionEvent`.

Pour chaque composant dont on désire traiter un type d'événement particulier, il faut lui attacher un *mouchard* (*listener*) d'événements de ce type. Ce mouchard est une classe qui implante l'interface correspondante (`ActionListener` pour un événement de type `ActionEvent`, par exemple) et qui définit les méthodes d'écoute de l'interface en question, méthodes dont le rôle est de traiter l'événement.

6.1.1 Exemple introductif

Prenons l'exemple d'une application simple : une fenêtre, un bouton. Le fait de cliquer sur le bouton permet de quitter l'application.

```
import java.awt.*;
import java.awt.event.*;
public class MonPremierEvenement extends Frame
                          implements ActionListener {
   private Button b;
   public MonPremierEvenement() {
      b = new Button("Quitter");
```

```
      add("South", b);
      b.addActionListener(this);
   }
   public void actionPerformed(ActionEvent ae) {
      System.exit(0);
   }
   public static void main(String[] args) {
      MonPremierEvenement mf = new MonPremierEvenement();
      mf.setTitle("Titre");
      mf.pack();
      mf.setVisible(true);
   }
}
```

Cette façon de procéder marche bien ici, mais que se passe-t-il si l'on dispose de plusieurs boutons ? C'est là que l'objet passé en paramètre intervient :

```
import java.awt.*;
import java.awt.event.*;
public class MonPremierEvenement extends Frame
                        implements ActionListener {
   private Button b1;
   private Button b2;
   public MonPremierEvenement() {
      b1 = new Button("Quitter");
      add("South", b);
      b1.addActionListener(this);
      b2 = new Button("Rien");
      add("North", b);
      b2.addActionListener(this);
   }
   public void actionPerformed(ActionEvent ae) {
      String nom;
      nom = ae.getActionCommand();
      if (nom.equals("Quitter")
         System.exit(0);
   }
   public static void main(String[] args) {
      MonPremierEvenement mf = new MonPremierEvenement();
      mf.setTitle("Titre");
      mf.pack();
      mf.setVisible(true);
   }
}
```

6.2 Les mouchards

6.2.1 `ActionListener`

Un tel mouchard possède une seule méthode, `actionPerformed(ActionEvent)`. Les objets susceptibles de générer ce type d'événements sont les suivants :

- `Button` : si l'on a cliqué sur le bouton
- `List` : lors d'un double-clic sur un article d'une liste ou bien si la touche `return` est enfoncée tandis qu'un article est sélectionné ;
- `MenuItem` : lorsqu'un article de menu est sélectionné ;
- `TextField` : lorqu'un caractère est frappé.

6.2.2 `ItemListener`

Ce genre de mouchard contrôle des événements de type `ItemEvent` au moyen de la méthode `itemStateChanged(ItemEvent)`. Les objets susceptibles de générer ce genre d'événements sont d'un des types suivants :

- Checkbox : lorsque l'état d'une case à cocher est modifié ;
- CheckboxMenuItem : Lorsque l'état d'un article de menu de type *case à cocher* est modifié ;
- Choice : lorsque la sélection est changée.

Les méthodes applicables à l'objet de type `ItemEvent` passé en paramètre sont les suivantes :

- `getItem()`, qui renvoie l'objet dont l'état a été modifié ;
- `getStateChange()`, qui renvoie l'une des deux constantes `SELECTED` ou `DESELECTED`, selon le type de changement d'état effectué.

6.2.3 `WindowListener`

Ces mouchards surveillent les événements de type `WindowEvent`, et ce au moyen de plusieurs méthodes, selon l'événement survenu. Ces méthodes sont les suivantes :

- `windowClosing(WindowEvent)` : lorsqu'une fenêtre est en cours de fermeture (on vient de cliquer sur l'icône de fermeture de la fenêtre) ;
- `windowOpened(WindowEvent)` : lorsqu'une fenêtre vient d'être ouverte ;
- `windowIconified(WindowEvent)` : lorsqu'une fenêtre est iconifiée ;
- `windowDeiconified(WindowEvent)` : lorsqu'une fenêtre est désiconifiée ;

- `windowClosed(WindowEvent)` : lorsqu'une fenêtre vient d'être fermée ;
- `windowActivated(WindowEvent)` : lorsqu'une fenêtre a été rendue active ;
- `windowDeactivated(WindowEvent)` : lorsqu'une fenêtre a été rendue inactive.

Les seules classes pouvant générer ce type d'événements sont les suivantes :

- `Dialog`
- `Frame`

6.2.4 `ComponentListener`

Les classes implantant cette interface contrôlent les événements de type `ComponentEvent`. Ils sont générés par les classe `Dialog` et `Frame`, et sont traités par l'une des méthodes suivantes :

- ComponentMoved(ComponentEvent) ;
- ComponentHidden(ComponentEvent) ;
- ComponentResized(ComponentEvent) ;
- ComponentShown(ComponentEvent).

La méthode `getComponent()` de la classe `ComponentEvent` permet de connaître le composant auquel l'événement est associé.

6.2.5 `MouseListener`

Les mouchards de ce type gèrent une partie des événements de type `MouseEvent` au moyen des méthodes suivantes :

- `mousePressed(MouseEvent)` : lorsque le bouton de la souris a été enfoncé ;
- `mouseReleased(MouseEvent)` : si le bouton de la souris a été relaché ;
- `mouseEntered(MouseEvent)` : lorsque le pointeur de la souris vient d'entrer dans le composant générateur de l'événement ;
- `mouseExited(MouseEvent)` : lorque le pointeur de la souris vient de quitter le composant générateur de l'événement ;
- `mouseClicked(MouseEvent)` : lorsque l'on a cliqué sur la souris (bouton enfoncé puis relaché).

Ces événements peuvent être générés par l'une des classes suivantes :

- `Canvas` ;

- `Dialog` ;
- `Frame` ;
- `Panel` ;
- `Window` ;

Les méthodes applicables à un événement de type `MouseEvent` sont les suivantes :

- `getClickCount()` : pour connaître le nombre de clics associés à l'événement ;
- `getPoint()`, `getX()`, `getY()` : pour connaître les coordonnées du pointeur de la souris, relativement au composant ayant généré l'événement.
- `isPopupTrigger()` : pour savoir si cet événement est celui généré par un menu popup.

6.2.6 `MouseMotionListener`

Les classes implantant cette interface gèrent les deux autres événements souris possibles, contrôlés par les méthodes suivantes :

- `mouseDragged(MouseEvent)` : si la souris a été déplacée bouton enfoncé ;
- `mouseMoved(MouseEvent)` : si la souris a été déplacée bouton relaché.

Bien évidemment, les méthodes qui s'appliquent aux objets MouseEvent passés en paramètre à ces methodes sont les mêmes que pour les méthodes de l'interface `MousListener`.

6.2.7 `AdjustementListener`

Les événements que contrôle cette interface sont ceux provenants des ascenseurs (`Scrollbar`). Ils sont de type `AdjustementEvent` et sont gérés par la méthode `adjustementValueChanged`.

Les méthodes qui s'appliquent à un objet de type `AdjustementEvent` sont les suivantes :

- `getAdjustable()` : renvoie l'objet émetteur de l'événement ;
- `getAdjustementType()` : renvoie le type de déplacement ayant eu lieu. Les types en question sont les suivants :
  - `BLOCK_INCREMENT`
  - `BLOCK_DECREMENT`
  - `UNIT_INCREMENT`

- UNIT_DECREMENT
- TRACK

- `getValue()` : pour connaître la nouvelle valeur représentée par l'ascenseur.

6.2.8 KeyListener

6.2.9 FocusListener

6.2.10 ContainerListener

6.2.11 TextListener

6.3 Les adaptateurs

6.3.1 Principe

Dans l'exemple introductif, le conteneur auquel appartenaient les composants générateurs d'événements implantait directement l'interface du mouchard associé. Cependant, ce procédé peut s'avérer assez peu lisible. En général, on préfère définir une classe (interne en général) qui joue le rôle d'*adaptateur*. Elle hérite d'une classe *adaptateur* adéquate qui implante l'interface associée aux événements à filtrer, et se charge d'appeler les méthodes adéquates.

En pratique, si l'on a besoin d'implanter toutes les méthodes d'une interface, alors l'adaptateur implantera l'interface en question. Sinon, il héritera plutôt d'une classe *adaptateur* par défaut.

6.3.2 Exemple

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

public class MonAdaptateur extends Frame {
   private Label affichage;
   private Button quitter;
   private Button bouton1;
   private Button bouton2;

   public MonAdaptateur() {
      quitter = new Button("Quitter");
      bouton1 = new Button("Bouton 1");
      bouton2 = new Button("Bouton 2");

      affichage = new Label();

      MonAdaptateur1 ma = new MonAdaptateur1();
```

```
      quitter.addActionListener(ma);
      bouton1.addActionListener(ma);
      bouton2.addActionListener(ma);

      BorderLayout border = new BorderLayout();
      add("North", affichage);
      add("West", bouton1);
      add("Center", quitter);
      add("East", bouton2);

      addMouseListener(new MonAdaptateur2());
   }

   class MonAdaptateur1 implements ActionListener {
      public void actionPerformed(ActionEvent ae) {
         String nom = ae.getActionCommand();
         if (nom.equals("Bouton 1"))
            affichage.setText("Bouton 1");
         else if (nom.equals("Bouton 2"))
            affichage.setText("Bouton 2");
         else if (nom.equals("Quitter"))
            System.exit(0);
      }
   }

   class MonAdaptateur2 extends MouseAdapter {
      public void mouseEntered(MouseEvent me) {
         affichage.setText("Dedans");
      }
      public void mouseExited(MouseEvent me) {
         affichage.setText("Dehors");
      }
   }

   public static void main(String[] args) {
      MonAdaptateur ma = new MonAdaptateur();
      ma.pack();
      ma.show();
   }
}
```