Chapitre I: Pourquoi des objets ?

1. Introduction

Ce cours n'est pas conçu dans l'intention de débattre des différents paradigmes de programmation et de leurs intérêts respectifs. L'existence de ces paradigmes et les succès relatifs des langages qui s'en inspirent prouve que les domaines d'application de l'informatique sont suffisamment étendus pour laisser de nombreux champs d'application possibles à chacun d'eux.
Il est simplement à savoir que la naissance d'un nouveau paradigme vient évidemment des limites atteintes par les langages qui sont associés aux paradigmes courants. La programmation impérative a bien répondu aux attentes, dès lors que les applications développées ne comptaient guère plus de quelques milliers de lignes et étaient le produit d'un ou deux programmeurs. Plusieurs pistes ont été et sont toujours explorées parmi lesquelles, celle de la programmation logique chère à tous ceux qui s'intéressent à l'intelligence artificielle et aux domaines dans lesquels la quantité de données à traiter est gigantesque.
Mais il y a d'autres aspects.
Ainsi, l'augmentation phénoménale de la complexité des logiciels fait qu'aujourd'hui, le nombre de lignes de code s'élève très souvent à des dizaines de milliers. Ce sont des équipes de programmeurs qui travaillent à la réalisation de ces logiciels et il faut éviter des pertes de temps dues à d'éventuelles synchronisations du travail. Chacun doit pouvoir travailler dans son coin en faisant confiance aux autres et en se montrant digne de la confiance des autres. En conséquence, des impératifs nouveaux apparaissent.
Il devient nécessaire :

- ✓ de penser les traitements d'une manière plus naturelle, plus proche de la réalité[1] afin de dégager le programmeur de tout une série de contraintes opérationnelles;
- ✓ d'encapsuler les traitements dans des modules qui soient des boîtes noires équipées d'une interface, permettant au programmeur de les utiliser sans savoir comment ces traitements sont implantés, mais connaissant les éventuelles données à fournir et les éventuels résultats retournés;
- ✓ d'évoluer ainsi vers un développement qui favorise la réutilisation logicielle et l'adaptabilité.

C'est essentiellement de ces nécessités qu'est née la programmation orientée objet. Cela explique notamment que ce n'est pas un programme d'addition de nombres saisis au clavier jusqu'à ce qu'un signal soit donné (valeur spéciale ou bidon) qui vous convaincra du bien-fondé de ce type de programmation et cela, même si un tel programme peut être écrit au moyen d'objets.
Quant à la programmation impérative, elle a encore de beaux jours devant elle, ne fusse que par la nécessité de maintenir une quantité de programmes utiles et qui fonctionnent[2].
Le but de ce cours n'est pas de vous convaincre que la programmation orientée objet est la solution à tous les problèmes, mais seulement de vous faire percevoir quels en sont les mécanismes fondateurs afin de vous permettre de juger, dans quelles situations celle-ci se révèle intéressante. Toutes ces choses étant dites, il faudra sans doute perdre un peu les habitudes de programmation pour bien cerner tout le profit que vous pourrez tirer de cette autre manière de programmer.

[1] Dans certains cas, un raisonnement déterministe n'est pas toujours celui qui est le plus approprié.
[2] Une entreprise se moque pas mal de faire réécrire des programmes qui fonctionnent bien dans de nouveaux langages, dans la mesure où la réécriture est une source de problèmes potentiels.

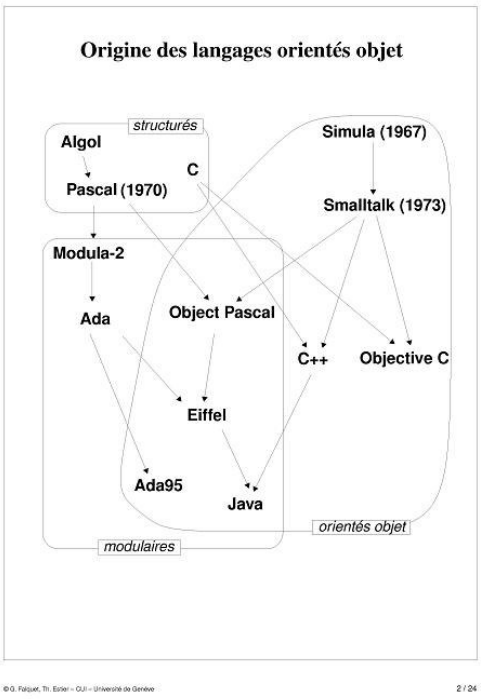
2. Le choix d'un langage

Il n'est pas possible de parler langage de programmation sans en choisir un en particulier. Pour un cours sur la programmation impérative le langage le plus souvent utilisé est le langage Pascal pour développer des applications. Beaucoup d'experts du domaine de l'enseignement estimaient, sans doute à raison, que ce langage était pédagogique, même s'il n'offrait pas les facilités de développement de certains autres langages. Nous utiliserons principalement le langage Java. En voici, les raisons principales :

- ✓ Java est un langage de programmation qui met en oeuvre assez clairement les mécanismes fondamentaux de la programmation orientée objet que sont l'héritage, la surcharge et le polymorphisme;
- ✓ Java demande de la rigueur dans la structure et l'écriture des programmes et les implicites y sont moins nombreux que chez la plupart des langages de script (Javascript, Perl, Php, Python,...);
- ✓ Java a grandi avec le Web, ce qui le rend bien adapté à son exploitation;
- ✓ un programme écrit en Java et compilé en octets de code peut tourner sur n'importe quel type de matériel et n'importe quel type de plateforme.

Il est aussi malheureusement clair que les cours qui sont prévus s'avèreront insuffisants pour couvrir un aperçu assez large de ce que propose Java.

Voici un schéma donnant l'évolution des langages impératifs vers l'orienté objet :



3. L'installation

Cette section est un peu plus technique mais il faut bien décrire quelque part, comment acquérir une version du langage Java et comment l'installer.
Je vous suggère d'installer Java2 SDK (Software Development Kit) version 5 update 4. Cette version est actuellement appelée Java2 Platform Standard Edition 5. Vous le trouverez à la source c'est-à-dire à l'adresse http://java.sun.com/j2se/1.5.0/download.jsp et cela, que votre OS préféré soit Windows, Linux ou même Solaris. Un exécutable d'environ 56 Mo est à télécharger. Son exécution installe le langage et toute la hiérarchie des librairies dont il a besoin. Sous Windows, le dossier par défaut est C:\Program Files\Java\jdk1.5.0 mais il est possible de faire un autre choix de dossier[3].

4. La machine Java virtuelle (JVM)

La description qui suit n'est pas nécessaire à la compréhension du prochain chapitre. Sa lecture peut être différée. Cependant, on évoque tellement souvent la machine virtuelle qu'est utile d'en donner une explication à ce stade à l'intention de ceux qui connaissent déjà un peu les tenants et aboutissants de la programmation orientée objet.
Une des préoccupations qui se fait jour au moment du développement du projet Java, c'est la portabilité des applications sur différentes plate-formes. C'est une exigence qui naît du développement d'Internet mais aussi de l'évolution de l'architecture informatique de nombreuses grosses entreprises, de systèmes centralisés vers des systèmes distribués. La variété du matériel et des systèmes d'exploitation en interconnexion impose que les applications développées dans des langages modernes puissent être hébergées sur des systèmes très hétéroclites.
Comment cette caractéristique de portabilité est-elle obtenue? Que représente cette machine virtuelle qu'on évoque pour expliquer cette qualité du langage. En voici une très brève explication.
L'installation de Java dont il vient d'être question consiste notamment en la fourniture :

- ✓ d'un grand nombre de classes prédéfinies
- ✓ de plusieurs outils indispensables (compilateur, interpréteur, générateur de documentation, débogueur,...)

[3] Il existe une autre solution: installer Jbuilder 5 Personal un produit de Borland qui offre d'excellentes possibilités d'édition de programmes en Java. Ce logiciel contient une version du JDK 1.3 (Java Development Kit). Cette version est gratuite pour autant que vous respectiez le contrat de licence qui vous est proposé. Elle est téléchargeable à partir de l'adresse http://www.borland.com/jbuilder/ . Le fichier compressé à télécharger est de 40 Mo pour la version Windows et de 53 Mo pour la version Linux.
Bien entendu, il est tout à fait possible de rédiger du code Java en utilisant le simple bloc- notes de Windows ou un éditeur de texte élémentaire. Un produit comme Jbuilder fournit aux développeurs des tas de services intéressants allant de la coloration syntaxique à la génération de documentation de programme en passant par l'indentation automatique, une gestion souple des fichiers etc. La relative convivialité du produit parle en la faveur de son auto-découverte car soyons clairs, la connaissance d'un tel outil dédié aux développeurs professionnels n'est pas l'objectif du cours. Son étude nous éloignerait d'ailleurs de l'essentiel.
A vous de voir si l'installation de Jbuilder, à terme, vous sera utile. Tenez compte également des performances de votre matériel.

Chapitre I: Pourquoi des objets ?

Pourquoi un compilateur et un interpréteur ?
Le principe est le suivant: sur un système donné, le compilateur transforme le code source (texte des programmes) en « octets de code » (byte codes en anglais). Il s'agit d'un langage universel que chacun des systèmes s'engage à déchiffrer au moyen de son interpréteur spécifique capable de digérer ces octets de code en tenant compte du contexte local.
Les byte codes générés par la machine A gouvernée par l'OS A' peuvent maintenant être interprétés par l'interpréteur de la machine B gouvernée par l'OS B'.
En d'autres termes, un même programme, compilé par différents systèmes, fournit les mêmes byte codes. A l'exécution du programme sur un système donné, ces byte codes sont interprétés grâce à un interpréteur propre au système. Le couple constitué de ce système donné et d'une instance de l'interpréteur est appelé machine virtuelle. Un même système peut donc être à la base de l'existence de plusieurs machines virtuelles[4].
Nous verrons par la suite que pour qu'un programme puisse être exécuté, il faut qu'existe une classe exécutable. Elle doit être enregistrée dans un fichier qui porte le même nom qu'elle suivi de l'extension java.
Si la classe exécutable s'appelle Application, elle doit être enregistrée dans un fichier qui s'appelle Application.java.
Le code contenu dans ce fichier (et dans les fichiers qui lui sont liés[5]) est compilé grâce à la commande javac suivie du nom du fichier. La commande javac correspond donc à l'activation de l'outil de compilation. Le fichier résultant portera le nom de la classe suivi cette fois de l'extension class.
La compilation du fichier Application.java se fait par la commande javac Application.java et produit un fichier Application.class.
Le code de cette classe est exécuté en utilisant la commande java suivi du nom de ce dernier fichier, sans l'extension. Elle correspond à l'activation de l'interpréteur de byte codes.
La commande java Application produit l'exécution du programme[6].
Toutes les commandes qui viennent d'être évoquées sont données au niveau d'un shell de commandes du système d'exploitation utilisé et fonctionnent sous réserve d'un choix correct des paramètres concernant les variables d'environnement (notamment les chemins que l'OS doit suivre pour trouver les programmes et ceux que les programmes doivent suivre pour trouver le code des applications).

[4] Cette opportunité nous sera de peu d'utilité tant que nous ne développons pas des applications qui interagissent entre elles.
[5] Il est clair que la définition d'une classe, y compris celle d'une classe exécutable, fait généralement appel à d'autres définitions de classes contenues dans d'autres fichiers. La compilation consiste à générer des bytecodes qui intègrent les informations contenues dans ces différents fichiers.
[6] Cette commande peut comporter d'autres paramètres. En particulier, nous verrons qu'il est possible, et même souhaitable, de définir dans les différentes commandes les chemins d'accès aux fichiers. Pour ce qui est du lancement d'une JVM, on peut aussi fournir des paramètres qui sont récupérables par le programme.

Chapitre II: Dialogue des objets

1. Une approche naturelle

Un des reproches que l'on peut faire à la programmation impérative est d'éloigner le programmeur du monde réel en l'obligeant à décrire, d'une manière très élémentaire, des actions et des traitements qui sont parfois très complexes. Les limites de ce type de démarche sont atteintes dès l'instant où l'ensemble de ces instructions devient pléthorique. Il devient alors difficile de contrôler l'ensemble des modules d'une application, même lorsque l'analyse de celle-ci a fait l'objet d'une approche descendante. Une autre critique importante revient régulièrement lorsqu'on parle de programmation impérative: l'application programmée répond généralement à une vision, espérons-le correcte, que l'on a des traitements au moment du développement de celle-ci. La conséquence, c'est que les programmes rédigés manquent souvent d'adaptabilité et que le monde est (trop) souvent à refaire.
La programmation orientée objet trouve une partie de son intérêt dans la réponse qu'elle apporte à la difficulté qui vient d'être évoquée. Pour prendre un exemple simple, un patron ne peut gérer une très grosse entreprise en voulant s'occuper des moindres détails de son fonctionnement, des grandes décisions stratégiques, jusqu'à l'achat des produits d'entretien des locaux, en passant par la paie des ouvriers, par exemple. Il doit pouvoir faire confiance à ses subordonnés et se limiter à une interaction avec eux. C'est un peu l'idée de la programmation orientée objet d'arriver à constituer des logiciels basés sur l'interaction entre les objets, chacun pouvant demander à d'autres d'effectuer certaines tâches pour la réalisation desquelles eux-mêmes s'adresseront à d'autres objets et ainsi de suite. Un des principes sur lesquels nous reviendrons est basé sur le fait qu'une relation de confiance existe entre les objets. Chacun d'entre eux propose et s'engage à rendre au monde extérieur (les autres objets potentiels) un ensemble de services. Il le fait via une interface d'interaction qui ne permet pas de savoir comment ces services sont rendus.
Le déroulement d'une application ressemblera à une conversation entre objets[14]. Ceux-ci vont donc s'envoyer mutuellement des messages qui seront autant de demandes de sous-traitance. Si un objet ne peut réaliser une partie du travail qui lui est demandé, il devra pouvoir confier cette partie du travail à un autre objet. La conversation entre objets sera donc aussi constituée de réponses que les objets s'enverront les uns les autres. Pour établir une comparaison avec des choses connues, une demande de service d'un objet à un autre est une sorte d'appel de fonction. Dans le vocabulaire de la POO, on parlera d'appel de méthodes.

[14] Il faudra bien qu'un objet engage la conversation!

2. L'encapsulation

C'est un des principes essentiels de la POO. Un objet est caractérisé par son comportement et par son état. Son comportement est décrit par les services qu'il peut rendre et donc, d'une certaine manière par les fonctions qu'il peut remplir. Nous avons déjà signalé qu'on parlait ici de méthodes plutôt que de fonctions. L'état d'un objet est ce qui le caractérise à un moment donné. Cet état est variable, dépend de ses comportements et des comportements des autres objets. La structure d'un objet ne lui est généralement pas propre. Il fait partie d'une classe d'objets qui la partagent. Il en est ainsi dans le monde réel. Une personne, une voiture, un chien, une carte d'identité,... possèdent généralement une série de propriétés communes, même si les valeurs qui y sont attachées sont différentes. Les personnes n'ont pas le même nom, les chiens ne sont pas de la même race, les cartes d'identité ne contiennent pas la même photo,...
Ils ont également une série de comportements identiques. Les personnes parlent, les chiens aboient, les voitures peuvent freiner et accélérer,... Il n'est pas du tout souhaitable que chaque objet puisse modifier anarchiquement et directement l'état des autres objets. Il est préférable de réserver la modification de l'état à l'objet lui-même. Ceci n'implique pas qu'un objet puisse demander cette modification mais il devra le demander à l'objet lui-même qui pourra, notamment, s'assurer que cette modification est valide. Il vaut mieux, en effet, qu'un seul objet prenne cette responsabilité en charge. Ces remarques vont nous conduire à définir, plus tard, la portée des variables et des méthodes. Si le comportement d'un objet est caractérisé par les méthodes qu'il supporte, son état est principalement décrit par des variables propres appelées variables d'instances. Pour désigner cette association de l'état d'un objet à son comportement, on parle d'encapsulation. Les données et les traitements ne semblent plus séparés mais donnent l'impression d'être regroupés à l'intérieur de l'objet lui-même. L'encapsulation présente un avantage incontestable. Lorsqu'un programmeur utilise des classes qu'il n'a pas développées lui-même et en crée des instances, il ne connaît pas la manière dont les traitements correspondant à l'exécution des méthodes sont implantés. La seule chose que celui-ci connaisse, c'est l'ensemble des services rendus par la classe, le contrat qu'elle s'engage à remplir en quelque sorte[15]. Cet avantage n'est pas à négliger lorsqu'il s'agit d'évoluer vers un type de programmation dont une partie importante de la démarche consiste à réutiliser des modules (logiciels) existants.
Un exemple simple : considérons un carré.
On peut imaginer que son état soit décrit par deux données variables que sont la longueur de son côté et la coordonnée dans un repère défini de son coin supérieur gauche.
Le fait d'évoquer un carré contient implicitement la référence à un objet de la classe des carrés. On peut demander beaucoup de services à un objet de type carré. Les plus évidents sont sans doute :

- ✓ calculer et fournir son périmètre,
- ✓ calculer et fournir son aire,
- ✓ se dessiner,
- ✓ changer ses propres dimensions ou sa position,
- ✓ fournir certains de ces renseignements à la demande, ...[16]

[15] Dans le contexte d'une bonne programmation, les classes sont généralement bien documentées, qu'il s'agisse de décrire le contrat rempli par chacune des méthodes ou de détailler la nature des autres membres de la classe.
[16] Dans une perspective graphique plus poussée, on pourrait considérer que le carré possède un fond coloré, un bord ayant lui-même une épaisseur et une couleur, etc.

Lorsqu'on demandera à un objet carré de changer la valeur de la longueur de son côté, celui-ci pourra vérifier que la valeur fournie est cohérente et faire en sorte de corriger les éventuelles erreurs pour que les données soient consistantes. Par exemple, si le paramètre fourni pour la longueur du côté est négatif, l'objet pourra décider de lui donner la valeur 0. Le fait de confier cette tâche à l'objet lui-même combiné au fait que l'accès aux données peut lui être réservé, empêche tout objet (logiciel[17]) extérieur de rendre le système de données inconsistant.

3. Les classes d'objets et les mécanismes de la POO

En programmation orientée objet, chaque objet fait partie d'une classe dont la description mentionne les données et le comportement qui le caractérisent ainsi que ses semblables. En suivant l'exemple, la description de la classe des carrés mentionnera l'existence de deux champs de données: la longueur du côté et la position du sommet supérieur gauche. Elle précisera les méthodes disponibles, celle qui calcule et affiche le périmètre, celle qui modifie la position du sommet, ... en indiquant chaque fois le nombre et le type des paramètres à fournir.
Un programmeur qui utilisera la classe « carré » saura qu'il peut en obtenir le périmètre sans savoir comment le calcul est effectué[18]. Le mot ``type'' est bien connu des programmeurs. La programmation orientée objet fera généralement apparaître deux sortes de types: les types primitifs (entiers, caractères, booléens,...) et les types construits (ceux qui correspondent aux classes d'objets). Ainsi, un objet peut être déclaré de type carré, cela signifiera qu'il est caractérisé par tel et tel types de données et par telle et telle méthodes. Notre exemple peut constituer une application directe de ce qui précède. Nous pouvons décider que le type de donnée pour la longueur du côté est ``nombre entier'' qui est un type primitif et que le type de donnée pour le sommet est point ou point constitue une autre classe d'objets.
Toutes ces possibilités prendront forme dans le chapitre suivant consacré à Java. En attendant, et pour montrer que la programmation orientée objet fait la part belle à l'abstraction, nous pourrions encore avoir une autre vue des choses et décider de créer d'abord une classe point qui contiendrait, par exemple, deux données de type nombre entier (une abscisse et une ordonnée). Nous choisirions alors de dériver la classe carré de la classe point en précisant qu'un carré, c'est d'abord un point (son sommet supérieur gauche) avec en plus, la longueur d'un côté (ce qui permettrait de le définir complètement). Ceci permet d'amorcer la présentation d'un des mécanismes classiques de la POO à savoir l'héritage.
L'héritage est ce que l'on a inventé de mieux pour justement ne pas devoir tout réinventer. En se basant sur le principe qui veut que ce qui a déjà été défini peut être réutilisé ou légèrement modifié et que le reste peut être ajouté, une classe d'objets peut hériter d'une autre classe en la spécialisant. Cette spécialisation consiste :

- ✓ à rajouter des choses qui n'existaient pas dans la définition de la classe précédente, à savoir des champs de données ou des méthodes;

[17] De par le fait qu'un objet encapsule ses données et ses méthodes, on assimile parfois le mot ``objet'' au mot ``logiciel''. Certains objets peuvent en effet être d'une grande complexité.
[18] Dans ce cas-ci, ça peut paraître évident. Notez toutefois que le calcul pourrait consister à faire multiplier l'aire du carré par 16 et prendre la racine carrée du résultat. La seule chose qui intéresse l'utilisateur (programmeur) c'est la garantie d'obtenir la valeur du périmètre.

- ✓ à redéfinir certaines méthodes en leur faisant faire les choses de manière différente, tout en se ménageant la possibilité de considérer l'objet comme faisant aussi partie de la classe d'héritage.

On peut aussi envisager les choses d'une autre manière en considérant que des objets appartenant à des classes différentes ont des points communs qui pourraient permettre de les considérer comme faisant tous partie d'une classe d'objets plus générale. L'héritage s'examinera donc souvent dans le sens de la spécialisation mais parfois aussi dans celui de la généralisation. Cette petite réflexion nous permet aussi d'amorcer la description d'un autre mécanisme fondamental: le polymorphisme.

Si la classe d'un objet hérite d'une autre classe, tout objet de la première est aussi un objet de la seconde[19]. De ce fait, l'objet a plusieurs formes possibles et on peut donc l'invoquer en le considérant de différentes manières. C'est pourquoi on parle de polymorphisme.

Il existe en POO d'autres mécanismes tels, par exemple, la surcharge. Un résultat peut être renvoyé par une méthode sur base de données différentes. Ainsi, le calcul de l'aire du carré peut être réalisé sur base de la connaissance de la longueur du côté de celui-ci, mais aussi, pourquoi pas, sur base de la connaissance des coordonnées de deux sommets opposés. Dans le premier cas, l'unique paramètre est un simple entier alors que dans le second, les deux paramètres sont des objets « points ». Pour qu'un compilateur puisse identifier une méthode sans équivoque, il se réfère à sa signature qui comprend nécessairement le nom de la méthode mais aussi la liste (éventuellement vide) des paramètres et de leurs types. C'est en ce sens qu'on parle de surcharge de la méthode. On peut donc dire qu'un même nom de méthode dissimule des sémantiques différentes[20]. Nous avons évoqué le mécanisme d'héritage. Beaucoup de langages de POO implantent un mécanisme d'héritage multiple. Une classe peut hériter de plusieurs autres classes au sens où nous l'avons défini. La sous-classe hérite alors de caractéristiques de toutes les classes dont elle hérite. Cela ne va pas sans poser un certain nombre de problèmes[21]. Cette première approche devrait suffire à comprendre dans quel esprit fonctionne la POO. L'algorithmique reste bien présente, notamment au niveau de la description des méthodes. Néanmoins, tout code préalablement rédigé et qui fonctionne, c'est-à-dire qui remplit son contrat, ne doit plus être pris en compte par celui qui utilise l'objet. Cette approche autorise, vous l'aurez compris, un meilleur débogage et une meilleure localisation des problèmes.

Dans le chapitre suivant, nous vous proposons de découvrir comment Java, un langage qualifié de totalement orienté objet, implante les concepts qui viennent d'être décrits. Ce sera aussi l'occasion de découvrir la syntaxe de ce langage et de souligner quelques bonnes habitudes rédactionnelles. A la fin du syllabus, nous tentons une comparaison, si tant est qu'elle ait du sens, entre Java et JavaScript que l'on pourrait qualifier de faiblement orienté objet.

[19] Attention, l'inverse n'est pas vrai car la classe qui hérite ``spécialise'' la classe d'héritage. Il est fort probable, par exemple, qu'elle supporte des méthodes que la précédente ignore.

[20] Dans notre exemple, le résultat renvoyé est un entier qui représente l'aire du carré mais rien n'empêche que la sémantique même des résultats soient tout à fait différentes selon le nombre et le type des paramètres de la méthode.

[21] Le langage Java, que nous étudierons au prochain chapitre, n'implante pas directement la notion d'héritage multiple. Une classe ne peut hériter que d'une seule autre classe. Néanmoins, le concept d'interface autorise sa simulation en en supprimant les désavantages.

4. Exercice unique

En vous basant sur l'exemple du carré dont les sommets sont des points qui ont chacun une coordonnée, identifiez, dans les domaines que vous connaissez, des objets dont la définition fait éventuellement appel à d'autres objets. Imaginez, pour ces objets et ceux qui les composent, des méthodes que l'on pourrait invoquer auprès d'eux en précisant ce que ces méthodes sont sensées effectuer comme traitement ou renvoyer comme résultat.
Pour que les choses soient claires, voici d'autres orientations possibles pour étoffer l'exemple du carré et des points. Elles seront traitées dans les chapitres qui suivent. Vous devriez pouvoir tenir un discours comme celui qui suit, discours qui vous permettrait de dégager les classes, les champs de données et leurs types (donc éventuellement d'autres objets) ainsi que quelques méthodes supportées par ces classes d'objets.
Une droite est constituée de points. Deux points permettent de définir une droite, mais il y peut y avoir d'autres moyens. On peut demander à une droite de changer de direction, de glisser parallèlement à elle-même. Pour autant que l'on ait défini ce qu'est un cercle, on peut demander à une droite si elle est tangente à un cercle donné, si elle comprend un point donné, etc. De la description qui précède, on peut retenir qu'à tout le moins, il existe des points, des droites et des cercles. Pour ce qui est des droites, on peut imaginer une méthode qui répondra par vrai ou faux à la question: comprends-tu tel point? Il s'agira d'un résultat. Une autre méthode ne fournira pas de résultat mais modifiera les paramètres de la droite lorsqu'on lui demandera de glisser parallèlement à elle-même pour passer par un point donné, etc.

1. Premier programme

Voici un exemple de programme qui affiche dans une fenêtre console, le texte : « Mon premier programme JAVA ».

```
public class PremProg {
   public static void main (String args[]) {
      System.out.println (″Mon premier programme JAVA″) ;
   }
}
```

1.1. Structure générale du programme

```
public class PremProg {
   …
}
```

Elle correspond à la définition d'une classe nommé *PremProg*. La première ligne identifie la classe ; elle est suivie d'un bloc { … }, c'est-à-dire des instructions qui définissent le contenu de la classe. Dans notre cas, seule une méthode particulière *main* est définie.
La structure de cette méthode est identique à la structure de la définition de la classe.

```
public static void main (String args[]) {
   System.out.println (″Mon premier programme JAVA″) ;
}
```

La première ligne identifie la méthode ; elle est suivie d'un bloc qui en fournit les différentes instructions.
Cette première ligne sera toujours présente lors de programme en Java.
Le mot *public* permet à la machine virtuelle de pouvoir accéder à la méthode *main*.
Le mot *static* précise que la méthode *main* n'est pas liée à une instance particulière de la classe. C'est ce qui fait que cette méthode est l'équivalent d'une procédure ou une fonction dans un autre langage de programmation. Comme dans notre cas, elle s'appelle *main* c'est la fonction principale (l'équivalent du programme principal en Pascal).
L'argument *String args[]* ou *String [] args* permet de récupérer des arguments transmis au programme au moment de son lancement.
Le mot *void* précise que cette méthode ne renvoie rien. Il est tout à fait normal que la méthode principale ne renvoie aucune valeur ou résultat.

1.2. Contenu du programme

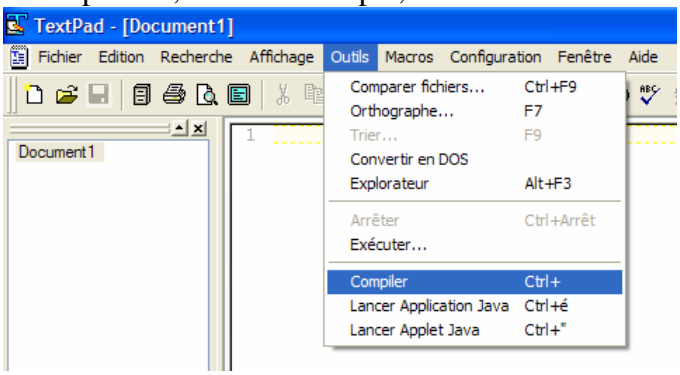
Notre programme ne comporte qu'une seule ligne :

```
System.out.println (″Mon premier programme JAVA″) ;
```

Nous verrons plus tard pourquoi les mots *System* et *out* sont présents et que seul *println* ne suffit pas.
Le texte qui apparaîtra à l'écran est mis entre guillemets.

2. Exécution d'un programme

A l'aide d'un éditeur (TextPad) qui possède en son sein un accès au compilateur et à l'interpréteur, il est très simple, car il suffit de choisir le chemin suivant :



Par contre, si aucun éditeur spécifique n'est installé et que vous utilisez le bloc note, il faut faire en sorte que les chemins soient définis au niveau de l'environnement.
La commande permettant de compiler sera *javac PremProg.java*.
Par contre, la commande permettant son exécution sera *java PremProg*.

3. Quelques instructions de base

Voici un programme simple au niveau programmation java :

```
public class Exemple1 {
   public static void main (String args[]) {
      int n ;
      double x ;
      n = 5 ;
      x = 2*n + 1.5 ;
      System.out.println (″n = ″ + n) ;
      System.out.println (″x = 2*n + 1.5 = ″ + x) ;
      double y ;
      y =n * x + 12 ;
      System.out.println (″valeur de y = n*x + 12 : ″ + y) ;
   }
}
```

Celui-ci peut être raccourci en plaçant en une ligne la déclaration du type et l'initialisation :

```
public class Exemple2 {
   public static void main (String args[]) {
      int n = 5 ;
      double x = 2*n + 1.5 ;
      System.out.println (″n = ″ + n) ;
      System.out.println (″x = 2*x + 1.5 = ″ + x) ;
      double y =n * x + 12 ;
      System.out.println (″valeur de y= n*x + 12 : ″ + y) ;
   }
}
```

Ce qui est important dans ces deux exemples c'est le fait que l'on ne doive pas déclarer au départ les variables. On les déclare au moment où on en a besoin. De plus, ces variables n'ont comme temps de vie que celui lié à la méthode *main*.
Il est à noter que l'appel à la méthode *println* ne peut se faire qu'avec des chaînes de caractères. Les variables écrites en paramètre sont converties en chaîne de caractères.
Ainsi l'opérateur « + » dans la méthode *println* est l'opérateur de concaténation.

Quand à la lecture, il n'existe pas de classe prédéfinie mais il est possible de la construire et de l'utiliser. Pour cela, il faut que la classe *Clavier* soit compilée et se trouve en byte code dans le dossier d'exécution du programme que l'on veut.

```
public class Exemple3 {
   public static void main (String args[]) {
      int x ;
      double y ;
      String z ;
      System.out.println (″Nous allons effectuer des tests de lecture d'un entier,″) ;
      System.out.println (″d'un réel et d'une chaîne de caractères″) ;
      x = Clavier.lireInt() ;
      y = Clavier.lireDouble() ;
      z = Clavier.lireString() ;
      System.out.println(″La valeur entière introduite au clavier est ″ + x) ;
      System.out.println(″La valeur réelle introduite au clavier est ″ + y) ;
      System.out.println(″La chaîne de caractères introduite au clavier est ″ + z) ;
   }
}
```

Voici un quatrième exemple qui concerne les boucles et les choix.

```
public class Exemple4 {
   public static void main (String args[]) {
      final int NFOIS = 5 ;
      int i ;
      double x ;
      double racx ;
      System.out.println (″Bonjour″) ;
      System.out.println (″Je vais calculer ″ + NFOIS + ″ racines carrees″) ;
      for (i=0 ; i < NFOIS ; i++) {
            System.out.println(″Donner votre nombre″) ;
            x = Clavier.lireDouble() ;
            if ( x < 0.0)
               System.out.println(x + ″ ne possède pas de racines carrées″) ;
            else {
               racx = Math.sqrt(x) ;
               System.out.println(x + ″ a pour racine carrée : ″ + racx) ;
            }
      }
      System.out.println (″Travail terminé – Au revoir″) ;
   }
}
```

Pour les boucles de type *pour*, il faut utiliser la structure suivante :
for (condition de début ; condition de fin ; incrémentation)
{ … }
Pour le test, la structure est la suivante :
if (test) { … } else { … }
ou encore
(test) ? <expression1> : <expression2> Cette structure doit renvoyer une valeur à une variable.
i++ est équivalent à i = i + 1 et i– – équivaut à i = i – 1, nous y reviendrons plus tard.

Il est possible et même vivement conseillé de placer des commentaires dans les programmes en Java. Il existe un créateur de documentation qui établit en fonction des commentaires écrits dans le programme une documentation des variables, constructeurs, méthodes utilisées dans la classe définie. Ce programme s'appelle *javadoc.exe*.
Pour placer des commentaires de ce type, il faut que ceux-ci soient précédés /** et se termine pas */.
Pour une seule ligne, on peut utilisé // ou pour un commentaire qui ne doit pas être repris par *javadoc.exe*, il suffit que le commentaire soit précédé par /* et se termine par */.

4. Les types primitifs de Java

Il existe quatre types primitifs de variables : entier, flottant, caractère et booléen.

4.1. Le type entier

4.1.1. Représentation en mémoire

Un bit est réservé au signe (0 pour les positifs et 1 pour les négatifs), les autres servent à représenter :

- ✓ Le nombre pour les positifs ;
- ✓ Le complément à deux du nombre pour les négatifs.

4.1.2. Les différents types entiers

Dans le cas d'une représentation en 8 bits, 1 octet, les valeurs minimales et maximales sont –128 et 127. La définition de ce type est *byte*.
Dans le cas d'une représentation en 16 bits, 2 octets, les valeurs minimales et maximales sont –32768 et 32767. La définition de ce type est *short*.
Dans le cas d'une représentation en 32 bits, 4 octets, les valeurs minimales et maximales sont –2 147 483 648 et 2 147 483 647. La définition de ce type est *int*.
Dans le cas d'une représentation en 64 bits, 8 octets, les valeurs minimales et maximales sont –9 223 372 036 854 775 808 et 9 223 372 036 854 775 807. La définition de ce type est *long*.

4.2. Le type flottant

Les types flottants permettent de représenter, de manière approchée, une partie des nombres réels. Ils s'inspirent de la notation dite scientifique ($1.5\ 10^{22}$ ou $0.472\ 10^{-8}$).
1.5 ou 0.472 représente la mantisse et 22 ou –8 représente l'exposant. Au niveau de la représentation, il y a a un certain nombre de chiffres qui apparaissent. C'est ce que l'on appelle la *précision*.

4.2.1. Les différents types flottants

Dans le cas d'une représentation en 32 bits, 4 octets, la précision est de 7 chiffres significatifs et les valeurs minimales et maximales sont $–1.402398^{E}–45$ et $3.402823^{E}38$. La définition de ce type est *float*.
Dans le cas d'une représentation en 64 bits, 8 octets, la précision est de 15 chiffres significatifs et les valeurs minimales et maximales sont $4.94065645841246^{E}–324$ et $1.79769313486231^{E}308$. La définition de ce type est *double*.

Si une variable est définie en *float*, pour lui attribuer une valeur, il faut dire au compilateur qu'il s'agit bien d'une valeur *float*, pour cela, il faut ajouter f à la fin du nombre : 1.5f

4.3. Le type caractère

Java écrit les caractères en mémoire à l'aide de 2 octets (16 bits), c'est-à-dire 65536 possibilités qui sont bien évidemment beaucoup plus que les 128 possibilités du code ASCII ou encore les 256 du code ASCII/ANSI (windows).

Pour déclarer une variable de type caractère, on utilisera *char c ;*

Pour attribuer une valeur à ce type de variable, on écrira : *c = 'a'*.
Il existe des notations conventionnelles qui en utilisant le caractère \ jouent le rôles de délimiteur.
Ainsi : \b ≡ retour arrière
\t ≡ tabulation horizontale
\n ≡ saut de ligne
\f ≡ saut de page
\r ≡ retour chariot

4.4. **Le type booléen**

Ce type sert à représenter une valeur logique du type *vrai/faux*.
En Java, il est possible d'affecter le résultat d'un test à un booléen.
Exemple :

```
boolean ordonne ;
ordonne = n<p
```

Dans ce cas, la variable *ordonne* reçoit la valeur de l'expression n < p.

5. Initialisation et constante

5.1. Initialisation d'une variable

Une variable peut recevoir une valeur initiale au moment de sa déclaration comme vu lors de l'exemple 2, mais rien n'empêche cette variable d'évoluer par la suite.

5.2. Cas des variables non initialisées

En Java, une variable n'ayant pas encore reçu de valeur ne peut être utilisée, sous peine d'aboutir à une erreur de compilation.

Exemple :

```
int n ;
System.out.println(″n = ″ + n) ;
```

Dans ce cas, il y a erreur lors de la compilation car *n* n'est pas définie.

```
int n ;
int p = n + 3 ;
n = 12 ;
```

Dans ce cas, il y a erreur lors de la compilation car *n* n'est pas définie.

```
int n ;
if (…) n = 30 ;
p = 2*n ;
```

Dans ce cas, il y a erreur lors de la compilation car *n* n'est pas dans tous les cas.

```
int n ;
if(…) n = 30 ;
   else n = 10 ;
p = 2*n ;
```

Il n'y a aucun problème, la compilation se passe bien.

Toutes ces considérations sont uniquement valables lorsqu'on parle de variables locales et non les variables d'instances des classes ou objets.

5.3. Le mot clé *final*

Ce mot permet de définir qu'une variable devienne une constante dès l'instant qu'elle reçoit une valeur.
Erreur de compilation dans les cas suivants :

```
final int n = 20 ;                  int p                              final int n ;
….                                  ….                                 ….
n = n+ 5 ;                  p = Clavier.lireInt() ;     n = Clavier.lireInt() ;
n = Clavier.lireInt() ;     final int n = 2 * p ;      …
                                    ….                                 n++ ;
                                    n++ ;
```

Cas où l'initialisation est différée et donc la compilation sera correcte. (A utiliser le moins possible à cause du manque de lisibilité)

```
final int n ;
….
if (…) n = 10 ;
   else n = 20 ;
```

6. Les opérateurs

6.1. Présentation des opérateurs

Comme dans la plupart des langages, Java possède

- ✓ deux opérateurs unaires (c'est-à-dire portant uniquement sur un seul opérande) : – et +, opérateur qui attribue le signe à une variable ou d'une expression ;
- ✓ Quatre opérateurs binaires (c'est-à-dire portant sur deux opérandes) : +, –, * et / qui effectue respectivement l'addition, la soustraction, la multiplication et la division de deux nombres.

Dans la plupart des langages, ces opérateurs ne fonctionnent que si les types des opérandes sont identiques, nous verrons plus tard, qu'en Java, par un jeu de conversion implicite, il sera possible d'obtenir un résultat.
Il existe l'opérateur modulo noté % mais qui contrairement à beaucoup de langages ne fonctionne pas uniquement qu'avec des entiers, il donne aussi un résultat dans le cas de flottants.
Exemple :
11%4 vaut 3
–11%–3 vaut –2
12.5%3.5 vaut 2
–15.2%7.5 vaut –0.2

Les règles de priorités de ces opérateurs sont similaires aux autres langages, la priorité absolue aux opérateurs unaires, suivi de *, / et % et ensuite des opérateurs + et –.

6.2. Les exceptions

Il existe des circonstances où un opérateur ne pourra fournir un résultat correct, on parle alors d'exception.

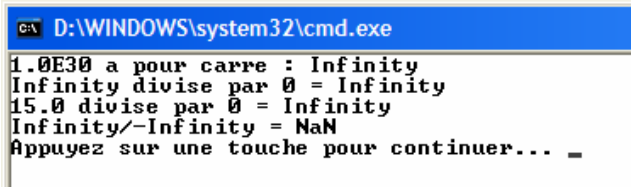
6.2.1. Les entiers

Le dépassement de capacité n'est jamais détecté, il se contente de conserver les bits les moins significatifs du résultat.
En revanche, la division par 0 conduit à une erreur d'exécution, plus précisément il y a un déclenchement d'une exception de type *ArithmeticException*. Nous verrons plus tard comment intercepter une telle exception et ce que l'on peut en faire.

6.2.2. Les flottants

En Java, aucune opération sur les flottants ne conduit à l'arrêt de l'exécution (pas même une division par zéro). Du fait qu'un flottant respecte la représentation IEEE 754, il existe un motif particulier pour représenter +∞, –∞ et une valeur non calculable. En Java, ces valeurs peuvent être affichées ou affectées à des variables.

Voici le résultat de l'exécution :



NaN : Not a Number

Exemple :

```
public class IEEE {
   public static void main (String [] args) {
      float x = 1e30f;
      float y;
      y=x*x;
      System.out.println ( x + " a pour carre : " + y);
      float zero=0.f;
      float z=y/zero;
      System.out.println ( y + " divise par 0 = " + z);
      y=15;
      z=y/zero;
      System.out.println ( y + " divise par 0 = " + z);
      float x1 = Float.POSITIVE_INFINITY;
      float x2 = Float.NEGATIVE_INFINITY;
      z=x1/x2;
      System.out.println (x1 + "/" + x2 + " = " + z);
   }
}
```

6.3. Conversion implicite

6.3.1. Expression mixte

Les opérateurs ne fonctionnent qu'avec des opérandes de même type. Lorsque que l'on écrit : n * x + p avec n et p des entiers et x un flottant, le compilateur connaissant les règles de priorités, convertira la valeur de n en flottant (cela est possible car on approchera de la vraie valeur de n, par contre l'inverse est impossible et il y aurait dès lors erreur de compilation) et le résultat sera donc un flottant. Il agira de la même manière avec p pour donner donc un résultat final en flottant.
Ce type de conversion s'appelle ajustement de type : int → long → float → double.
Par contre, il existe aussi des promotions numériques.
Ce cas arrive lorsque l'on veut effectuer une opération avec des types *byte*, *char* ou *short*.
Dans ce cas, le résultat sera de type *int*.
Exemple :
p1 * p2 + p3 * x avec p1, p2, p3 en *short* et x en *float*.
En premier lieu, p1 et p2 seront convertis par promotion numérique en *int* pour donner un résultat en *int* pour la multiplication.
Ensuite p3 * x, il y a deux choses qui se passent : p3 converti par promotion numérique en *int* et ensuite par ajustement de type en *float* pour effectuer la multiplication qui donnera un *float*.
En dernier lieu, le résultat de p1 * p2 qui est en *int* devra être ajouté au résultat de p3 * x qui est en *float*, le premier sera converti par ajustement en *float* pour donner comme résultat final un *float*.

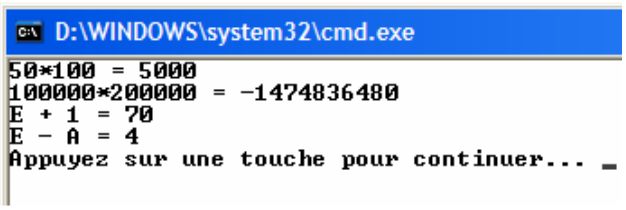
Mise en garde par cet exemple.

```
public class Conver {
   public static void main (String [] args) {
      byte b1 = 50, b2 = 100;
      int n;
      n = b1 * b2;
      System.out.println (b1 + "*" + b2 + " = "+ n);
      int n1 =100000, n2 = 200000;
      long p;
      p = n1 * n2;
      System.out.println (n1 + "*" + n2 + " = "+ p);
   }
}
```

6.3.2. Le cas du type *char*

En Java, une variable de type *char* peut intervenir dans une expression arithmétique car il existe une promotion numérique de *char* vers *int*.
En Java, un caractère est codé à l'aide de 16 bits qui pourra toujours correspondre à un entier.
Par exemple, le caractère E est représenté par 00000000 01000101 qui correspond à 69.
Soit c1 de type *char* et vaut ´E´, on effectue c1 + 1, le résultat de cette opération sera 70.
Si on a c1 et c2 de type *char* qui valent respectivement ´E´et ´A´, le résultat de l'opération c1 – c2 sera 4.

```
public class Conver {
   public static void main (String [] args) {
      byte b1 = 50, b2 = 100;
      int n;
      n = b1 * b2;
      System.out.println (b1 + "*" + b2 + " = "+ n);
      int n1 =100000, n2 = 200000;
      long p;
      p = n1 * n2;
      System.out.println (n1 + "*" + n2 + " = "+ p);
      char c1 ='E', c2='A';
      System.out.println (c1 + " + 1 = " + (c1 + 1));
      System.out.println (c1 + " – " + c2 + " = " + (c1 – c2));
   }
}
```



6.4. Transtypage

Si l'on désire qu'un résultat soit d'un type spécifique, il existe un procédé pour le renvoyer dans un autre type, il suffit de placer devant la variable *(type) nom de variable*.
Cette technique est dangereuse puisqu'il faut que vous soyez certain de ce que vous faites, car il peut y avoir des erreurs.

Exemple :

```
float x = 1.0 ;
int y = (int) x ;
```

Cela suppose que la valeur de x tiendra dans *int*. Il faut aussi savoir que les décimales de x seront perdues lors de la conversion. Java arrondit au nombre entier le plus proche.

6.5. Les opérateurs relationnels

Ce type d'opérateurs permet de comparer des expressions. Le résultat est une variable booléenne (*true* ou *false*).
Les opérateurs relationnels par niveau de priorités :

- ✓ <, <=, >, >=, respectivement « *inférieur à* », « *inférieur ou égal à* », « *supérieur* » et « *supérieur ou égal à* »
- ✓ = =, !=, respectivement « *égal à* » et « *différent de* ».

Dans le cas de contrôle avec des variables *char*, la comparaison se fait en fonction de leur UNICODE.
Il est à savoir que = = et != peuvent être aussi utilisées dans le cas d'objets.

6.6. Les opérateurs logiques

Java dispose de 6 opérateurs logiques qui sont prioritaires les uns sur les autres de la manière suivante :

! : négation
& : et (prend la valeur *true* si les deux expressions sont vraies)
^ : ou exclusif (ne prend la valeur *true* que si une et une seule condition est vraie)
| : ou inclusif (prend la valeur *true* si au moins une des deux conditions donne vrai)
&& : et (avec court circuit) (idem & mais la seconde condition sera testée si la première est vraie. En effet, si la première donne faux, alors peu importe le résultat de la seconde, le tout sera faux.)
|| : ou inclusif (avec court circuit) (idem | mais la seconde condition sera testée si la première est fausse. En effet, si la première donne vraie, alors peu importe le résultat de la seconde, le tout sera vrai.)

6.7. Les opérateurs d'incrémentation et de décrémentation

Leur rôle est d'alléger les écritures des instructions du type i = i + 1 ou i = i – 1.
Ces opérateurs permettent aussi d'éviter des erreurs de compilation lorsque l'on utilise des types *byte* ou *char*. En effet, si b est de type *byte*, alors b = b + 1 ; sera une erreur de compilation.
L'incrémentation se fera comme suit ++i et la décrémentation – –i.
Cela correspond au cas habituel d'incrémentation et décrémentation.
Il est possible d'incrémenter et décrémenter une variable mais auparavant attribuer sa valeur à une autre variable.

```
i = 3 ;
n = ++i – 5 ;
```
Après exécution, n = –1 et i = 4
Dans ce cas, on parle de préincrémentation

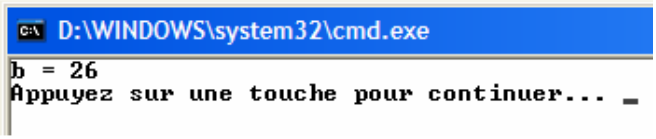
```
i = 3 ;
n = i++ – 5 ;
```
Après exécution, n = –2 et i = 4
Dans ce cas, on parle de postincrémentation

Il existe la même chose avec la décrémentation.

6.8. Les opérateurs d'affectation élargie

Comme les opérateurs d'incrémentation et de décrémentation, Java dispose d'opérateurs encore plus puissants qui permettent de remplacer i = i + k, i = i * n, …
D'une manière générale, Java permet de condenser les affectations de la forme :
variable = variable opérateur expression en *variable opérateur= expression*.
i = i + k devient i += k et i = i * n devient i *= k
Comme les opérateurs d'incrémentation et de décrémentation, les opérateurs d'affectations élargies permettent d'éviter des erreurs de compilation à cause de conversion. Attention tout de même au résultat.

```
public class ErrAffE1
{   public static void main (String [ ] args)
    {   byte b=10;
        int n = 10000;
        b += n;
        System.out.println ("b = " + b);
    }
}
```



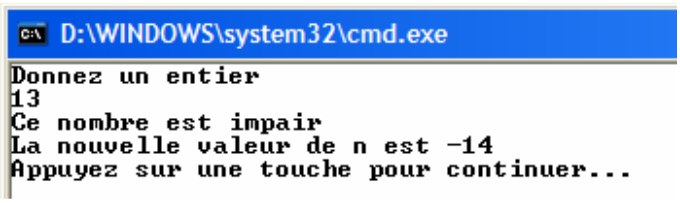
6.9. Les opérateurs de manipulation de bits

Java dispose d'opérateurs permettant de travailler directement sur le motif binaire d'une valeur. Ceux-ci procurent ainsi des possibilités réservées à la programmation assembleur.

Opérande 1	0	0	1	1
Opérande 2	0	1	0	1
&	0	0	0	1
|	0	1	1	1
^	0	1	1	0

Grâce aux conversions implicites (ajustement de type et promotions numériques), on pourra les utiliser dans les types suivants : *byte, short, char, int, long*.
Voici les opérateurs :
Les opérateurs bit à bit : & : et
| : ou inclusif
^ : ou exclusif
~ : complément à un
Les opérateurs de décalage : << : décalage à gauche
>> : décalage arithmétique à droite
>>> : décalage logique à droite

```
public class Manip1 {
   public static void main (String [] args) {
      int n;
      System.out.println ("Donnez un entier");
      n = Clavier.lireInt();
      if(((n & 1) == 1) )
         System.out.println ("Ce nombre est impair");
      else
         System.out.println ("Ce nombre est pair");
      n = ~n;
      System.out.println ("La nouvelle valeur de n est " + n);
   }
}
```



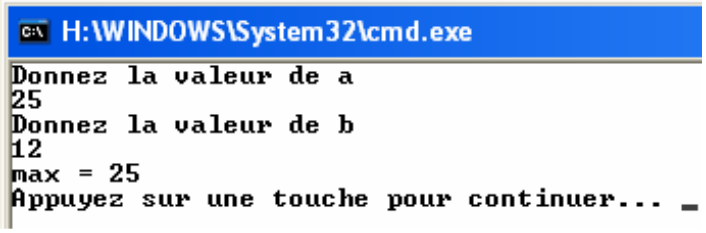
6.10. L'opérateur conditionnel

Considérons l'instruction suivante :

```
if ( a > b)
   max = a ;
else
   max = b ;
```

Cela permet d'attribuer à la variable max le nombre qui est le plus grand entre a et b.
Cette instruction peut être traduite en français de manière suivante : si a > b alors a sinon b.
En Java, il est possible grâce à l'opérateur conditionnel de traduire presque littéralement comme suit : `max = a > b ? a : b ;`
Il faudra que cette instruction soit affectée à une variable.

```
public class Condition {
   public static void main (String [] args) {
   int a,b,max;
   System.out.println("Donnez la valeur de a");
   a = Clavier.lireInt();
   System.out.println("Donnez la valeur de b");
   b = Clavier.lireInt();
   max = (a > b ? a : b);
   System.out.println ("max = " + max);
   }
}
```



6.11. Exercices

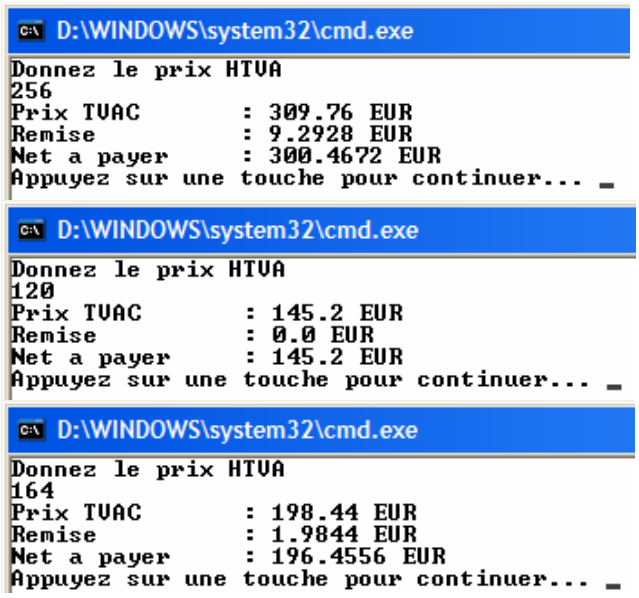
a) Ecrire un programme qui permette de trouver l'image d'une fonction du type $ax^2 + bx + c$ dont les valeurs a, b et c sont introduites par l'utilisateur, le tout en entier.
b) Le même programme mais cette fois en réel.
c) Ecrire un programme qui permette de savoir si deux nombres sont multiples. L'utilisateur introduit le plus grand en premier.

7. Les instructions de contrôle de Java

7.1. L'instruction *if*

Cette instruction est l'instruction classique du test. La manière de l'écrire est la suivante : *if (condition) { … } else { … }* . Cette dernière partie n'est pas obligatoire.
Il est clair que l'on peut imbriqué autant d'instructions *if* que l'on veut. Il est à noter que *else* se rapportera toujours au dernier *if*.
Exemple :
Programme de facturation avec remise en fonction du montant TVAC.
Voici le pourcentage de remise :
0% pour un montant inférieur à 150 €
1% pour un montant entre 150 € et 250 €
3% pour un montant entre 250 € et 750 €
5% pour un montant supérieur à 750 €.

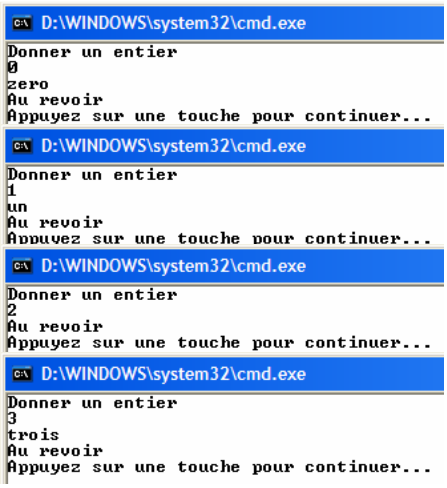
```
public class Tva {
   public static void main (String [] args){
      double tauxTva = 21.;
      double ht, ttc, net, tauxRemise, remise;
      System.out.println ("Donnez le prix HTVA");
      ht = Clavier.lireDouble();
      ttc = ht * (1. + tauxTva/100.);
      if (ttc < 150.) tauxRemise = 0.;
         else if (ttc < 250.) tauxRemise = 1.;
            else if (ttc < 750.) tauxRemise = 3.;
               else tauxRemise = 5.;
      remise = ttc*tauxRemise/100.;
      net = ttc - remise;
      System.out.println ("Prix TVAC" + "\t" + ": " + ttc + "EUR");
      System.out.println ("Remise " +"\t\t"+ ": " + remise + "EUR");
      System.out.println ("Net a payer " + "\t" + ": " + net + "EUR");
   }
}
```



7.2. L'instruction *switch*

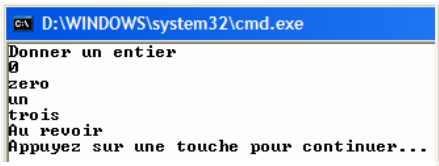
Voici trois exemples pour mieux maîtriser cette instruction qui est l'équivalent du *case of* en Pascal.

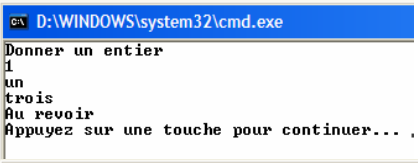
```
public class Switch1 {
   public static void main (String [] args){
      int n;
      System.out.println ("Donner un entier");
      n = Clavier.lireInt();
      switch (n) {
         case 0 :   System.out.println ("zero"); break;
         case 1 :   System.out.println ("un"); break;
         case 3 :   System.out.println ("trois"); break;
      }
      System.out.println ("Au revoir");
   }
}
```

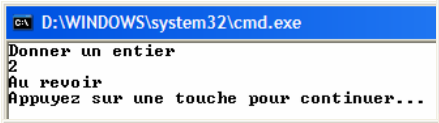


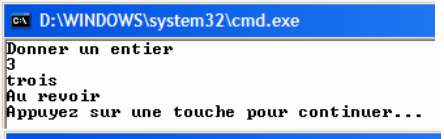

```
public class Switch2 {
   public static void main (String [] args){
      int n;
      System.out.println ("Donner un entier");
      n = Clavier.lireInt();
      switch (n) {
         case 0 :   System.out.println ("zero");
```

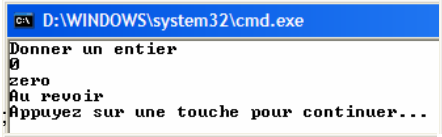
```
            case 1 :   System.out.println ("un");
            case 3 :   System.out.println ("trois");
        }
        System.out.println ("Au revoir");
    }
}
public class Switch3 {
    public static void main (String [] args){
        int n;
        System.out.println ("Donner un entier");
        n = Clavier.lireInt();
        switch (n) {
            case 0 :   System.out.println ("zero");
                       break;
            case 1 :   System.out.println ("un");
                       break;
            default :  System.out.println ("plus grand");
        }
        System.out.println ("Au revoir");
    }
}
```

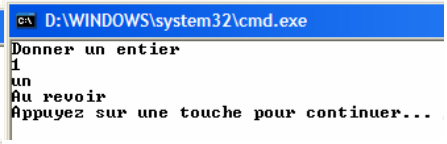


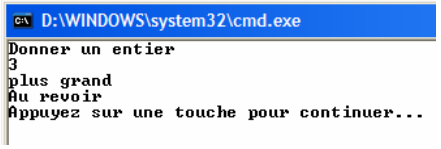










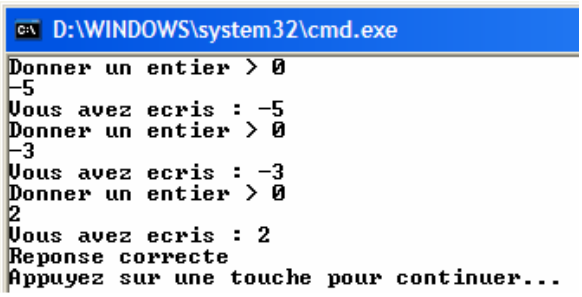


A l'aide de ces trois exemples, on s'aperçoit que si l'instruction *break ;* n'est pas présente, la suite des instructions où on se trouve sera exécutée. Il est donc assez important d'utiliser l'instruction *break ;* avec *switch*.
Une étiquette particulière qui permet d'exécuter quelque chose dans le cas où aucune valeur rencontrée précédemment n'a satisfait. Cette étiquette est *default*.
Il est à noter que cette instruction *switch* ne peut fonctionner qu'avec des types entiers ou caractères mais ne fonctionne pas avec un type flottant (*float* ou *double*).

7.3. L'instruction *do … while*

Cette instruction se traduit *faire … jusqu'à ce que (condition)*. Il est différent du *repeat* en Pascal, car la condition est ici contraposée. Il y aura donc au moins un passage dans la boucle.

```
public class Do1 {
    public static void main (String [] args){
        int n;
        do {
            System.out.println ("Donner un entier > 0");
            n = Clavier.lireInt();
            System.out.println ("Vous avez ecris : " + n);
        }
        while (n <=0);
        System.out.println ("Reponse correcte");
    }
}
```



On peut écrire des choses plus rapides s'il n'y a qu'une instruction comme dans le cas suivant : `do c = Clavier.lireString () ; while (c !='x') ;`

7.4. Exercice

Ecrire un programme qui permette en fonction du choix de l'utilisateur d'effectuer une opération. (1 = somme, 2 = différence, 3 = produit, 4 = division, 5 = racine carrée, 0 = fin)

7.5. L'instruction *while*

Cette instruction est identique à celle vue en Pascal, elle permet de faire une série d'instructions jusqu'à ce que la condition soit vérifiée.

```
public class Do2 {
   public static void main (String [] args){
      int n, som = 0;
      while (som < 100) {
         System.out.println ("Donner un entier");
         n = Clavier.lireInt();
         som += n;
      }
      System.out.println ("La somme obtenue :" + som);
   }
}
```

```
D:\WINDOWS\system32\cmd.exe
Donner un entier
15
Donner un entier
23
Donner un entier
14
Donner un entier
27
Donner un entier
35
La somme obtenue :114
Appuyez sur une touche pour continuer...
```

Avec ce type de boucle, il se peut que l'on ne fasse aucun passage.

7.6. L'instruction *for*

Ce type d'instruction ne doit être utilisé que lorsque l'on connaît exactement le nombre de passage à effectuer.
En Java, ce type de boucle est fort puissant car en une seule ligne on peut incrémenter ou décrémenter deux variables ou plus à la fois.
Supposons que l'on veuille additionner les 100 premiers nombres entiers.
Voici une écriture courte mais efficace :

```
public class Do3 {
   public static void main (String [] args){
      int i , j , som = 0;
      for ( i = 1 , j = 100 ; i <= 50 ; i++, j—) {
         som += (i + j);
      }
      System.out.println ("La somme des 100 premiers nombres entiers vaut " + som);
   }
}
```

```
D:\WINDOWS\system32\cmd.exe
La somme des 50 premiers nombres entiers vaut 5050
Appuyez sur une touche pour continuer...
```

On peut utiliser l'instruction *break ;* pour sortir d'une boucle *for* à tout moment.
Il est évident que l'on pourrait alors choisir une boucle de type *while*.
Il existe aussi une instruction *continue ;* qui elle permet de passer un tour dans la boucle *for*.

```
public class Do4 {
   public static void main (String [] args){
      int i;
      for ( i = 1 ; i <= 5 ; i++) {
         System.out.println ("passage " + i);
         if(i<4) continue;
         System.out.println ("passage complet pour i = " + i);
      }
      System.out.println ("Fin de boucle");
   }
}
```

```
D:\WINDOWS\system32\cmd.exe
passage 1
passage 2
passage 3
passage 4
passage complet pour i = 4
passage 5
passage complet pour i = 5
Fin de boucle
Appuyez sur une touche pour continuer...
```

7.7. Exercices

a) Ecrire un programme qui permette de trouver la factorielle d'un nombre entier.
n ! = 1 . 2 . 3 . 4 . … . (n – 1) . n. Exemples : 3 ! = 1 . 2 . 3 = 6 et 5 ! = 1 . 2 . 3 . 4 . 5 = 120

b) Ecrire un programme qui calcule la somme de n premiers termes de la « série harmonique » $1+\frac{1}{2}+\frac{1}{3}+...+\frac{1}{n}$. La valeur de n étant introduite par l'utilisateur.

8. Les tableaux

8.1. Déclaration et création de tableaux

int t [ ] ; cette déclaration précise que *t* est destiné à contenir la référence à un tableau d'entiers. Aucune dimension ne figure lors de cette déclaration. Cette déclaration est proche de celle de la référence à un objet. On crée un tableau comme un crée un objet à l'aide de l'instruction *new*. On doit préciser outre le type d'éléments qui sera utilisé, la dimension.
t = new int [5] ; cela veut dire que *t* fait référence à un tableau de 5 entiers. Un tableau, en Java, débute toujours en position 0.
Par défaut chacune des valeurs de ce tableau sera 0.
On peut lors de la création du tableau lui affecter des valeurs. Cela est possible à l'aide des accolades. Ce type de notation, ne pourra être utilisé que lors de la création.
*int t [ ] = {1, n , n + p, 2*p, 12} ;* permet de créer un tableau de 5 entiers dont les valeurs seront 1,n , n + p , 2*p et 12.
On peut aussi comme en Pascal, écrire : *t = new int[5] ; t[0] = 1 ; t[1] = n ; t[2]= n + p ; t[3] = 2*p ; t[4] = 12 ;*

8.2. Utilisation

8.2.1. Accès individuel aux éléments d'un tableau

Chacun des éléments d'un tableau aura un type, on pourra effectuer toutes les opérations qui sont permises par ce type comme une variable habituelle.
Si, lors de l'exécution, la valeur d'un indice est négative ou plus grande que la dimension prévue, on obtient une erreur d'exécution qui déclenche une exception de type *ArrayIndexOutOfBoundsException*. Le programme s'arrêtera sauf si on gère cette exception.
Voici un exemple pour calculer la moyenne d'un nombre d'élèves qui sera lu au clavier :

```
public class Moyenne {
   public static void main (String [] args){
      int i, nbEleves, nbElevesSuperieurMoyenne = 0;
      double somme = 0. , moyenne;
      System.out.println ("Combien d'eleves ?");
      nbEleves = Clavier.lireInt();
      double notes[] = new double[nbEleves];
      for (i=0 ; i < nbEleves ; i ++) {
         System.out.print ("Donner la note de l'eleve numero " + (i+1));
         notes[i] = Clavier.lireDouble();
         somme += notes[i];
      }
      moyenne = somme / nbEleves;
      System.out.println ("\tLa moyenne de la classe est " + moyenne);
      for (i = 0 ; i < nbEleves ; i++) if (notes[i]>moyenne) nbElevesSuperieurMoyenne++;
      System.out.println (nbElevesSuperieurMoyenne + " eleves ont plus que la moyenne");
   }
}
```

```
D:\WINDOWS\system32\cmd.exe
Combien d'eleves ?
6
Donner la note de l'eleve numero 1 : 14.5
Donner la note de l'eleve numero 2 : 12
Donner la note de l'eleve numero 3 : 17
Donner la note de l'eleve numero 4 : 15.5
Donner la note de l'eleve numero 5 : 8
Donner la note de l'eleve numero 6 : 9
        La moyenne de la classe est 12.666666
3 eleves ont plus que la moyenne
Appuyez sur une touche pour continuer...
```

8.2.2. Affectation de tableaux

Il est possible, en Java, d'affecter le contenu d'un tableau à un autre directement à l'aide de l'opérateur d'affectation =. Attention au type des tableaux et donc leur conversion.

```
public class Tableau {
   public static void main (String [] args){
      int [ ] t1 = new int [3];
      int [ ] t2 = new int [2];
      for (int i =0 ; i < 3 ; i++) t1[i] = i ;
      for (int i =0 ; i < 2 ; i++) t2[i] = 10 + i ;
```

```
D:\WINDOWS\system32\cmd.exe
t2[1] =5
Appuyez sur une touche pour continuer...
```

```
        t1 = t2;
        t1[1] = 5 ;
        System.out.println (″t2[1] =″ + t2[1]) ;
    }
}
```

Cela fonctionne comme les pointeurs en Pascal.

8.2.3. La taille d'un tableau

Il existe en Java un champ défini par défaut avec les tableaux qui est *length*. Ce champ permet de connaître le nombre d'éléments d'un tableau de référence donnée.

```
int t [ ] = new int[5] ;
System.out.println (″taille de t : ″ + t.length) ;
t [ ] = new int [3] ;
System.out.println (″taille de t : ″ + t.length) ;
```

Dans le premier cas, la valeur sera 5, dans le second cela sera 3.

8.3. **Tableaux à plusieurs indices**

De nombreux langages disposent de la notion de tableau à plusieurs indices. Par exemple, un tableau à deux indices permet de représenter une matrice.
Java ne dispose pas d'une telle notion. Mais il est possible de la simuler en créant des tableaux de tableaux, c'est-à-dire en créant des tableaux qui seront eux même composés de tableaux. Mais cette possibilité est plus riche que dans les autres langages, du fait qu'il sera possible de créer des tableaux irréguliers, pour chaque ligne, le nombre de colonnes peut différer.

8.3.1. Déclaration

Pour déclarer voici trois possibilités :

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

Ainsi `int t [ ] [ ] = { new int [3], new int [2] } ;` correspond au fait que l'on crée un tableau à deux lignes, la première sera composée de trois colonnes et la seconde de deux colonnes.
`t[0]` correspondra à la première ligne entière.
`t[0] [1]` correspondra à l'élément de la première ligne et seconde colonne.
`t[1] [2]` correspondra à l'élément de la seconde ligne et troisième colonne.
`t.length` vaudra 2, car le tableau t est composé de deux tableaux.
`t[0].length` vaudra 3, car la première ligne est un tableau de 3 éléments.
`t[1].length` vaudra 2, car la première ligne est un tableau de 2 éléments.

8.3.2. Initialisation

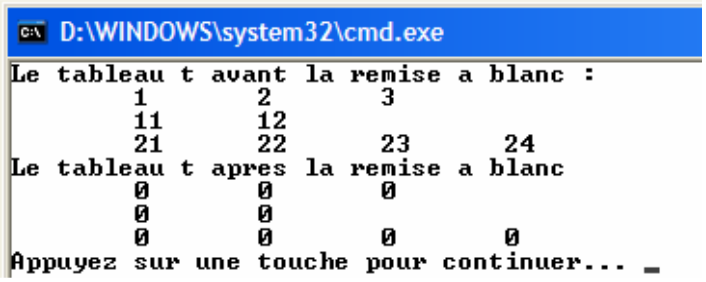
Il est possible comme pour un tableau à un indice d'initialiser le tableau à l'aide d'accolades imbriquées.

```
int t [ ] [ ] = { {1, 2, 3} , {11, 12} } ;
```

Cette écriture a permis de construire un tableau qui possède deux lignes dont la première possède trois colonnes et la seconde deux.

Dans l'exemple qui suit, on crée deux méthodes *static* (propre à la classe) définies dans une classe *Util* qui permet d'afficher les valeurs d'un tableau entier à deux indices, ligne par ligne et de mettre à zéro les éléments du tableau d'entier à deux indices.

```
public class Tableau2 {
   public static void main (String [] args){
      int t [] [] = {{1,2,3},{11,12},{21,22,23,24}};
      System.out.println ("Le tableau t avant la remise a blanc :");
      Util.affiche(t);
      Util.raz(t);
      System.out.println ("Le tableau t apres la remise a blanc");
      Util.affiche(t);
   }
}
class Util {
   static void raz (int t [] []) {
      for (int i = 0 ; i < t.length ; i++)
         for (int j = 0 ; j < t[i].length ; j++)
            t[i] [j] = 0;
   }
   static void affiche (int t [] []) {
      for (int i = 0 ; i < t.length ; i++) {
         for (int j = 0 ; j < t[i].length ; j++)
            System.out.print ("\t" + t[i] [j]);
         System.out.println();
      }
   }
}
```



On peut comme avec Pascal, ne pas utiliser le champ *length* mais il faut alors connaître la dimension du tableau. Le gros avantage avec ce champ, c'est que le tableau peut varier en fonction de l'utilisateur.

8.4. **Exercice**

a) Ecrire un programme qui crée un tableau comportant les valeurs des carrés des n premiers nombres impairs, la valeur de n étant introduite par l'utilisateur et qui affiche les valeurs sous la forme « 9 a pour carre 81 ».

b) Ecrire un programme qui donne les nombres premiers entre 1 et 1000 (cette dernière valeur peut-être introduite par l'utilisateur) et affichée par un tableau comme suit :

1	2	3	5	7	11	13	17	19	23
29	31	37	41	43	47	53	59	61	67
71	73	79	83	89	97	101	103	107	109
113	127	131	137	139	149	…			

Pour cet exercice, vous pouvez aller chercher une aide quant à savoir trouver si un nombre est premier ou pas. (Crible d'Eratosthène)

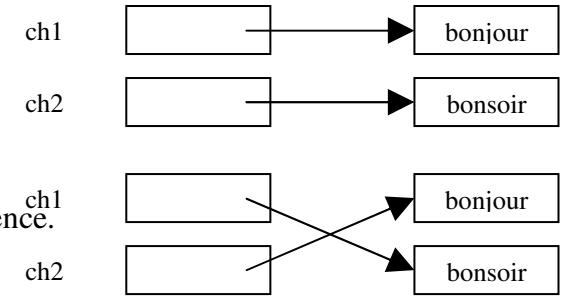
9. Les chaînes de caractères

Les chaînes de caractères, en Java, ne sont pas des types primitifs. Deux classes existent, *String* et *StringBuffer* pour combler le vide.

9.1. La classe *String*

Du fait qu'une variable est de type *String*, l'objet créé pointe vers une référence.
Ainsi, dès que l'objet est créé, on ne peut plus le modifier. Tout ce qui est permis est de changer sa référence et ainsi donc d'en « changer » sa valeur. Attention, cette manière de faire, vous oblige à créer un autre objet.

```
String ch1, ch2, ch;
ch1 = "bonjour" ;
ch2 = "bonsoir" ;
ch = ch1 ;
ch1 = ch2 ;
ch2 = ch ;
```



Les deux objets n'ont pas changé de valeur mais de référence.

9.1.1. Construction

Pour construire un objet de ce type, la commande est similaire à tous les objets `new` .
Il existe cinq constructeurs dont en voici trois :

`String ch = new String() ;` constructeur par défaut qui crée une chaîne vide.

`String ch = new String("bonjour") ;` (qui peut être remplacé par `String ch = "bonjour" ;`)

`String ch = new String({'b','o','n','j','o','u','r'})` constructeur qui donne à ch la valeur bonjour mais à partir d'un tableau de caractères.

9.1.2. Méthodes

Plusieurs méthodes existent mais en voici quelques unes fréquemment utilisées.

a) *length*
Cette méthode permet de connaître la longueur de la chaîne de caractères.

```
String ch = "bonjour" ;
int n = ch.length() ;
```

La valeur de n est 7.

b) *charAt*
Cette méthode permet d'accéder à un caractère d'un rang donné (en Java, la numérotation commence à 0).

```
String ch = "bonjour" ;
```

`ch.charAt(0) ;` correspond au caractère b
`ch.charAt(2);` correspond au caractère n

c) *indexOf*
Cette méthode donne la première position du caractère ou de la chaîne spécifié

```
String ch = "bonjour" ;
```

`ch.indexOf('o') ;` correspond à la valeur 1
`ch.indexOf("jo") ;` correspond à la valeur 3
Il existe une méthode identique mais qui donne la dernière position du caractère ou de la chaîne spécifié : *lastIndexOf*.
Remarque : si la valeur spécifiée n'est pas trouvée, la valeur retournée est –1.

d) *equals*

Cette méthode n'est pas propre à la classe *String* mais elle est héritée de la classe *Object*. Elle permet de savoir si le contenu de la chaîne de caractère est identique à une autre.

```
String ch1 = "bonjour" ;
String ch2 = "bonsoir" ;
ch1.equals(ch2); Le résultat est un booléen FAUX
ch1.equals("bonjour"); Le résultat est un booléen VRAI
```

Une méthode presque similaire existe mais qui ne tient pas compte de la casse, c'est la méthode *equalsIgnoreCase*. Cette méthode est propre à la classe *String*.

e) *compareTo*

Cette méthode permet de comparer (dans le sens plus grand, plus petit) deux chaînes de caractères.

```
String ch1 = "bonjour" ;
String ch2 = "bonsoir" ;
ch1.compareTo(ch2); Le résultat est un entier négatif (le mot bonjour se trouve avant bonsoir)
ch2.compareTo(ch1); Le résultat est un entier positif (le mot bonsoir se trouve après bonjour)
```

f) *concat*

Cette méthode permet de concaténer une chaîne à une autre mais dans un nouvel objet. En effet, un objet de type *String* n'est pas modifiable.

Il existe un moyen plus rapide d'effectuer la concaténation, c'est d'utiliser le symbole +. Ce symbole permet aussi d'effectuer des conversions de type.

```
int n = 26;
String titre = new String("resultat : ") ;
String monnaie = "$" ;
String resul = titre + n + " " + monnaie;
System.out.println(resul); Le résultat obtenu est : resultat : 26 $.
```

Remarque : Avec la méthode *concat*, il faut passer par plusieurs intermédiaires pour parvenir au résultat final.

concat ne peut se faire d'une variable avec une autre. Dans notre cas précédent, il faut déjà utiliser trois variables intermédiaires ou allonger l'écriture.

De plus, pour concaténer la valeur de n, il faut utiliser une méthode qui convertisse l'entier en chaîne de caractère.

Il est aussi possible d'utiliser l'opérande += pour concaténer.

g) *replace*

Cette méthode permet de copier une chaîne de caractère en une autre en remplaçant toutes les occurrences d'un caractère donné par un autre.

```
String ch1 = "bonjour" ;
String ch2 = ch1.replace('o','a') ; Le résultat est banjaur
```

h) *substring*

Cette méthode permet de créer une nouvelle chaîne de caractères en extrayant de la chaîne courante, soit tous les caractères depuis une position donnée, soit tous les caractères entre deux positions données (la première incluse, la seconde exclue).

```
String ch1 = "bonjour" ;
String ch2 = ch1.substring(3) ; Le résultat est jour
String ch3 = ch1.substring(0, 3) ; Le résultat est bon
```

i) *toLowerCase / toUpperCase*
Ces méthodes permettent de créer une nouvelle chaîne de caractères en remplaçants toutes les majuscules par des minuscules ou toutes les minuscules par des majuscules.
```
String ch1 = "LanGaGe 3" ;
String ch2 = ch1.toLowerCase() ; Le résultat est langage 3
String ch3 = ch1.toUpperCase() ; Le résultat est LANGAGE 3
```

j) *valueOf*
Cette méthode permet de changer n'importe quelle objet en chaîne de caractère.
```
int n = 427 ;
String ch = String.valueOf(n) ; Le résultat est 427 comme chaîne de caractères.
```
Il est possible de faire cela avec des réels, booléens, …
Pour effectuer la conversion inverse, il faut utiliser des méthodes propres aux classes généralisées des types : *Byte.parseByte*, *Short.parseShort*, *Integer.parseInt*, *Long.parseLong*, *Float.parseFloat* et *Double.parseDouble*.
Attention aux erreurs possibles. Ces erreurs, il faut les gérer pour que le programme puisse continuer.

9.2. La classe *StringBuffer*

L'avantage de cette classe par rapport à la classe *String* est que les objets de type *StringBuffer* peuvent être modifiés. Malheureusement cette classe n'a aucun lien d'héritage de la classe *String*.
La seule chose qui relie les deux est que l'on peut construire un objet de type *StringBuffer* à l'aide d'un objet de type *String*.
```
StringBuffer ch1 = new StringBuffer("Bonjour") ;
```

Il est possible aussi uniquement de construire l'objet en lui définissant uniquement sa taille.
```
StringBuffer ch1 = new StringBuffer(5) ;
```

Le constructeur par défaut construit une chaîne vide de 16 caractères.
```
StringBuffer ch1 = new StringBuffer() ;
```

9.2.1. Les méthodes

a) *setCharAt*
Cette méthode permet la modification d'un caractère de rang donné.

b) *charAt*
Cette méthode permet l'accès à un caractère de rang donné.

c) *append*
Cette méthode permet d'ajouter au même objet une chaîne de caractères ou tout autre type primitif en fin de chaîne.

d) *insert*
Cette méthode permet d'insérer au même objet une chaîne à un emplacement donné.

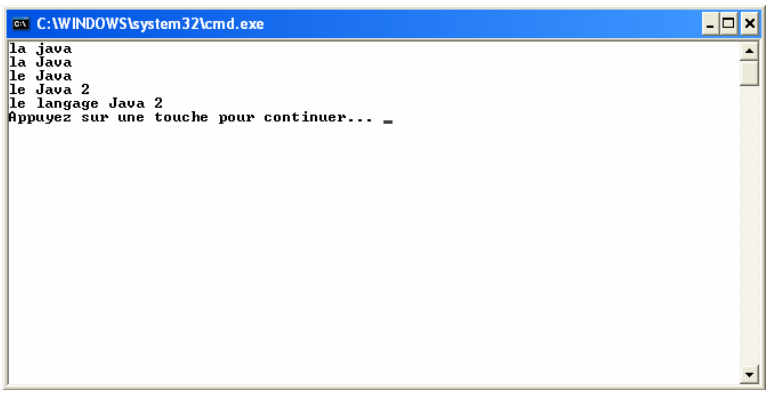
e) *replace*
Cette méthode permet de remplacer au même objet une partie de chaîne par une autre donnée.

f) *toString*
Cette méthode permet de convertir l'objet de type *StringBuffer* en *String*.

```
StringBuffer ch1 = new StringBuffer("Bonjour") ;
String ch2 = StringBuffer.toString(ch1);
```

9.2.2. Exemple

```
class TstStb {
   public static void main (String args[]) {
      String ch = "la java";
      StringBuffer chBuf = new StringBuffer(ch);
      System.out.println(chBuf);
      chBuf.setCharAt(3,'J');
      System.out.println(chBuf);
      chBuf.setCharAt(1,'e');
      System.out.println(chBuf);
      chBuf.append(" 2");
      System.out.println(chBuf);
      chBuf.insert(3,"langage ");
      System.out.println(chBuf);
   }
}
```



9.3. **Exercice**

a) Ecrire un programme qui conjugue un verbe régulier du premier groupe à l'indicatif présent et au futur simple.
b) Ecrire le programme du PENDU.
c) Ecrire un programme qui permette de trier un tableau de chaîne de caractère introduit par l'utilisateur.

Chapitre IV: Java : Gestion des exceptions

1. La notion d'exception

La gestion des exceptions offre des moyens structurés de capturer les erreurs d'exécution d'un programme et de fournir des informations significatives à leur sujet.
Il est possible de définir un gestionnaire d'exception pour qu'il effectue certaines actions avant de permettre au programme de s'arrêter.
La gestion des exceptions utilise les mots clés *try*, *catch* et *finally*. Une méthode peut déclarer une exception à l'aide des mots clés *throws* et *throw*.
En Java, les erreurs portent le nom d'exception mais toutes les exceptions ne sont pas des erreurs, elles peuvent signaler un événement particulièrement inhabituel qui se produit dans le programme et qui mérite une attention particulière.
Elle permet aussi d'alléger le programme lui-même car les types d'erreurs sont normalement exceptionnels.
La gestion des erreurs en Java, utilise beaucoup plus le processeur qu'à l'habitude, il est donc normal de ne pas surcharger ou de gérer trop d'exceptions.
Dans Java, une exception est un objet qui est créé lorsqu'une situation anormale se produit dans le programme. Cet objet possède des membres de données qui stockent des informations relatives à la nature du problème. Dès cet instant, l'exception est levée puisqu'une partie du programme la traitera. Le code du reçoit l'objet exception capture ou intercepte l'exception.
Quatre catégories de situations à l'origine d'exceptions sont répertoriées.

1°) Erreurs de code ou de données : essai de transtypage non valide, utilisation d'un index qui se trouve en dehors des limites du tableau ou évaluation d'une expression arithmétique entière avec un diviseur nul.
2°) Exception d'une méthode standard : par exemple utilisation de la méthode *substring()* dans la classe *String* peut lever une exception *StringIndexOutOfBoundsException*.
3°) Lever des exceptions personnelles : quelques exemples sont vus en fin de chapître.
4°) Erreurs Java : erreurs dues à la machine virtuelle. Elles surviennent généralement comme conséquence d'une erreur dans le programme.

2. Types d'exceptions

2.1. Exceptions RuntimeException

2.1.1. Noms des classes utilisées

a) *ArithmeticException*
Cela arrive pour une condition arithmétique non valide, par exemple une tentative de division d'une valeur entière par zéro.
b) *IndexOutOfBoundsException*
Cela arrive lorsque l'index se trouve en dehors des limites. Il peut s'agir d'un tableau, d'un *String*.
c) *NegativeArraySizeException*
Cela arrive lorsque l'on veut définir un tableau de dimension négative.
d) *NullPointerException*
Cela arrive lorsque l'on utilise une variable objet contenant la valeur *null*.
e) *ArrayStoreException*
Cela arrive lorsque l'on essaye de stocker un objet dans un tableau qui n'est pas autorisé pour le type du tableau.
f) ClassCastException
Cela arrive lorsque l'on veut transtyper un objet vers un autre type non valide.
g) *IllegalArgumentException*
Cela arrive lorsque l'on transmet vers une méthode un objet de type différent du type du paramètre.
h) *SecurityException*
Cela arrive lorsqu'une applet essaie de lire un fichier sur une machine locale.
i) *IllegalStateException*
Cela arrive lorsque l'on essaie d'appeler une méthode à un moment illégal

3. Traiter les exceptions

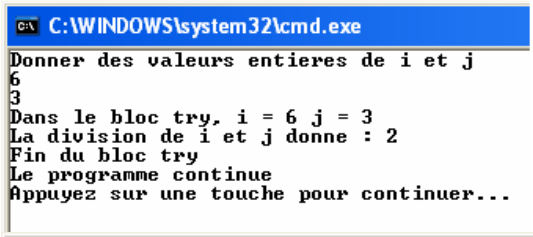
Pour gérer les exceptions à l'endroit où elles se produisent, on peut inclure dans le code trois types de bloc dans une méthode : *try*, *catch* et *finally*
Le bloc *try* contient du code qui peut produire une ou plusieurs exceptions.
Le bloc *catch* contient du code destiné à gérer des exceptions d'un type particulier qui peuvent être levées dans un bloc *try*.
Le bloc *finally* contient du code qui est toujours exécuté avant qu'une méthode se termine, que les exceptions soient levées ou non dans le bloc *try*.

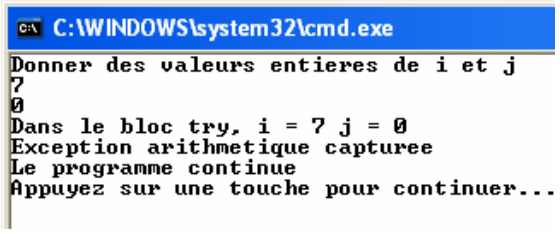
3.1. Exemples

3.1.1. Exemple simple

Cet exemple permet de voir que lorsqu'il y a une exception de type *ArithmeticException* certaines actions peuvent être effectuées dans le cas sans que le programme s'arrête pour autant.

```
public class TestTryCatch {
   public static void main (String args []) {
      System.out.println("Donner des valeurs entieres de i et j");
      int i = Clavier.lireInt(),j = Clavier.lireInt();
      try {
         System.out.println("Dans le bloc try, i = " + i + " j = " + j);
         System.out.println( La division de i et j donne :   + (i/j));
         System.out.println("Fin du bloc try");
      }
      catch(ArithmeticException e) {
         System.out.println("Exception arithmetique capturee");
      }
      System.out.println("Le programme continue");
   }
}
```

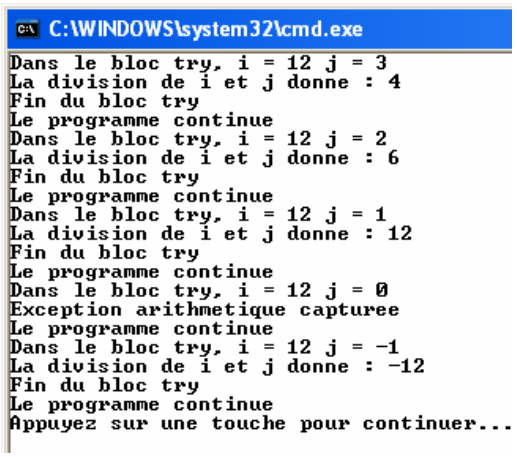




Attention à la portée des variables car un bloc *try* fonctionne comme tout autre bloc et les variables définies dans le bloc ne peuvent être utilisées à l'extérieur.
Les blocs *try* et *catch* doivent être consécutifs, il ne peut y avoir aucune instruction les séparant.

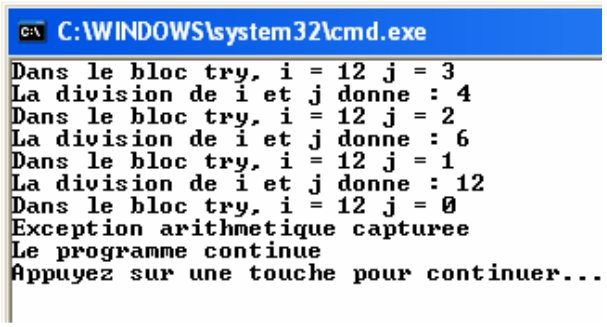
3.1.2. Exemple dans une boucle

```
public class TestBoucleTryCatch {
   public static void main (String args []) {
      int i = 12;
      for (int j = 3 ; j >=-1 ; j--) {
         try {
            System.out.println("Dans le bloc try, i = " + i + " j = " + j);
            System.out.println("La division de i et j donne : "  + (i/j));
            System.out.println("Fin du bloc try");
         }
         catch(ArithmeticException e) {
            System.out.println("Exception arithmetique capturee");
         }
         System.out.println("Le programme continue");
      }
   }
}
```



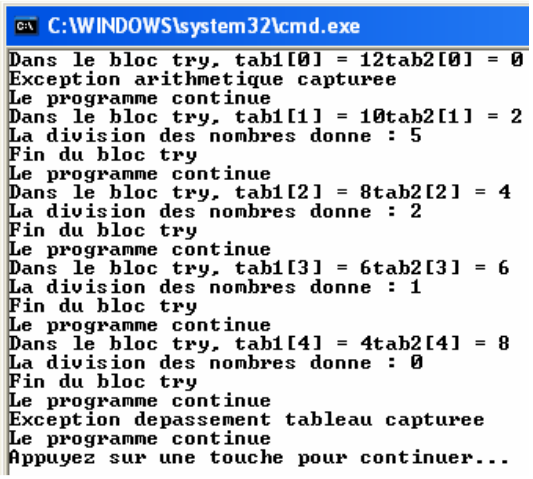
3.1.3. Exemple dans une boucle avec arrêt de la boucle

```
public class TestBoucleTryCatch2 {
   public static void main (String args []) {
      int i = 12;
      try {
         for (int j = 3 ; j >=-1 ; j--) {
            System.out.println("Dans le bloc try, i = " + i + " j = " + j);
            System.out.println("La division de i et j donne : "  + (i/j));
         }
         System.out.println("Fin du bloc try");
      }
      catch(ArithmeticException e) {
         System.out.println("Exception arithmetique capturee");
      }
      System.out.println("Le programme continue");
   }
}
```



3.1.4. Exemple avec plusieurs levées d'exceptions

```
public class TestTryCatchMultiple {
   public static void main (String args []) {
      int tab1 [] = {12,10,8,6,4};
      int tab2 [] = {0,2,4,6,8};
      for (int i = 0 ; i <= 5 ; i++) {
         try {
            System.out.println("Dans le bloc try, tab1[" + i + "] = " + tab1[i] +
            "tab2[" + i + "] = " + tab2[i]);
            System.out.println("La division des nombres donne : "  +
            (tab1[i]/tab2[i]));
            System.out.println("Fin du bloc try");
         }
         catch(ArithmeticException e) {
            System.out.println("Exception arithmetique capturee");
         }
         catch(IndexOutOfBoundsException e) {
            System.out.println("Exception depassement tableau capturee");
         }
         System.out.println("Le programme continue");
      }
   }
}
```



3.2. **Le bloc *finally***

La nature immédiate d'une exception levée signifie que le bloc *try* s'arrête, quelle que soit l'importance du code qui suit le point à partir duquel l'exception a été levée. Il se peut que si un fichier a été ouvert, il ne soit jamais fermé.
Le bloc *finally* fournit les moyens de tout nettoyer à la fin de l'exécution d'un bloc *try*. Il est possible d'utiliser ce bloc lorsque l'on doit être sûr qu'un code particulier est exécuté avant que la méthode ne renvoie une valeur.
Ce type de bloc est toujours exécuté, quoi qu'il arrive au cours de l'exécution de la méthode.

3.2.1. Exemple

Voici un exemple assez complet où des exceptions arrive dans la méthode *divise* mais aussi dans la méthode *main* avec plusieurs *catch* et un *finally*.
On a besoin d'une classe spéciale que l'on importe à l'aide de l'instruction *import*.

```
import java.io.IOException;
public class TestBlocTryCatchFinally {
   public static void main(String args []) {
      int [] x = {10, 5, 0};
      try {
         System.out.println("Premier bloc try de la methode main");
         System.out.println("Resultat = " + divise(x,0));
         x[1] = 0;
         System.out.println("Resultat = " + divise(x,0));
         x[1] = 1;
         System.out.println("Resultat = " + divise(x,1));
      }
      catch(ArithmeticException e) {
         System.out.println("Exception arithmetique capturee dans main");
      }
      catch(ArrayIndexOutOfBoundsException e) {
         System.out.println("Exception de depassement d'index tableau capturee dans main");
      }
      System.out.println("En dehors du premier bloc try de main");
      System.out.println("\nAppuyer sur Enter pour sortir");
      try {
         System.out.println("Second bloc try de la methode main");
         System.in.read();
      }
      catch(IOException e) {
         System.out.println("Exception d'entre/sortie dans main");
      }
      finally {
         System.out.println("Bloc finally du second bloc try de main");
      }
      System.out.println("Code execute apres les blocs try de main");
   }
   public static int divise(int[] tab, int index) {
      try {
         System.out.println("\nPremier bloc try de la methode divise");
         tab[index + 2] = tab[index]/tab[index + 1];
         System.out.println("Code de fin du premier bloc try de divise");
         return tab[index + 2];
      }
      catch(ArithmeticException e) {
         System.out.println("Exception aritmetique capturee dans divise");
      }
      catch(ArrayIndexOutOfBoundsException e){
         System.out.println("Exception de depassement d'index tableau capturee dans divise");
      }

      finally {
         System.out.println("Bloc finally de divise");
      }
      System.out.println("code execute apres le bloc try de divise");
      return tab[index + 2];
   }
}
```

```
C:\WINDOWS\system32\cmd.exe
Premier bloc try de la methode main

Premier bloc try de la methode divise
Code de fin du premier bloc try de divise
Bloc finally de divise
Resultat = 2

Premier bloc try de la methode divise
Exception aritmetique capturee dans divise
Bloc finally de divise
code execute apres le bloc try de divise
Resultat = 2

Premier bloc try de la methode divise
Exception de depassement d'index tableau capturee dans divise
Bloc finally de divise
code execute apres le bloc try de divise
Exception de depassement d'index tableau capturee dans main
En dehors du premier bloc try de main

Appuyer sur Enter pour sortir
Second bloc try de la methode main

Bloc finally du second bloc try de main
Code execute apres les blocs try de main
Appuyez sur une touche pour continuer...
```

4. Exercice

Créer une calculette qui gère les erreurs d'encodage, les divisions interdites et les racines carrées interdites.

Ce chapitre va nous permettre de mieux manipuler et utiliser des classes et des objets graphiques prédéfinis. Il est clair que tous ces classes et objets peuvent être construits par vous-même. Cette possibilité sera étudiée plus finement avec un exemple lors du chapitre suivant.
La différence fondamentale entre une fenêtre « console » et une fenêtre graphique réside sur le fait que la première utilise de la programmation séquentielle quant à la seconde c'est de la programmation événementielle. Nous commencerons par créer des fenêtres graphiques, ensuite on y ajoutera un bouton et finalement on complexifiera la fenêtre pour la rendre de plus en plus conviviale.

1. Première fenêtre

1.1. La classe JFrame

Pour créer une fenêtre graphique, on dispose, dans le paquetage *javax.swing*, d'une classe standard nommée *JFrame*, possédant un constructeur sans arguments.

```
JFrame fen = new JFrame() ;
```

Par cette instruction, on crée un objet de la classe *JFrame* qui représente une fenêtre graphique.
Avec cette seule instruction, rien n'apparaît lors de l'exécution du programme, il est nécessaire à l'aide de la méthode *setVisible(boolean x)* de la rendre visible.

```
fen.setVisible (true) ;
```

Par défaut, une fenêtre graphique est définie avec une taille nulle, il faut donc lui définir des dimensions, hauteur et largeur, pour qu'elle apparaisse correctement.

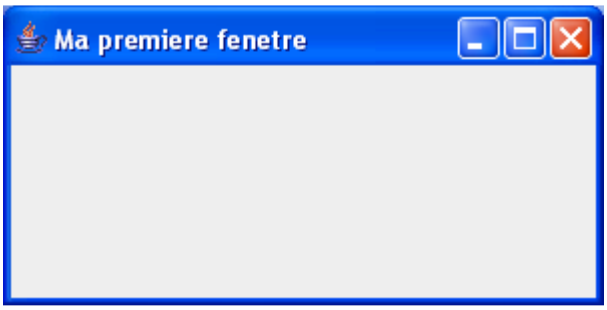
```
fen.setSize (300,150) ;
```

En général, on place un titre dans la barre de titre, on utilise à cet effet la méthode *setTitle(String x)* qui permet d'ajouter un titre à la barre de titre de la fenêtre graphique.

```
fen.setTitle (″Ma premiere fenetre″) ;
```

Le code source de ce programme qui fait apparaître une fenêtre graphique est :

```
import javax.swing.*;
public class PremFen0 {
   public static void main (String [] args) {
      JFrame fen = new JFrame();
      fen.setSize (300,150);
      fen.setTitle ("Ma premiere fenetre");
      fen.setVisible (true);
   }
}
```



Bien que rien n'ait été prévu de particulier, l'utilisateur peut manipuler cette fenêtre graphique comme n'importe quelle fenêtre graphique d'un logiciel :

- ✓ la redimensionner
- ✓ la déplacer
- ✓ la réduire à une icône

1.2. **Arrêt du programme**

La fermeture de la fenêtre ne ferme pas la console qui est ouverte en dessous, il faut la fermer elle aussi. C'est pourquoi, en utilisant la méthode *setDefaultCloseOperation(3);* la fenêtre active se ferme et la fenêtre console est prête à être fermée.

On pourrait faire apparaître dans la fenêtre console du texte qui suit son cours de manière séquentielle indépendamment de la fenêtre graphique. Nous verrons plus tard qu'en Java plusieurs processus peuvent fonctionner en même temps sans qu'ils interagissent entre eux. On appelle ces processus des *thread*.
Il est possible de créer une classe personnelle qui étend la classe *JFrame*. Nous verrons cela plus en profondeur lors du prochain chapitre dans lequel on travaillera la POO proprement dite.
Cette classe est une classe personnalisable. Voici le code source d'une fenêtre personnalisée qui donne le même résultat que précédemment.

```
import javax.swing.*;
class MaFenetre extends JFrame {
   public MaFenetre () {
      setTitle ("Ma seconde fenetre");
      setSize (300, 150);
   }
}
public class PremFen2 {
   public static void main (String [] args) {
      MaFenetre fen = new MaFenetre();
      fen.setVisible(true);
      fen.setDefaultCloseOperation(3);
   }
}
```

1.3. **Méthodes complémentaires**

Il est possible de positionner à l'écran, la fenêtre graphique que l'on a créée à l'aide de la méthode *setBounds (int x, int y, int haut, int larg).*

```
fen.setBounds (400,200,300,200) ;
```

Cette instruction permet de placer le coin supérieur gauche de la fenêtre graphique en coordonnée (400, 200) avec une taille 300 de hauteur et 200 de largeur.

La méthode *setBackground(Color bg)* permet de définir une couleur. Il est à remarquer que cette méthode a comme paramètre une variable de type *Color* qui est une classe d'objets qui attribue une couleur.
On peut choisir une couleur prédéfinie en écrivant comme paramètre *Color.green* ou pour une couleur spécifique, il faudra au préalable créer l'objet *bg* et lui définir une couleur qui sera composée de trois valeurs qui correspondent aux valeurs classiques des codes couleurs (0–255, 0–255, 0–255).
Il faut ajouter au début du code source *import java.awt.** qui permet d'utiliser la classe *Color*.

```
Color bg = new Color (15,200,124);
fen.setBackground (bg);
```

```
fen.setBackground (Color.green) ;
```

Cela permet d'obtenir une couleur de fond verte.
Voici un exemple qui permet de modifier la fenêtre graphique à l'aide de lecture au clavier de valeur introduite via la fenêtre console.

```
import javax.swing.*;
import java.awt.*;
public class PremFen1 {
    public static void main (String [] args) {
        JFrame fen = new JFrame();
        fen.setVisible(true);
        fen.setDefaultCloseOperation(3);
        while (true) {
            System.out.println("Donner les coordonnees du coin superieur gauche de la fenetre graphique");
            int x = Clavier.lireInt();
            int y = Clavier.lireInt();
            System.out.println("nouvelle largeur :");
            int larg = Clavier.lireInt ();
            System.out.println("nouvelle hauteur :");
            int haut = Clavier.lireInt ();
            System.out.println("donner les codes RVB pour definir la couleur de fond");
            int r = Clavier.lireInt();
            int v = Clavier.lireInt();
            int b = Clavier.lireInt();
            System.out.println("nouveau titre : (vide pour finir)");
            String tit = Clavier.lireString ();
            Color bg = new Color (r,v,b);
            if (tit.length() == 0) break;
            fen.setBounds(x,y,larg,haut);
            fen.setBackground(bg);
            fen.setTitle(tit);
        }
    }
}
```

1.4. **Exercice**

Créer une fenêtre personnelle d'un utilisateur qui choisit la taille, le titre et la couleur.

2. Gestion des clics

La programmation événementielle constitue la caractéristique essentielle d'interface graphique. La plupart des événements sont créés par des composants que l'on a introduit dans la fenêtre (menus, boutons, boîtes de dialogue, …). Nous allons voir comment traiter les événements générés par ces composants. Pour cela notre premier exemple est construit autour de la fenêtre sans qu'aucun autre objet ne soit créé.
Nous allons devoir implémenter des interfaces (classes abstraites). L'implémentation et les interfaces sont des notions que nous verrons plus tard.

2.1. L'interface MouseListener

Cette interface qui est une classe abstraite qui permet d'« écouter la souris ». Cette classe comporte 5 méthodes qui devront **toutes** être définies.
Lors d'une implémentation, toutes les méthodes de la classe implémentée doivent être définies. Nous verrons que ce n'est pas le cas lors d'une extension.
Les 5 méthodes sont :

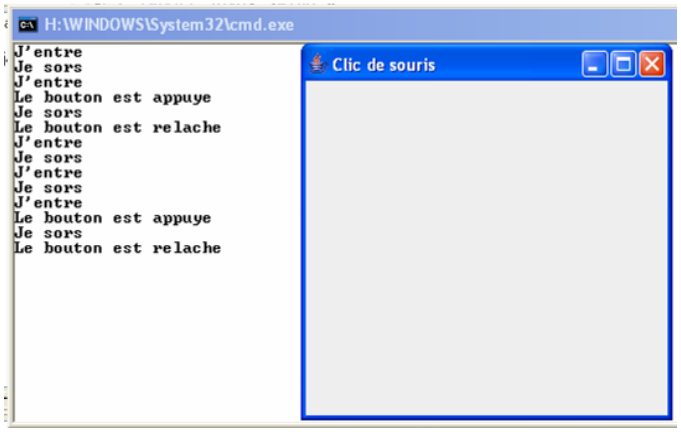
- ✓ *mouseClicked (MouseEvent ev) { …}* : clic usuel (bouton pressé et relâché immédiatement)
- ✓ *mousePressed (MouseEvent ev) { …}* : lorsque le bouton est pressé
- ✓ *mouseReleased (MouseEvent ev) { …}* : lorsque le bouton est relâché
- ✓ *mouseEntered (MouseEvent ev) { …}* : lorsque la souris arrive sur l'objet
- ✓ *mouseExited (MouseEvent ev) { …}* : lorsque la souris sort de l'objet

Si une méthode n'est pas à définir, il suffit de ne rien mettre entre les accolades.
Dans notre exemple, nous nous occupons uniquement de la première méthode en la définissant comme suit :

```
public void mouseClicked (MouseEvent ev) { System.out.println ("Clic dans la fenetre") ; }
```

Pour traiter un clic de souris dans notre fenêtre, il suffit d'associer celle-ci un objet de type *EcouteurSouris*. Pour ce faire, nous utilisons la méthode *addMouseListener* dans le constructeur et tel que l'objet lui-même soit l'écouteur.

```
import javax.swing.*;
import java.awt.event.*;
public class Clic1 {
   public static void main (String [] args){
      MaFenetre fen = new MaFenetre();
      fen.setVisible (true);
      fen.setDefaultCloseOperation(3);
   }
}
class MaFenetre extends JFrame implements MouseListener {
      public MaFenetre () {
         setBounds (150,150,300,300);
         setTitle ("Clic de souris");
         addMouseListener(this);
      }
   public void mouseClicked (MouseEvent ev) {
      System.out.println ("Clic dans la fenetre");
   }
   public void mousePressed (MouseEvent ev) {
      System.out.println ("Le bouton est appuye");
   }
   public void mouseReleased (MouseEvent ev) {
      System.out.println ("Le bouton est relache");
   }
```



```
    public void mouseEntered (MouseEvent ev) {
        System.out.println ("J'entre");
    }
    public void mouseExited (MouseEvent ev) {
        System.out.println ("Je sors");
    }
}
```

Dans ce code, nous avons été obligé d'importer un nouveau paquetage, celui qui concerne les écouteurs d'événements.

2.2. Utilisation de l'information associée à un événement

Dans notre cas, nous n'avons pas utilisé d'informations concernant l'événement « clic ». Mais on pourrait demander quelle est la position de la souris à l'endroit du clic.
Pour cela, il faut utiliser des méthodes concernant l'événement *ev* de type *MouseEvent*.
De plus, il existe deux méthodes *getX* et *getY* qui permettront de connaître l'abscisse et l'ordonnée à l'écran de la souris.
Il suffit de remplacer dans la méthode *mouseClicked* par ce code :

```
public void mouseClicked (MouseEvent ev) { int x = ev.getX() ; int y = ev.getY() ;
System.out.println(″La coordonnee de l'endroit du clic est ″ + x + ″, ″ + y) ; }
```

2.3. Exercice

Créer une application qui place la fenêtre à l'endroit du clic de la souris et si le nouvel endroit devient plus grand en X de 1000 et en Y de 700, le programme se ferme.

3. Premier composant : un bouton

3.1. Création d'un bouton et ajout dans la fenêtre

On crée un bouton à l'aide du constructeur de la classe *JButton*, auquel on ajoute du texte que l'on désire faire apparaître sur le bouton.

```
JButton monBouton = new JButton(″Essai″) ;
```

Maintenant, il faut l'introduire dans la fenêtre. Mais les choses ne sont pas si naturelles que l'on veut croire. En effet, il va falloir définir un container dans lequel on place les différents éléments que l'on veut, en l'occurrence dans notre cas, le bouton créé.

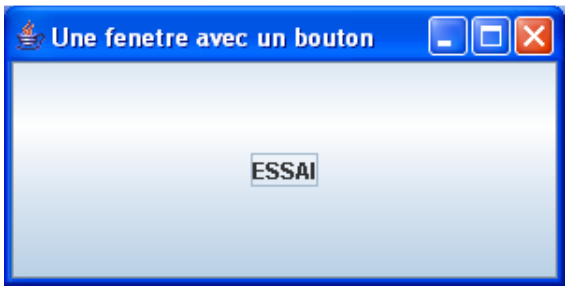
`Container co = getContentPane () ;` qui crée une variable de type *Container* et on lui réfère le contenu de la variable *JFrame* que l'on a créée au départ.

Il reste à utiliser la méthode *add* pour ajouter le bouton au container qui le fait apparaître dans la fenêtre : `co.add(monBouton) ;`

Si on n'a pas besoin de la variable *co*, on peut s'en passer et condenser les deux lignes en une seule comme suit : `getContentPane().add(monBouton) ;`

Contrairement au fenêtre, un bouton est directement visible, il est donc inutile d'appliquer la méthode *setVisible(true)*.

```
import javax.swing.*;
public class PremFen3 {
   public static void main (String [] args) {
      FenetreAvecBouton fen = new FenetreAvecBouton();
      fen.setVisible(true);
      fen.setDefaultCloseOperation(3);
   }
}
class FenetreAvecBouton extends JFrame {
   public FenetreAvecBouton () {
      setTitle ("Une fenetre avec un bouton");
      setSize (300, 150);
      JButton monBouton = new JButton ("ESSAI");
      getContentPane().add(monBouton);
   }
}
```



Il est à remarquer que le bouton prend tout l'espace disponible de la fenêtre graphique.

3.2. Gestionnaire de mise en forme

Pour palier au problème précédent, il faut utiliser un gestionnaire de mise en forme ou de disposition (*Layout Manager*).

Lorsque rien n'est indiqué, Java utilise le gestionnaire par défaut, la classe *BorderLayout* qui en absence d'informations spécifiques occupe toute la fenêtre.

Nous allons nous attarder sur un gestionnaire plus intéressant, la classe *FlowLayout*. Ce gestionnaire dispose les composants en « flots », c'est-à-dire les uns derrières les autres.

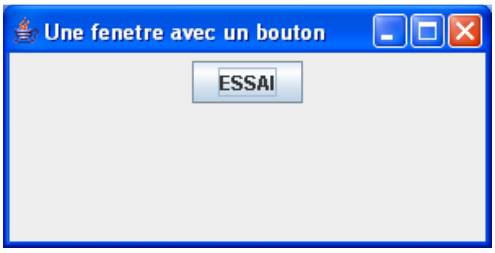
Pour choisir un gestionnaire, il faut appliquer la méthode *setLayout* à l'objet contenu de la fenêtre.

Ainsi si on a défini `Container co = getContentPane() ;` il suffira d'écrire `co.setLayout(new FlowLayout()) ;`.

Par contre, si on n'utilise pas *co*, alors cela devient `getContentPane().setLayout(new FlowLayout)) ;`.

Voici le code source précédent corrigé en ce sens :

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class PremFen4 {
   public static void main (String [] args) {
      FenetreAvecBouton fen = new FenetreAvecBouton();
      fen.setVisible(true);
      fen.setDefaultCloseOperation(3);
   }
}
class FenetreAvecBouton extends JFrame {
   public FenetreAvecBouton () {
      setTitle ("Une fenetre avec un bouton");
      setSize (300, 150);
      JButton monBouton = new JButton ("ESSAI");
      getContentPane().setLayout(new FlowLayout());
      getContentPane().add(monBouton);
   }
}
```



3.3. Gestion du bouton avec un écouteur

Un bouton ne peut déclencher qu'un seul événement correspondant à l'action de l'utilisateur sur ce bouton. Généralement, cette action est déclenchée par un clic sur le bouton, mais elle peut aussi être déclenchée à partir du clavier (sélectionner le bouton et appuyer sur la barre d'espacement).
La démarche expliquée au point 2.1 pour gérer un clic de souris dans une fenêtre s'applique pour gérer l'action sur un bouton. La différence est que la catégorie d'événements est nommée maintenant *Action*.
Il faut donc créer un écouteur qui est un objet d'une classe qui implémente l'interface *ActionListener* (cette classe ne comporte qu'une seule méthode *actionPerformed*) et d'associer cet écouteur au bouton par la méthode *addActionListener*.
Voici le code source d'une action qui permet de créer une autre fenêtre du même type :

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class PremFen5 {
   public static void main (String [] args) {
      FenetreAvecBouton fen = new FenetreAvecBouton();
      fen.setVisible(true);
   }
}
class FenetreAvecBouton extends JFrame implements ActionListener {
   public FenetreAvecBouton () {
      setTitle ("Une fenetre avec un bouton");
      setSize (300, 150);
      JButton monBouton = new JButton ("ESSAI");
      getContentPane().setLayout(new FlowLayout());
      getContentPane().add(monBouton);
      monBouton.addActionListener(this);
   }
   public void actionPerformed(ActionEvent ae) {
      FenetreAvecBouton win = new FenetreAvecBouton();
      win.setVisible(true);
      win.setDefaultCloseOperation(3);
   }
}
```

4. Gestion de plusieurs composants

Au point précédent, dans la fenêtre graphique, on y a placé qu'un seul bouton. Il est clair qu'une fenêtre ne doit pas en posséder qu'un seul. Au niveau de la création de nouveaux boutons, il suffit de procéder de la même manière que dans le point précédent.
Par contre, la gestion des actions sur ces différents boutons va différer. Java offre une grande liberté car chaque élément de chaque composant peut disposer de son propre objet écouteur. Nous allons voir différents cas mais qui ne sont pas les seuls.

4.1. **La fenêtre écoute les boutons**

On peut choisir que la fenêtre soit l'objet écouteur de tous les boutons. Dans ce cas, on peut agir selon notre choix :

- ✓ même action pour tous les boutons ;
- ✓ une action différente par bouton. On utilise dans ce cas *getSource* ou *getActionCommand* pour identifier le bouton concerné.

4.1.1. Tous les boutons déclenchent la même réponse

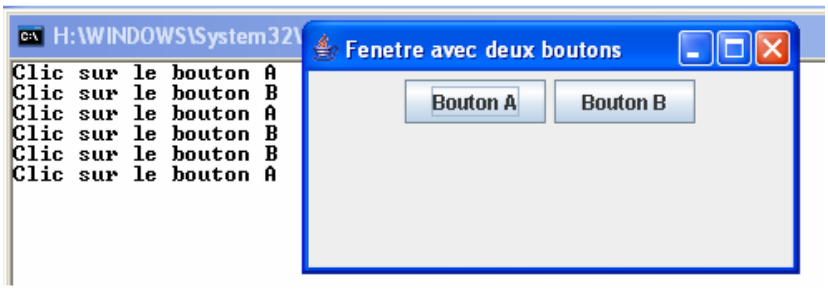
Dans l'exemple qui va suivre, tous les boutons déclenchent l'affichage à la console du même message : « Clic sur le bouton ».

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class PremFen6 {
   public static void main (String [] args) {
      FenetreAvecBouton fen = new FenetreAvecBouton();
      fen.setVisible(true);
      fen.setDefaultCloseOperation(3);
   }
}
class FenetreAvecBouton extends JFrame implements ActionListener {
   public FenetreAvecBouton () {
      setTitle ("Fenetre avec deux boutons");
      setSize (300, 150);
      JButton monBoutonA = new JButton ("Bouton A");
      JButton monBoutonB = new JButton ("Bouton B");
      Container c = getContentPane();
      c.setLayout(new FlowLayout());
      c.add(monBoutonA);
      c.add(monBoutonB);
      monBoutonA.addActionListener(this);
      monBoutonB.addActionListener(this);
   }
   public void actionPerformed(ActionEvent ae) {
      System.out.println("Clic sur le bouton");
   }
}
```

4.1.2. La méthode *getSource*

Dans l'exemple qui suit, on n'utilise qu'une seule méthode mais un test est effectué pour connaître le bouton actionné et donner une résultat en fonction du bouton choisi.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class PremFen7 {
   public static void main (String [] args) {
      FenetreAvecBouton fen = new FenetreAvecBouton();
      fen.setVisible(true);
      fen.setDefaultCloseOperation(3);
   }
}
class FenetreAvecBouton extends JFrame implements ActionListener {
   private JButton monBoutonA, monBoutonB;
   public FenetreAvecBouton () {
      setTitle ("Fenetre avec deux boutons");
      setSize (300, 150);
      monBoutonA = new JButton ("Bouton A");
      monBoutonB = new JButton ("Bouton B");
      Container c = getContentPane();
      c.setLayout(new FlowLayout());
      c.add(monBoutonA);
      c.add(monBoutonB);
      monBoutonA.addActionListener(this);
      monBoutonB.addActionListener(this);
   }
   public void actionPerformed(ActionEvent ae) {
      if (ae.getSource() == monBoutonA)
         System.out.println("Clic sur le bouton A");
      else
         System.out.println("Clic sur le bouton B");
   }
}
```



Il faut remarquer que jusqu'à présent, nous avions déclaré les boutons dans le constructeur *FenetreAvecBouton* mais cette fois du fait que nous avons besoin de ces variables pour les tester dans la méthode *actionPerformed*, il faut qu'elles soient déclarées pour toute la classe. Pour que ces variables ne puissent pas être affectées autrement que par les méthodes de la classe, il faut préciser que les variables sont *private*.

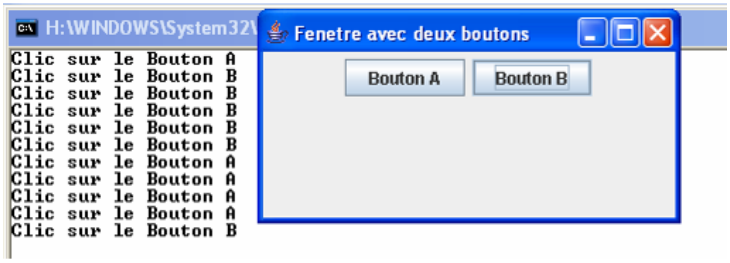
4.1.3. La méthode *getActionCommand*

La méthode *getSource* permet d'identifier la source de l'événement et elle a le mérite de s'appliquer à tous les événements générés par tous ses composants.
Il existe une autre méthode qui ne s'applique qu'aux événements de la catégorie *Action*. Cette méthode renvoie la chaîne de commande associée à la source de l'événement. Par défaut, elle renvoie le nom de l'étiquette de ce bouton.

Voici la source du programme précédent modifié en ce sens :

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class PremFen8 {
   public static void main (String [] args) {
      FenetreAvecBouton fen = new FenetreAvecBouton();
      fen.setVisible(true);
      fen.setDefaultCloseOperation(3);
   }
}
class FenetreAvecBouton extends JFrame implements ActionListener {
   public FenetreAvecBouton () {
      setTitle ("Fenetre avec deux boutons");
      setSize (300, 150);
```

```
        JButton monBoutonA = new JButton ("Bouton A");
        JButton monBoutonB = new JButton ("Bouton B");
        Container c = getContentPane();
        c.setLayout(new FlowLayout());
        c.add(monBoutonA);
        c.add(monBoutonB);
        monBoutonA.addActionListener(this);
        monBoutonB.addActionListener(this);
    }
    public void actionPerformed(ActionEvent ae) {
        String nom = ae.getActionCommand();
        System.out.println("Clic sur le " + nom);
    }
}
```



Dans cet exemple, comme nous n'avons pas besoin des variables *monBoutonA* et *monBoutonB*, elles peuvent ne plus être définies pour la classe entière mais rien que pour le constructeur. Cela évite d'écrire le mot *private* devant la déclaration.
Il est à noter que la méthode *getActionCommand()* donne, par défaut, le nom de l'étiquette mais il est possible de changer le nom de l'étiquette à l'aide de la méthode *setActionCommand(String texte) ;*.
Cela permet de laisser toujours la même chaîne de commande et d'uniquement changer le libellé qui peut être adapté à la langue si le programme doit fonctionner pour plusieurs langues.

4.2. Classe écouteur différente de la fenêtre

Cette fois ce n'est plus la fenêtre qui écoute les actions mais une classe ou plusieurs qui sont définies. On peut choisir qu'il y une classe écouteur par bouton ou une seule classe écouteur pour tous les boutons.

4.2.1. Une classe écouteur pour chaque bouton

Dans l'exemple qui suit, nous allons devoir créer deux classes écouteurs et donc définir la méthode *actionPerformed* dans nos deux classes.
On peut ne pas définir *monBoutonA* et *monBoutonB* comme *private* car on n'a pas besoin d'eux dans la classe.
Mais il ne s'agit pas non plus d'une erreur.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class PremFen9 {
    public static void main (String [] args) {
        FenetreAvecBouton fen = new FenetreAvecBouton();
        fen.setVisible(true);
        fen.setDefaultCloseOperation(3);
    }
}
class FenetreAvecBouton extends JFrame {
    private JButton monBoutonA, monBoutonB;
    public FenetreAvecBouton () {
        setTitle ("Fenetre avec deux boutons");
        setSize (300, 150);
        monBoutonA = new JButton ("Bouton A");
        monBoutonB = new JButton ("Bouton B");
        Container c = getContentPane();
        c.setLayout(new FlowLayout());
        c.add(monBoutonA);
```

```
        c.add(monBoutonB);
        EcouteBoutonA ecoutA = new EcouteBoutonA ();
        EcouteBoutonB ecoutB = new EcouteBoutonB ();
        monBoutonA.addActionListener(ecoutA);
        monBoutonB.addActionListener(ecoutB);
    }
}
class EcouteBoutonA implements ActionListener {
    public void actionPerformed(ActionEvent ae) {
        System.out.println("Clic sur le bouton A");
    }
}
class EcouteBoutonB implements ActionListener {
    public void actionPerformed(ActionEvent ae) {
        System.out.println("Clic sur le bouton B");
    }
}
```

4.2.2. Une seule classe écouteur pour les deux boutons

On peut utiliser une seule classe écouteur pour les deux boutons. Cela a pour avantage de ne spécifier qu'une seule fois la méthode *actionPerformed*.
Pour connaître qui des deux boutons a été actionnés on peut utiliser la méthode *getActionCommand* pour identifier le bouton concerné.
Voici une autre manière de procéder, il suffit d'associer à chaque objet écouteur créé un champ de valeur spécifique. Ce champ peut être initialisé lors de la construction de l'objet en attribuant une information lors de l'appel du constructeur.

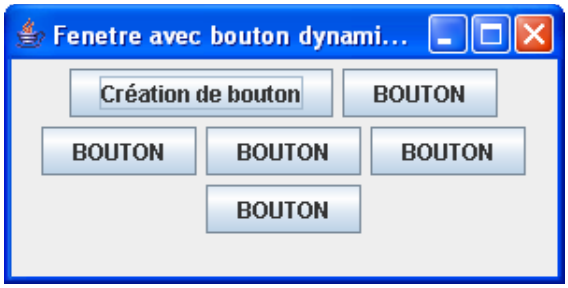
```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class PremFen10 {
    public static void main (String [] args) {
        FenetreAvecBouton fen = new FenetreAvecBouton();
        fen.setVisible(true);
        fen.setDefaultCloseOperation(3);
    }
}
class FenetreAvecBouton extends JFrame {
    private JButton monBoutonA, monBoutonB;
    public FenetreAvecBouton () {
        setTitle ("Fenetre avec deux boutons");
        setSize (300, 150);
        monBoutonA = new JButton ("Bouton A");
        monBoutonB = new JButton ("Bouton B");
        Container c = getContentPane();
        c.setLayout(new FlowLayout());
        c.add(monBoutonA);
        c.add(monBoutonB);
        EcouteBouton ecoutA = new EcouteBouton ("A");
        EcouteBouton ecoutB = new EcouteBouton ("B");
        monBoutonA.addActionListener(ecoutA);
        monBoutonB.addActionListener(ecoutB);
    }
}
class EcouteBouton implements ActionListener {
    private String texte;
    public EcouteBouton (String texte) {
        this.texte = texte ;
    }
    public void actionPerformed(ActionEvent ae) {
        System.out.println("Clic sur le bouton " + texte);
    }
}
```

4.3. Dynamique des composants

Lors de l'exemple du point 3.3, nous avons pu créer un bouton qui permettait de construire une nouvelle fenêtre. On peut aussi faire en sorte que lorsqu'on actionne un bouton, cela produit un ajout, une suppression, une activation ou encore une désactivation de bouton. On parle alors de composant dynamique.

Exemple 1 :

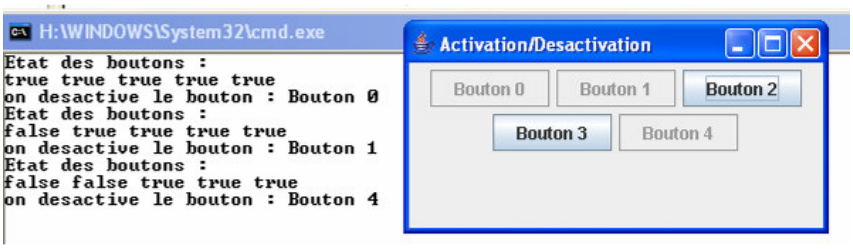
```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class PremFen11 {
   public static void main (String [] args) {
      FenetreAvecBouton fen = new FenetreAvecBouton();
      fen.setVisible(true);
      fen.setDefaultCloseOperation(3);
   }
}
class FenetreAvecBouton extends JFrame {
   private JButton monBouton;
   public FenetreAvecBouton () {
      setTitle ("Fenetre avec bouton dynamique");
      setSize (300, 150);
      monBouton = new JButton ("Création de bouton");
      Container c = getContentPane();
      c.setLayout(new FlowLayout());
      c.add(monBouton);
      EcouteBouton ecout = new EcouteBouton (c);
      monBouton.addActionListener(ecout);
   }
}
class EcouteBouton implements ActionListener {
   private Container c;
   public EcouteBouton (Container c) {
      this.c = c ;
   }
   public void actionPerformed(ActionEvent ae) {
      JButton nouveauBouton = new JButton ("BOUTON");
      c.add(nouveauBouton);
      c.validate();
   }
}
```



Exemple 2 :

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class PremFen12 {
   public static void main (String [] args) {
      FenetreAvecBouton fen = new FenetreAvecBouton();
      fen.setVisible(true);
      fen.setDefaultCloseOperation(3);
   }
}
class FenetreAvecBouton extends JFrame implements ActionListener {
   final int NBOUTONS = 5;
   private JButton tabBouton[];
   public FenetreAvecBouton () {
      setTitle ("Activation/Desactivation");
      setSize (300, 150);
      Container c = getContentPane();
      c.setLayout(new FlowLayout());
```

```
        tabBouton = new JButton[NBOUTONS];
        for (int i=0 ; i < NBOUTONS ; i++) {
            tabBouton[i] = new JButton ("Bouton " + i);
            c.add(tabBouton[i]);
            tabBouton[i].addActionListener(this);
        }
    }
    public void actionPerformed(ActionEvent ae) {
        System.out.println ("Etat des boutons : ");
        for (int i = 0 ; i < NBOUTONS ; i++) {
            System.out.print (tabBouton[i].isEnabled() + " ");
        }
        System.out.println();
        JButton source = (JButton) ae.getSource();
        System.out.println ("on desactive le bouton : " + source.getActionCommand());
        source.setEnabled(false);
    }
}
```



4.4. **Exercice**

Créer une application calculatrice dont l'affichage se fait à la fenêtre console.

5. Les gestionnaires de mise en forme

Nous avons déjà utilisé deux des gestionnaires *BorderLayout* (gestionnaire par défaut) et *FlowLayout* dans nos exemples précédents. Voyons plus en détails ces deux gestionnaires ainsi que d'autres.
Ces gestionnaires servent à disposer des composants dans les fênetres ou les panneaux.

5.1. Le gestionnaire *BorderLayout*

Ce gestionnaire dispose les composants suivant l'un des quatre bords du conteneur ou au centre.
Les composants de bord ont une largeur fixe par contre les composants au centre prennent la largeur restante.Tant qu'un bord n'est pas utilisé, le composant central utilise ce côté.
Pour le positionnement, c'est lors de l'invocation à la méthode *add* que l'on doit ajouter une des constantes suivantes :

Constante symbolique	Valeur	Emplacement correspondant
BorderLayout.NORTH	"North"	En haut
BorderLayout.SOUTH	"South"	En bas
BorderLayout.EAST	"East"	A droite
BorderLayout.WEST	"West"	A gauche
BorderLayout.CENTER	"Center"	Au centre

Si aucune valeur n'est précisée, le composant est ajouté au centre.
Par défaut les composant sont espacés de 5 pixels. Cette valeur peut être modifiée au moment de la création à l'aide de deux paramètres : conteneur.setLayout(new BorderLayout(10,20))
Si le gestionnaire est lié à une variable, on peut défir ces espacements à l'aide des méthodes *setHgap* pour l'intervalle horizontal et *setVgap* pour l'intervalle vertical.

Exemple :

```
import javax.swing.*;
import java.awt.*;
public class Gestionnaire1 {
   public static void main (String args[]) {
      MaFenetre fen = new MaFenetre() ;
      fen.setVisible(true);
      fen.setDefaultCloseOperation(3);
   }
}
class MaFenetre extends JFrame {
   public MaFenetre() {
      setTitle("Exemple de BorderLayout");
      setBounds(200,200,300,180);
      Container c = getContentPane();
      JButton [] bouton = new JButton[5];
      for (int i = 0 ; i < 5 ; i++ ) bouton[i] = new JButton("Bouton "+ i);
      c.add(bouton[0]);
      c.add(bouton[1],BorderLayout.NORTH);
      c.add(bouton[2],BorderLayout.SOUTH);
      c.add(bouton[3],BorderLayout.EAST);
      c.add(bouton[4],BorderLayout.WEST);
   }
}
```

5.2. Le gestionnaire *FlowLayout*

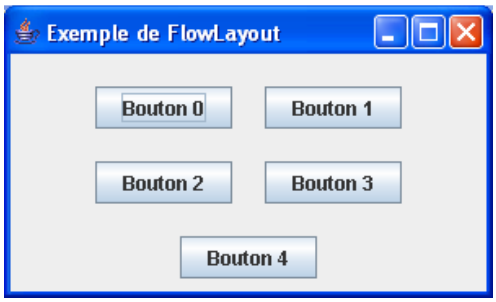
Ce gestionnaire dispose les composants les uns à la suite des autres, sur une même ligne. Si la ligne ne possède plus suffisamment de place, l'affichage se poursuit à la ligne.
Dans ce gestionnaire, la taille des composants est respectée.
Les composants peuvent avoir une taille soit prédéfinie soit choisie par le concepteur par la méthode *setPreferredSize*.
Lors de la construction, on peut définir un paramètre concernant l'alignement des composant.
Par défaut, l'alignement est à gauche.

Constante symbolique	Valeur	Emplacement correspondant
FlowLayout.LEFT	"Left"	A gauche
FlowLayout.RIGHT	"Right"	A droite
FlowLayout.CENTER	"Center"	Au centre

Il est aussi possible comme dans le gestionnaire précédent de définir un espacement horizontal ou vertical à la construction : conteneur.setLayout(new FlowLayout(FlowLayout.Right,10,20))
ou à l'aide des méthodes *setHgap* pour l'intervalle horizontal et *setVgap* pour l'intervalle vertical.

Exemple :

```
import javax.swing.*;
import java.awt.*;
public class Gestionnaire2 {
   public static void main (String args[]) {
      MaFenetre fen = new MaFenetre() ;
      fen.setVisible(true);
      fen.setDefaultCloseOperation(3);
   }
}
class MaFenetre extends JFrame {
   public MaFenetre() {
      setTitle("Exemple de FlowLayout");
      setBounds(200,200,300,180);
      Container c = getContentPane();
      c.setLayout(new FlowLayout(FlowLayout.CENTER,20,20));
      JButton [] bouton = new JButton[5];
      for (int i = 0 ; i < 5 ; i++ ) {
         bouton[i] = new JButton("Bouton "+ i);
         c.add(bouton[i]);
      }
   }
}
```



En combinant ces deux gestionnaires, on peut aboutir à des présentations élaborées.
Un container avec un gestionnaire peut posséder un autre container ou composant container (JPanel) qui lui aura son propre gestionnaire.
Ces deux gestionnaires sont fournis de base dans la version de Java.

5.3. Le gestionnaire *CardLayout*

Ce gestionnaire permet de disposer les comosants suivant une pile, de telle sorte que seul le composant supérieur est visible.
Il y a des méthodes qui permettent de parcourir la pile ou de se placer à un composant donné.
Pour ce type de gestionnaire, il est préférable de garder une trace de variable pour pouvoir accéder directement à l'objet.
La construction se fait donc comme suit :

```
Container c = getContentPane() ;
CardLayout pile = new CardLayout(30,30) ; //les valeurs permettent de définir les espacements horizontaux et verticaux.
c.setLayout(pile) ;
```

Les méthodes :

- ✓ `c.add(compo,"un certain composant") ;` : permet d'itenfier le composant ajouté par un nom spécifique ;
- ✓ `pile.next(c) ;` : permet d'afficher le composant suivant ;
- ✓ `pile.previous(c) ;` : permet d'afficher le composant précédent ;
- ✓ `pile.first(c) ;` : permet d'afficher le premier composant ;
- ✓ `pile.last(c) ;` : permet d'afficher le dernier composant ;
- ✓ `pile.show(c,"un certain composant") ;` : permet d'afficher le composant du nom spécifique mentionné.

Exemple :


```
import javax.swing.*; import java.awt.*;
import java.awt.event.*; import javax.swing.event.*;
public class Gestionnaire3 {
   public static void main (String args[]) {
      MaFenetre fen = new MaFenetre() ;
      fen.setVisible(true);
      fen.setDefaultCloseOperation(3);
   }
}
class MaFenetre extends JFrame implements ActionListener{
   private JButton suiv,prec,prem,dern;
   private JPanel panCard;
   private CardLayout pile;
   public MaFenetre() {
      setTitle("Exemple de CardLayout");
      setBounds(200,200,400,200);
      Container c = getContentPane();
      pile = new CardLayout(30,30);
      panCard = new JPanel();
      JPanel panCom = new JPanel();
      c.add(panCard); c.add(panCom,"South");
      panCard.setLayout(pile);
      JButton [] bouton = new JButton[5];
      for (int i = 0 ; i < 5 ; i++ ) {
         bouton[i] = new JButton("Bouton "+ i);
         panCard.add(bouton[i],"Bouton");
      }
      prec = new JButton("Précédent"); suiv = new JButton("Suivant"); prem = new JButton("Premier"); dern = new JButton("Dernier");
      panCom.add(prec); panCom.add(suiv); panCom.add(prem); panCom.add(dern);
      prec.addActionListener(this); suiv.addActionListener(this); prem.addActionListener(this); dern.addActionListener(this);
   }
   public void actionPerformed(ActionEvent e) {
      JButton source = (JButton) e.getSource();
      if (source == prec) pile.previous(panCard);
         else if (source == suiv) pile.next(panCard);
            else if (source == prem) pile.first(panCard);
               else pile.last(panCard);
   }
}
```

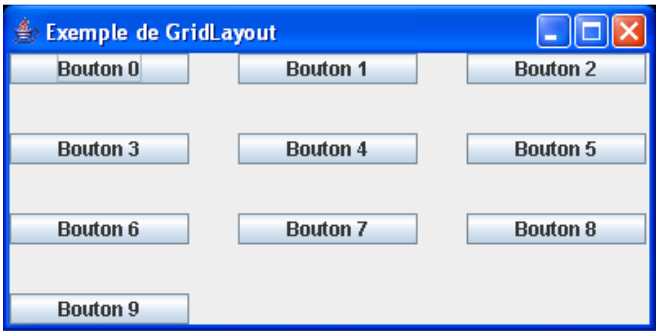
5.4. Le gestionnaire *GridLayout*

Ce gestionniare permet de disposer les différents composant suivant une grille régulière, chaque composant occupant une cellule.
A la construction, on choisit le nombre de lignes et de colonnes de la grille et éventuellement les espacement horizontaux et verticaux comme suit : conteneur.setLayout(new GridLayout(5,4,30,30) ;
Les composants sont ajoutés aux cases selon l'ordre dans lequel on les ajoute au conteneur (le parcours se fait suivant les lignes). Il est possible que les dernières cases restent vides. Toutefois, s'il y a pour une ligne d'éléments vide, le gestionnaire réorganise la grille pour éviter la perte de place.
La taille de chaque composant est étudiée par le gestionnaire ne fonction du conteneur.

Exemple :

Ce programme permet de disposer 10 boutons pour une grille 4 X 3.

```
import javax.swing.*;
import java.awt.*;
public class Gestionnaire4 {
   public static void main (String args[]) {
      MaFenetre fen = new MaFenetre() ;
      fen.setVisible(true);
      fen.setDefaultCloseOperation(3);
   }
}
class MaFenetre extends JFrame {
   public MaFenetre() {
      setTitle("Exemple de CardLayout");
      setBounds(200,200,400,200);
      Container c = getContentPane();
      c.setLayout(new GridLayout(4,3,30,30));
      JButton [] bouton = new JButton[10];
      for (int i = 0 ; i < 10; i++ ) {
         bouton[i] = new JButton("Bouton "+ i);
         c.add(bouton[i]);
      }
   }
}
```



5.5. Le gestionnaire *BoxLayout*

Ce gestionnaire permet de disposer les composants suivant une seule ligne ou une seule colonne. Comme ce gestionnaire est associé à un conteneur de type *Box*, il permet plus de souplesse que le *GridLayout* à une ligne ou une colonne.
Construction :

- ✓ Box ligne = Box.createHorizontalBox() ; : création d'un box horizontal ;
- ✓ Box colonne = Box.createVerticalBox() ; : création d'un box vertical ;

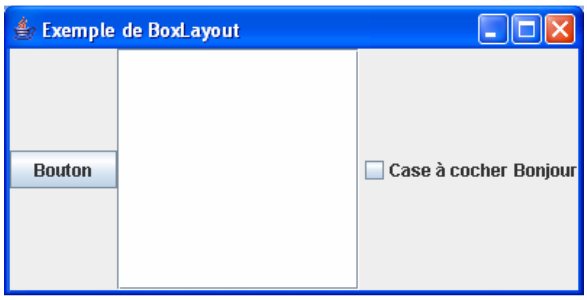
Ces conteneurs sont dotés par défaut d'un gestionnaire de type *BoxLayout*.
Dans un box horizontal, tous les composants sont ajoutés de gauche à droite ; il s sont contigus et occupent toute la largeur et toute la hauteur du conteneur. S'il y a trop de composant, certains ne sont pas visibles.
Dans un box vertical, tous les composants sont ajoutés de haut en bas ; il s sont contigus et occupent toute la largeur et toute la hauteur du conteneur. S'il y a trop de composant, certains ne sont pas visibles.

5.5.1. Box horizontal

Voici un programme qui place un bouton, un champ texte (de longueur 20), une case à cocher et une étiquette.

Exemple :

```
import javax.swing.*;
import java.awt.*;
public class Gestionnaire5 {
   public static void main (String args[]) {
      MaFenetre fen = new MaFenetre() ;
      fen.setVisible(true);
      fen.setDefaultCloseOperation(3);
   }
}
class MaFenetre extends JFrame {
   public MaFenetre() {
      setTitle("Exemple de BoxLayout");
      setBounds(200,200,400,200);
      Container c = getContentPane();
      Box bHor = Box.createHorizontalBox();
      c.add(bHor);
      JButton bouton = new JButton("Bouton");
      bHor.add(bouton);
      JTextField texte = new JTextField(20);
      bHor.add(texte);
      JCheckBox coche = new JCheckBox("Case à cocher");
      bHor.add(coche);
      JLabel label = new JLabel("Bonjour");
      bHor.add(label);
   }
}
```

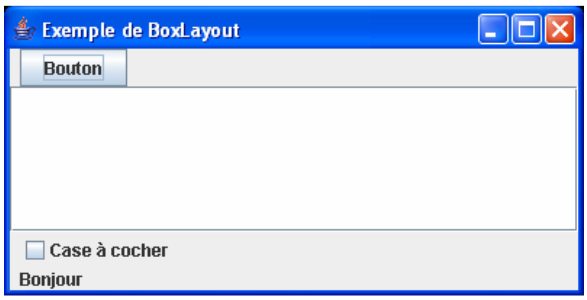


5.5.2. Box vertical

Programme identique pour un box vertical.

Exemple :

```
import javax.swing.*;
import java.awt.*;
public class Gestionnaire6 {
   public static void main (String args[]) {
      MaFenetre fen = new MaFenetre() ;
      fen.setVisible(true);
      fen.setDefaultCloseOperation(3);
   }
}
class MaFenetre extends JFrame {
   public MaFenetre() {
      setTitle("Exemple de BoxLayout");
      setBounds(200,200,400,200);
      Container c = getContentPane();
      Box bVer = Box.createVerticalBox();
      c.add(bVer);
      JButton bouton = new JButton("Bouton");
      bVer.add(bouton);
      JTextField texte = new JTextField(20);
      bVer.add(texte);
      JCheckBox coche = new JCheckBox("Case à cocher");
      bVer.add(coche);
      JLabel label = new JLabel("Bonjour");
      bVer.add(label);
   }
}
```



5.6. Le gestionnaire *GridBagLayout*

Ce gestionnaire est le plus souple fournit par Java mais aussi le plus difficile à employer.
Il permet de disposerles composant suivant une grille mais les composans peuvent occuper plusieurs cases.
La construction : `GridBagLayout g = new GridBagLayout() ;`
Ce n'est pas à la construction que l'on fournit le nombre de lignes et de colonnes mais lors de l'ajout par des paramètres :

Paramètre	Signification
gridx	Abscisse dans la grille du coiin supérieur gauche du composant
gridy	Ordonnée dans la grille du coiin supérieur gauche du composant
gridwidth	Largeur du composant
gridheight	Hauteur du composant
weightx	Poids horizontal du composant
weighty	Poids vertical du composant
fill	Manière dont le composant occupe l'espace disponible : ✓ GridBagConstraints.HORIZONTAL : largeur ajustée à l'espace disponible ✓ GridBagConstraints.VERTICAL : hauteur ajustée à l'espace disponible ✓ GridBagConstraints.BOTH : largeur et hauteur ajustées à l'espace disponible ✓ GridBagConstraints.NONE : aucun ajustement

Exemple :

Dans cet exemple, on place les composants suivant le modèle :
B1, B4 et B5 ont une largeur de 60 et B2, B3 et B6 de 40.
B2, B3, B4 et B5 ont une hauteur de 25.

B1	B2
	B3
B4	B6
B5	

```
import javax.swing.*; import java.awt.*;
public class Gestionnaire7 {
   public static void main (String args[]) {
      MaFenetre fen = new MaFenetre() ;
      fen.setVisible(true);
      fen.setDefaultCloseOperation(3);
   }
}
class MaFenetre extends JFrame {
   public static int x [] = {0,3,3,0,0,3};
   public static int y [] = {0,0,1,2,3,2};
   public static int larg [] = {3,2,2,3,3,2};
   public static int haut [] = {2,1,1,1,1,2};
   public static int px [] = {60,40,0,0,0,0};
   public static int py [] = {0,25,25,25,25,0};
   public MaFenetre() {
      setTitle("Exemple de GridBagLayout");
      setBounds(200,200,400,200);
      Container co = getContentPane();
      GridBagLayout g = new GridBagLayout();
      co.setLayout(g);
      GridBagConstraints c = new GridBagConstraints();
      c.fill= c.BOTH;
      for (int i = 0 ; i < x.length ; i++) {
         c.gridx = x[i]; c.gridy = y[i]; c.gridwidth = larg[i]; c.gridheight = haut[i]; c.weightx = px[i]; c.weighty = py[i];
         co.add(new JButton("Bouton"+(i+1)),c);
      }
   }
}
```

5.7. Le gestionnaire *XYLayout*

Ce gestionnaire qui ne se trouve pas dans le package d'installation de Java doit toujours être placé dans le dossier de travail en cours.

Création : `Container c = getContentPane();`
`c.setLayout(new XYLayout(0,0));`

Les valeurs dans *XYLayout* sont les dimensions : longueur et hauteur, de l'éléments.
Celles-ci peuvent être mises à 0 si le composant possède déjà une dimension.

Ajout de composants : `c.add("0,547,60,20",bouton);` : chaîne de caractère avec la position horizontale, verticale, la longueur et la hauteur du composant.

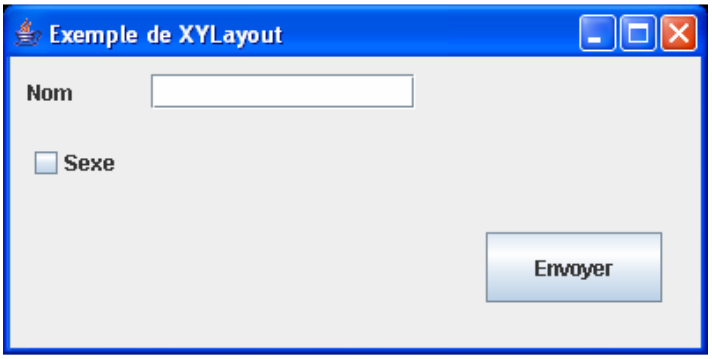
On doit être écrire une chaîne de caractère car la méthode fait appel à une méthode de la classe *LayoutManager*, *addLayoutComponent(String n, Component c) ;*.

Exemple :

Dans cette exemple, on place :

- ✓ une étiquette dont le texte est « Nom » à 10 pixels du bord gauche et à 10 pixels du haut et une dimension de 50 pixels en longueur et 20 pixels en hauteur ;
- ✓ une zone de texte à 20 pixels de l'élément précédent, de longueur 150 pixels et de hauteur 20 pixels.
- ✓ une case à cocher avec l'étiquette « Sexe » à 20 pixels plus bas et une longueur de 100 pixels pour une hauteur de 20 pixels ;
- ✓ un bouton « Envoyer » à 30 pixels du bord droit et une hauteur de 100 pixels, pour une dimension de 100 par 40 pixels.

```
import javax.swing.*;
import java.awt.*;
public class Gestionnaire8 {
   public static void main (String args[]) {
      MaFenetre fen = new MaFenetre() ;
      fen.setVisible(true);
      fen.setDefaultCloseOperation(3);
   }
}
class MaFenetre extends JFrame {
   public MaFenetre() {
      setTitle("Exemple de XYLayout");
      setBounds(200,200,400,200);
      Container c = getContentPane();
      c.setLayout(new XYLayout(0,0));
      JLabel nom = new JLabel("Nom");
      JTextField champNom = new JTextField();
      JCheckBox sexe = new JCheckBox("Sexe");
      JButton bouton = new JButton("Envoyer");
      c.add("10,10,50,20",nom);
      c.add("80,10,150,20",champNom);
      c.add("10,50,100,20",sexe);
      c.add("270,100,100,40",bouton);
   }
}
```



6. Premier dessin

Java permet de dessiner sur n'importe quel composant en utilisant des méthodes de dessin. Cependant, si vous utilisez directement ces méthodes, vous obtenez le dessin attendu mais il disparaît en totalité ou en partie dès que la fenêtre contenant le composant a besoin d'être réaffichée : comme cela peut arriver en cas de modification de sa taille, de déplacement, de restauration après une réduction en icône, …
Pour obtenir vos dessins en permanence, il est nécessaire de placer les instructions de dessin dans une méthode particulière du composant concerné, nommé *paintComponent*. Cette méthode est automatiquement appelée par Java chaque fois que le composant a besoin d'être dessiné ou redessiné.
Sans rentrer dans les problèmes historique et technique du Java, pour dessiner des objets, on ne travaille pas directement sur la fenêtre mais dans un panneau. Une fenêtre pourra être composée d'un ou plusieurs panneaux.

6.1. Création de panneau

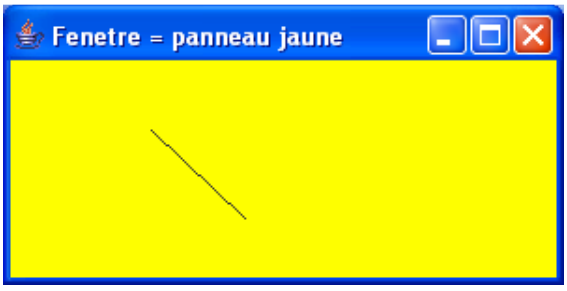
La fenêtre est un composant contenant (container) des composants. Ces composants, quant à eux, ne peuvent contenir d'autres composants. Ces composants basiques sont appelés *composants atomiques*. Il existe des composants intermédiaires qui se trouvent dans les fenêtres mais qui peuvent contenir aussi des composants. C'est le cas des panneaux.
Les panneaux sont en quelque sorte des sous-fenêtres, sans titre, ni bordure. Tant qu'on ne lui attribue aucune couleur, il sera impossible de distinguer la différence.
Il ne faut pas oublier d'ajouter le composant *panneau* dans le container de la fenêtre.
Dans ce premier exemple, nous construisons une fenêtre contenant un panneau de couleur jaune. Mais du fait que la mise en forme de la fenêtre est celle par défaut, le panneau prendra les dimensions complètes de la fenêtre.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class FenPan1 {
   public static void main (String [] args) {
      FenetreAvecPanneau fen = new FenetreAvecPanneau ();
      fen.setVisible(true);
      fen.setDefaultCloseOperation(3);
   }
}
class FenetreAvecPanneau extends JFrame {
   private JPanel panneau;
   public FenetreAvecPanneau () {
      setTitle ("Fenetre = panneau jaune");
      setSize (300, 150);
      panneau = new JPanel();
      Container c = getContentPane();
      panneau.setBackground (Color.yellow);
      c.add(panneau);
   }
}
```

6.2. Dessin dans un panneau

Pour obtenir un dessin permanent, il faut redéfinir la méthode *paintComponent*. Pour que l'on puisse redéfinir cette méthode, il faut alors que *panneau* soit un objet d'une classe dérivée de *JPanel*. La méthode a comme entête : *void paintComponent (Graphics g)*.
Son unique argument est un objet de la classe *Graphics* qui sert d'intermédiaire entre vos demandes de dessins et leur réalisation effective.
Nous nous contentons de dessiner une ligne, la méthode qui le fait est *drawLine*.
Dans le constructeur de la classe *Panneau*, on utilise un mot clé *super* (voir suite).
Voici un exemple :

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class FenPan2 {
   public static void main (String [] args) {
      FenetreAvecPanneau fen = new FenetreAvecPanneau ();
      fen.setVisible(true);
      fen.setDefaultCloseOperation(3);
   }
}
class FenetreAvecPanneau extends JFrame {
   private JPanel pan;
   public FenetreAvecPanneau () {
      setTitle ("Fenetre = panneau jaune");
      setSize (300, 150);
      pan = new Panneau();
      Container c = getContentPane();
      pan.setBackground (Color.yellow);
      c.add(pan);
   }
}
class Panneau extends JPanel {
   public void paintComponent (Graphics g) {
      super.paintComponent(g);
      g.drawLine (15, 10 , 100, 50);
   }
}
```

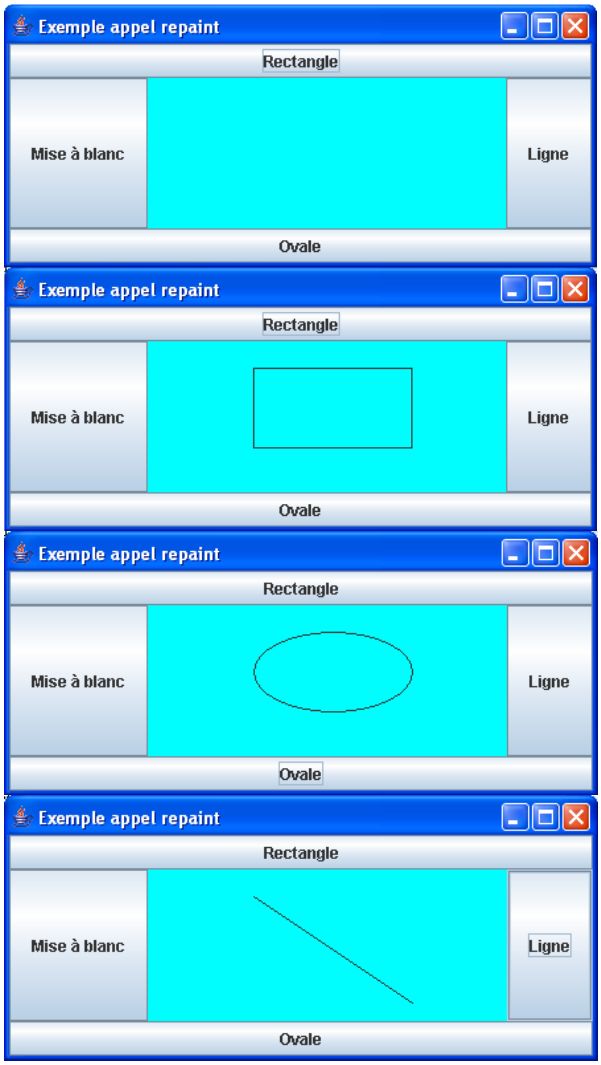


6.3. Forcer le dessin

Nous avons dessiné une ligne dès le début et celle-ci n'a pas été modifiée par la suite.
Souvent, on a besoin de dessiner à la suite d'une action de l'utilisateur, par exemple avec un bouton.
Voici un exemple contenant deux boutons, l'un permet de dessiner une ellipse dans le panneau et le second de dessiner à la place de l'ellipse un rectangle. Au démarrage, il n'y a rien d'afficher.
Le gestionnaire de mise en forme par défaut, nous permet à l'aide de *North*, *South*, *East* et *West* de placer les objets aux quatre points cardinaux du panneau.
Lorsque rien n'est indiqué, le composant se placera au centre du panneau.

```
import javax.swing.*; import java.awt.*; import java.awt.event.*;
public class FenPan3 {
   public static void main (String [] args) {
      FenetreAvecPanneau fen = new FenetreAvecPanneau ();
      fen.setVisible(true);
      fen.setDefaultCloseOperation(3);
   }
}
```

```
class FenetreAvecPanneau extends JFrame implements ActionListener{
   private Panneau pan;
   private JButton rectangle, ovale, ligne, rien;
   public FenetreAvecPanneau () {
      setTitle ("Exemple appel repaint");
      setBounds (300, 300 , 450 , 200);
      pan = new Panneau();
      Container c = getContentPane();
      pan.setBackground (Color.cyan);
      c.add(pan);
      rectangle = new JButton ("Rectangle");
      ovale = new JButton ("Ovale");
      ligne = new JButton ("Ligne");
      rien = new JButton ("Mise à blanc");
      c.add(rectangle,"North");
      c.add(ovale,"South");
      c.add(ligne,"East");
      c.add(rien,"West");
      rectangle.addActionListener(this);
      ovale.addActionListener(this);
      ligne.addActionListener(this);
      rien.addActionListener(this);
   }
   public void actionPerformed (ActionEvent ae) {
      if (ae.getSource() == rectangle) pan.setRectangle();
      else if (ae.getSource() == ovale) pan.setOvale();
         else if (ae.getSource() == ligne) pan.setLine();
            else pan.setRien();
      pan.repaint();
   }
}
class Panneau extends JPanel {
   private boolean rectangle = false, ovale = false, ligne = false;
   public void paintComponent (Graphics g) {
      super.paintComponent(g);
      if (ovale) g.drawOval (80,20,120,60);
      if (rectangle) g.drawRect (80,20,120,60);
      if (ligne) g.drawLine (80,20,200,100);
   }
   public void setRectangle () {
      rectangle = true;
      ovale = false;
      ligne = false;
   }
   public void setOvale () {
      rectangle = false;
      ovale = true;
      ligne = false;
   }
   public void setLine() {
      rectangle = false;
      ovale = false;
      ligne = true;
   }
   public void setRien () {
      rectangle = false;
      ovale = false;
      ligne = false;
   }
}
```



6.4. Exercice

Créer une application pour laquelle l'utilisateur choisi parmi 3 boutons le dessin qu'il veut réaliser. A la fenêtre console, il faut qu'il introduise les coordonnées des points supérieur gauche et inférieur droit des objets à dessiner. Un dernier bouton effaçant tout.

7. Les cases à cocher

7.1. Création

La case à cocher permet à l'utilisateur d'effectuer un choix de type oui/non. La classe qui définit ce type d'objet est *JCheckBox*, dont le constructeur requiert obligatoirement un libellé qui est affiché à côté de la case. Il faut, comme tout composant, l'ajouter dans un container. L'action sur la case à cocher se limite à la modification de son état.
Par défaut une case à cocher est décochée mais on peut à l'aide d'un autre constructeur définir son état initial.

7.2. Exploitation d'une case à cocher

Deux possibilités sont souvent utilisées pour un objet de ce type :
- ✓ réagir immédiatement à une action
- ✓ connaître son état à un instant donné

7.2.1. Réaction à l'action sur la case à cocher

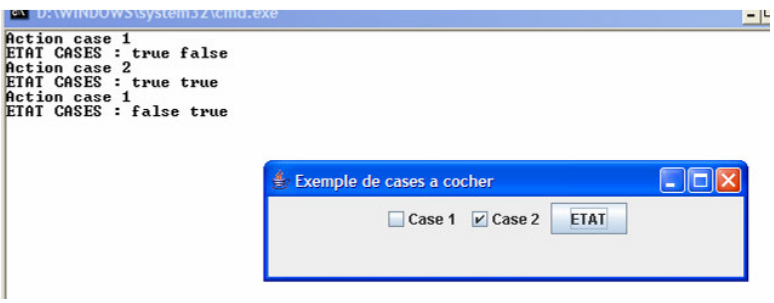
Deux possibilités se présentent, soit un événement *Action* soit un événement *Item*. Dans les deux cas, il y a aura un écouteur.
Si l'écouteur est lié à l'événement de type *Action*, il devra implémenter l'interface *ActionListener* et redéfinir la méthode *actionPerformed*.
Par contre, si l'écouteur est lié à l'événement de type *Item*, il devra implémenter l'interface *ItemListener* et redéfinir la seule méthode *itemStateChanged*.

7.2.2. Etat d'une case à cocher

On peut connaître l'état d'une case à un moment donné. Il suffit de faire appel à la méthode *isSelected* de la classe *JCheckBox*.
On peut aussi changer à tout moment, sans devoir effectuer un action, imposer à une case un état donné à l'aide de la méthode *setSelected*. Il est à noter qu'un tel appel génère un événement de type *Item*.

7.3. Exemple

```
import java.awt.*; import java.awt.event.*; import javax.swing.*;
public class ControleUsuel1 {
   public static void main(String args[]){
      CaseACocher cas = new CaseACocher();
      cas.setVisible(true);
      cas.setDefaultCloseOperation(3);
   }
}
class CaseACocher extends JFrame implements ActionListener {
   private JCheckBox coche1, coche2;
   private JButton etat;
   public CaseACocher() {
      setTitle ("Exemple de cases a cocher");
      setSize(400,100);
      Container co = getContentPane();
      co.setLayout(new FlowLayout());
      coche1 = new JCheckBox("Case 1");
      co.add(coche1);
      coche1.addActionListener(this);
      coche2 = new JCheckBox("Case 2");
      co.add(coche2);
```



```
        coche2.addActionListener(this);
        etat =new JButton("ETAT");
        co.add(etat);
        etat.addActionListener(this);
    }
    public void actionPerformed (ActionEvent ev) {
        Object source = ev.getSource();
        if(source == coche1) System.out.println("Action case 1");
        if(source == coche2) System.out.println("Action case 2");
        if(source == etat) System.out.println ("ETAT CASES : "+ coche1.isSelected() + " " + coche2.isSelected());
    }
}
```

Si, au lieu d'utiliser des écouteurs de type *Action*, on avait utilisé des écouteurs de type *Item*, `coche1.addActionListener(this);` devient `coche1.addItemListener(this);`.
Il en est de même pour l'objet *coche2*. Il ne faut pas oublier d'ajouter *ItemListener* après *implements ActionListener*.
Par contre, il faut scinder la méthode *actionPerformed*, la garder pour le bouton et écrire une nouvelle méthode *itemStateChanged* comme suit :

```
    public void itemStateChanged (ItemEvent ev) {
        Object source = ev.getSource();
        if(source == coche1) System.out.println("Action case 1");
        if(source == coche2) System.out.println("Action case 2");
    }
    public void actionPerformed (ActionEvent ev) {
    System.out.println ("ETAT CASES : "+ coche1.isSelected() + " " + coche2.isSelected());
    }
```

7.4. **Exercice**

Créer une application qui en fonction des cases à cocher « sexe » et « majeur » écrive une des informations suivantes à la fenêtre console :

- ✓ Fille mineure : « Ce jeu est interdit au fille minueure » ;
- ✓ Garçon mineur : « Ce jeu est interdit au garçon mineur » ;
- ✓ Fille majeure : « Ce jeu vous est autorisé sans aucune restriction » ;
- ✓ Garçon majeur : « Ce jeu vous est autorisé mais avec des restrictions ».

8. Les boutons radio

8.1. Création

Le bouton radio permet à l'utilisateur d'effectuer un choix de type oui/non. Contrairement à la case à cocher, il fait partie d'un groupe dans lequel seul un bouton radio sera actif.
La classe qui définit ce type d'objet est *JRadioButton*, dont le constructeur requiert obligatoirement un libellé qui est affiché à côté du bouton radio.
Il faut, comme tout composant, l'ajouter dans un container après l'avoir ajouté dans un groupe qui est un objet de type *ButtonGroup*.Ce dernier objet sert à assurer la désactivation automatique d'un bouton lorsqu'un autre du groupe est activé.
Par défaut un bouton radio est construit dans l'état non sélectionné mais on peut à l'aide d'un autre constructeur définir son état initial.

8.2. Exploitation d'un bouton radio

Deux possibilités sont souvent utilisées pour un objet de ce type :
- ✓ réagir immédiatement à une action
- ✓ connaître son état à un instant donné

8.2.1. Réaction à l'action sur le bouton radio

Deux possibilités se présentent, soit un événement *Action* soit un événement *Item*. Dans les deux cas, il y a aura un écouteur.
Si l'écouteur est lié à l'événement de type *Action*, il devra implémenter l'interface *ActionListener* et redéfinir la méthode *actionPerformed*.
Par contre, si l'écouteur est lié à l'événement de type *Item*, il devra implémenter l'interface *ItemListener* et redéfinir la seule méthode *itemStateChanged*.
Il y a dans le cas des boutons radios, une possibilité supplémentaire grâce au type *Item*.
En effet, celui-ci permet de cerner les changements d'états des différents boutons.

8.2.2. Etat d'un bouton radio

On peut connaître l'état d'une case à un moment donné. Il suffit de faire appel à la méthode *isSelected* de la classe *JRadioButton*.
On peut aussi changer à tout moment, sans devoir effectuer une action, imposer à une case un état donné à l'aide de la méthode *setSelected*. Il est à noter qu'un tel appel génère un événement de type *Item*.

8.3. Exemples

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class ControleUsuel2 {
   public static void main(String args[]){
      BoutonRadio br = new BoutonRadio();
      br.setVisible(true);
      br.setDefaultCloseOperation(3);
   }
}
class BoutonRadio extends JFrame implements ActionListener {
   private JRadioButton radio1, radio2, radio3;
   private JButton etat;
   public BoutonRadio() {
```

```
Action bouton 1
Action bouton 2
Action bouton 3
Action bouton 2
ETAT BOUTONS : false true false
Action bouton 1
ETAT BOUTONS : true false false
Action bouton 3
ETAT BOUTONS : false false true
```

```
        setTitle ("Exemple de boutons radios");
        setSize(400,100);
        Container co = getContentPane();
        co.setLayout(new FlowLayout());
        ButtonGroup groupe = new ButtonGroup();
        radio1 = new JRadioButton("Bouton 1");
        groupe.add(radio1);
        co.add(radio1);
        radio1.addActionListener(this);
        radio2 = new JRadioButton("Bouton 2");
        groupe.add(radio2);
        co.add(radio2);
        radio2.addActionListener(this);
        radio3 = new JRadioButton("Bouton 3");
        groupe.add(radio3);
        co.add(radio3);
        radio3.addActionListener(this);
        etat =new JButton("ETAT");
        co.add(etat);
        etat.addActionListener(this);
    }
    public void actionPerformed (ActionEvent ev) {
        Object source = ev.getSource();
        if(source == radio1) System.out.println("Action bouton 1");
        if(source == radio2) System.out.println("Action bouton 2");
        if(source == radio3) System.out.println("Action bouton 3");
        if(source == etat) System.out.println ("ETAT BOUTONS : " + radio1.isSelected() + " " + radio2.isSelected() + " " +
        radio3.isSelected());
    }
}
```

Cet exemple peut être affiné par l'utilisation de la classe *Item*.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class ControleUsuel3 {
    public static void main(String args[]){
    BoutonRadio br = new BoutonRadio();
    br.setVisible(true);
    br.setDefaultCloseOperation(3);
    }
}
class BoutonRadio extends JFrame implements ActionListener, ItemListener {
    private JRadioButton radio1, radio2, radio3;
    private JButton etat;
    public BoutonRadio() {
        setTitle ("Exemple de boutons radios");
        setSize(400,100);
        Container co = getContentPane();
        co.setLayout(new FlowLayout());
        ButtonGroup groupe = new ButtonGroup();
        radio1 = new JRadioButton("Bouton 1");
        groupe.add(radio1);
        co.add(radio1);
        radio1.addActionListener(this);
        radio1.addItemListener(this);
        radio2 = new JRadioButton("Bouton 2");
        groupe.add(radio2);
        co.add(radio2);
        radio2.addActionListener(this);
        radio2.addItemListener(this);
        radio3 = new JRadioButton("Bouton 3");
        groupe.add(radio3);
        co.add(radio3);
        radio3.addActionListener(this);
        radio3.addItemListener(this);
        etat =new JButton("ETAT");
```

```
        co.add(etat);
        etat.addActionListener(this);
    }
    public void actionPerformed (ActionEvent ev) {
        Object source = ev.getSource();
        if(source == radio1) System.out.println("Action bouton 1");
        if(source == radio2) System.out.println("Action bouton 2");
        if(source == radio3) System.out.println("Action bouton 3");
        if(source == etat) System.out.println ("ETAT BOUTONS : "+ radio1.isSelected() + " " + radio2.isSelected() + " " +
        radio3.isSelected());
    }
    public void itemStateChanged(ItemEvent ev) {
        Object source = ev.getSource();
        if(source == radio1) System.out.println("Changement etat bouton 1");
        if(source == radio2) System.out.println("Changement etat bouton 2");
        if(source == radio3) System.out.println("Changement etat bouton 3");
    }
}
```

8.4. **Exercice**

Créer une application qui permette à l'utilisateur de choisir entre plusieurs couleurs (rouge, vert, noir, bleu et blanc), la couleur de son véhicule.
Avant la validation, il faut qu'il coche la case concernant les conditions.
Le bouton de validation ne pourra donc être actif que lorsque l'utilisateur à cocher la case.
Attention, si la case est décochée, le bouton doit être de nouveau inactif.

9. Les étiquettes

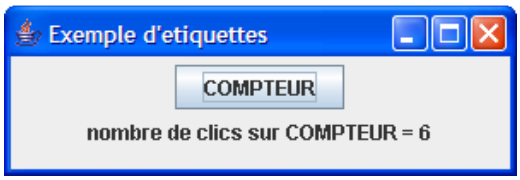
9.1. Généralités

Un composant de type *JLabel* permet d'afficher dans un container un texte non modifiable par l'utilisateur mais qui peut évoluer avec le programme.
Ce type de composant sert à afficher une information ou à identifier un composant qui ne l'est pas.
Ce composant n'a pas de bordure, ni couleur de fond. Par contre, on peut utiliser la méthode *setForeground* qui permet de changer la couleur du texte.
La méthode qui permet de changer le texte au cours du programme est *setText*.

9.2. Exemple

Dans cet exemple, on fera affiché le nombre de clics effectués à l'aide de la souris sur un bouton. Cet affichage ne sera plus dans une fenêtre console mais bien sur la fenêtre graphique.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Etiquette {
   public static void main(String args[]) {
      Labels lb = new Labels();
      lb.setVisible(true);
      lb.setDefaultCloseOperation(3);
   }
}
class Labels extends JFrame implements ActionListener {
   private JLabel compte;
   private JButton bouton;
   private int nbClics;
   public Labels() {
      setTitle ("Exemple d'etiquettes");
      setSize(300,100);
      Container co = getContentPane();
      co.setLayout(new FlowLayout());
      bouton =new JButton("COMPTEUR");
      co.add(bouton);
      bouton.addActionListener(this);
      nbClics=0;
      compte = new JLabel ("nombre de clics sur COMPTEUR = "+ nbClics);
      co.add(compte);
   }
   public void actionPerformed (ActionEvent ev) {
      nbClics++;
      compte.setText("nombre de clics sur COMPTEUR = "+ nbClics);
   }
}
```



9.3. Exercice

Reprendre la calculatrice mais cette fois afficher le résultat sur la fenêtre.

10. Les champs de texte

10.1. Généralités

Un champ de texte est une zone rectangulaire dans laquelle l'utilisateur peut entrer ou modifier un texte. Il s'obtient en utilisant la classe *JTextField*. Son constructeur, doit obligatoirement indiquer la taille donnant le nombre de caractères.
Il n'y a pas de libellé, il faudra donc utiliser *JLabel* pour y placer une étiquette.

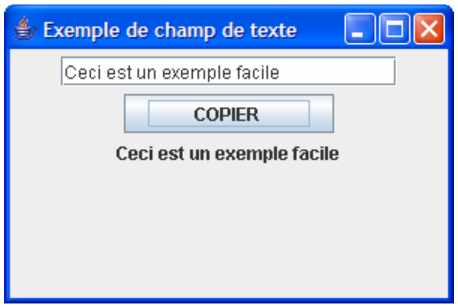
10.2. Exploitation usuelle d'un champ texte

Avec un objet de ce type, on veut connaître à tout moment le contenu de ce champ. La méthode qui fait cela est *getText*.
Le prélèvement peut se faire suite à une action ou à la prise ou perte d'un focus.
Ces deux derniers événements sont liés à la classe *Focus* qui implémente l'interface *FocusListener* avec deux méthodes *focusGained* et *focusLost*.
Par défaut, un champ de texte est modifiable mais on peut lui appliquer une méthode pour le rendre non modifiable, *setEditable (false)*.
On peut aussi modifier la taille d'un champ de texte à l'aide de la méthode *setColums* mais il faudra appliquer au container ou au champ de texte, la méthode *revalidate* ou *validate* pour que cette modification soit prise en compte.

10.3. Exemples

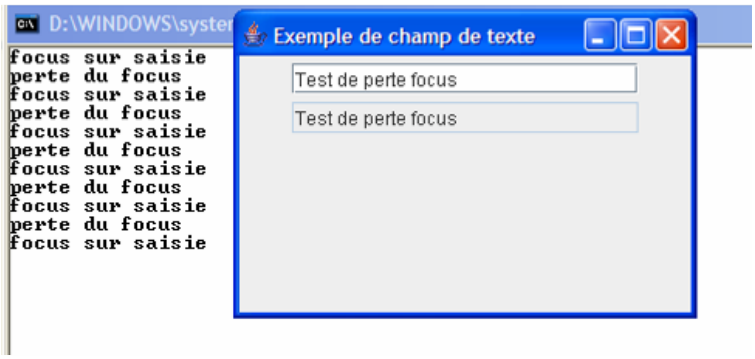
Un premier exemple qui lors de l'action sur le bouton, copie le contenu du champ de texte dans une étiquette.

```
import java.awt.*; import java.awt.event.*; import javax.swing.*;
public class Texte1 {
   public static void main(String args[]){
   Texte tx = new Texte();
   tx.setVisible(true);
   tx.setDefaultCloseOperation(3);
   }
}
class Texte extends JFrame implements ActionListener{
   private JLabel copie;
   private JButton copier;
   private JTextField saisie;
   public Texte() {
      setTitle ("Exemple de champ de texte");
      setSize(300,200);
      Container co = getContentPane();
      co.setLayout(new FlowLayout());
      saisie = new JTextField(20);
      co.add(saisie);
      saisie.addActionListener(this);
      copier = new JButton("         COPIER         ");
      co.add(copier);
      copier.addActionListener(this);
      copie = new JLabel ("");
      co.add(copie); }
   public void actionPerformed (ActionEvent ev) {
      if (ev.getSource() == copier) {
         String text = saisie.getText();
         copie.setText(text);
      }
   }
}
```



Ce second exemple nous montre que l'on peut mettre à jour un champ de texte non modifiable par l'utilisateur suite à la perte du focus d'un champ de texte modifiable.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Texte2 {
   public static void main(String args[]){
   Texte tx = new Texte();
   tx.setVisible(true);
   tx.setDefaultCloseOperation(3);
   }
}
class Texte extends JFrame implements ActionListener, FocusListener{
   private JTextField saisie, copie;
   public Texte() {
      setTitle ("Exemple de champ de texte");
      setSize(300,200);
      Container co = getContentPane();
      co.setLayout(new FlowLayout());
      saisie = new JTextField(20);
      co.add(saisie);
      saisie.addActionListener(this);
      saisie.addFocusListener(this);
      copie = new JTextField(20);
      copie.setEditable(false);
      co.add(copie);
   }
   public void actionPerformed (ActionEvent ev) {
      System.out.println("validation saisie");
      String text = saisie.getText();
      copie.setText(text);
   }
   public void focusLost (FocusEvent ev) {
      System.out.println("perte du focus");
      String text = saisie.getText();
      copie.setText(text);
   }
   public void focusGained (FocusEvent ev) {
      System.out.println("focus sur saisie");
   }
}
```



10.4. **Exploitation fine d'un champ de texte**

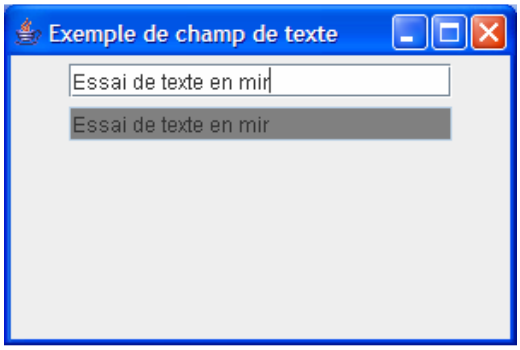
Nous avons lu le contenu d'un champ de texte à un moment précis mais il est possible de connaître le contenu même pendant la frappe au clavier. Pour cela, il faut importer *javax.swing.event* qui permet de gérer les événements de types *Document* qui implémentent l'interface *DocumentListener*. Cette interface possède trois méthodes : *insertUpdate*, *removeUpdate* et *changedUpdate*.
Il faudra donc redéfinir chacune des trois méthodes.

10.5. **Exemple**

Ce dernier exemple montre qu'il est possible d'écrire en miroir.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
public class Texte3 {
   public static void main(String args[]){
   Texte tx = new Texte();
   tx.setVisible(true);
```

```
    tx.setDefaultCloseOperation(3);
    }
}
class Texte extends JFrame implements DocumentListener{
    private JTextField saisie, copie;
    public Texte() {
        setTitle ("Exemple de champ de texte");
        setSize(300,200);
        Container co = getContentPane();
        co.setLayout(new FlowLayout());
        saisie = new JTextField(20);
        co.add(saisie);
        saisie.getDocument().addDocumentListener(this);
        copie = new JTextField(20);
        copie.setEditable(false);
        copie.setBackground(Color.gray);
        co.add(copie);
    }
    public void insertUpdate (DocumentEvent ev) {
        String text = saisie.getText();
        copie.setText(text);
    }
    public void removeUpdate (DocumentEvent ev) {
        String text = saisie.getText();
        copie.setText(text);
    }
    public void changedUpdate (DocumentEvent ev) {}
}
```



10.6. **Exercice**

Créer une application dans laquelle l'utilisateur encode les informations suivantes :
nom, prénom, sexe, âge, nationalité (belge, français, italien, espagnol, autre à préciser).
Une case devra être cochée pour acceptation des conditions de vente (En cas de perte ou déterioration, aucun remboursement sera effectué) du paiement de 275,24 EUR.
Un bouton de paiement est actif. C'est à ce momment que toutes les informations seront retranscrites sur la fenêtre console et dans une nouvelle fenêtre.

10.7. **Les événements liés au clavier**

Il est quelque fois intéressant de connaître les touches utilisées par l'utilisateur afin d'associer des actions. Les événements générés par le clavier appartiennent à la catégorie *KeyEvent* de l'interface *KeyListener*. Pour employer les méthodes de cette interface, elle devra être ajoutée aux autres interfaces implémentées.
Trois méthodes sont à redéfinir : *keyPressed* lorsque l'on appuye sur une touche, *keyReleased* lorsque l'on relâche la touche et *keyTyped* pour les deux actions précédentes, c'est-à-dire à la frappe de la touche.
Ainsi pour le caractère E, plusieurs appels aux méthodes se font :
1°) *keyPressed* : appui sur la touche « Shift » ;
2°) *keyPressed* : appui sur la touche « e » ;
3°) *keyReleased* : relâchement de la touche « e » ;
4°) *keyReleased* : relâchement de la touche « Shift » ;
5°) *keyTyped* : pour le caractère E.
Par contre si on appuie juste sur la touche « Alt » sans aucune autre touche, seuls les méthodes *keyPressed* et *keyReleased* sont appelées.

10.7.1. Identification des touches

L'objet événement de type *keyEvent* contient des informations très utiles.
Pour les obtenir, deux méthodes intéressantes :
– *getKeyChar* qui fournit le caractère concerné sous forme d'une valeur de type *char*
– *getKeyCode* qui fournit un entier (code de touche virtuelle) qui permet d'identifier la touche concernée.

Voici un tableau de constantes correspondant à chacune des touches qu'on peut retrouver sur un clavier :

VK_0 à VK_9	touches 0 à 9	VK_HOME	touche Home
VK_NUMPAD0 à VK_NUMPAD9	touches 0 à 9 (pavé numérique)	VK_INSERT	touche Insert
VK_A à VK_Z	touches A à Z	VK_LEFT	touche flèche gauche
VK_F1 à VK_F24	touches fonctionF1 à F24	VK_NUM_LOCK	touche vérouillage numérique
VK_ALT	touche Aalt	VK_PAGE_DOWN	touche Page suivante
VK_ALT_GRAPH	touche Alt graphique	VK_PAGE_UP	touche Page précédente
VK_CAPS_LOCK	touche vérouillage majuscule	VK_PRINTSCREEN	touche Impression écran
VK_CONTROL	touche Ctrl	VK_RIGHT	touche flèche droite
VK_DELETE	touche Suppr	VK_SCROLL_LOCK	touche arrêt défilement
VK_DOWN	touche flèche bas	VK_SHIFT	touche majuscule temporaire
VK_END	touche Fin	VK_SPACE	touche esapce
VK_ENTER	touche de validation	VK_TAB	touche de tabulation
VK_ESCAPE	touche Echap	VK_UP	touche flèche haut

10.7.2. Exemple

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
public class EcouteClavier {
   public static void main (String []args) {
      MaFenetre fen = new MaFenetre();
      fen.setVisible(true);
      fen.setDefaultCloseOperation(3);
   }
}
class MaFenetre extends JFrame implements KeyListener {
   public MaFenetre() {
      setTitle("Exemple lecture clavier");
      setBounds(100,100,300,150);
      addKeyListener(this);
   }
   public void keyPressed(KeyEvent ev) {
      System.out.println("Touche "+ ev.getKeyCode() + " pressee : " + ev.getKeyText(ev.getKeyCode()));
   }
   public void keyReleased(KeyEvent ev) {
         System.out.println("Touche "+ ev.getKeyCode() + " relachee : " + ev.getKeyText(ev.getKeyCode()));
   }
   public void keyTyped(KeyEvent ev) {
         System.out.println("Caractere frappe "+ ev.getKeyChar() + " de code : " + (int)ev.getKeyChar());
   }
}
```

```
D:\WINDOWS\system32\cmd.exe
Touche 16 pressee : Maj
Touche 69 pressee : E
Caractere frappe E de code : 69
Touche 16 relachee : Maj
Touche 69 relachee : E
Touche 17 pressee : Ctrl
Touche 18 pressee : Alt
Touche 69 pressee : E
Caractere frappe Ç de code : 8364
Touche 69 relachee : E
Touche 17 relachee : Ctrl
Touche 18 relachee : Alt
Touche 154 relachee : Impression d''Úcran
Touche 154 relachee : Impression d''Úcran
Touche 154 relachee : Impression d''Úcran
Touche 155 pressee : Insúrer
Touche 155 relachee : Insúrer
Touche 145 pressee : Verrouillage du dúfilement
Touche 145 relachee : Verrouillage du dúfilement
Touche 154 relachee : Impression d''Úcran
Touche 154 relachee : Impression d''Úcran
```

10.7.3. Etat des touches modificatrices

En Java, il est possible de connaître l'état des touches modificatrices («Shift », « Alt », « Alt Gr » et « Ctrl »). Pour cela, il existe des méthodes qui renvoient une valeur booléenne :
– *isAltDown* : pour la touche « Alt » ;
– *isAltGraphDown* : pour la touche « Alt Gr » ;
– *isControlDown* : pour la touche « Ctrl » ;
– *isShiftDown* : pour la touche « Shift » ;
– *isMetaDown* : pour l'une des quatre.
De plus, la méthode *getModifiers* fournit un entier qui indique l'état de ces touches que l'on peut tester à l'aide de masques binaires définis dans la classe *InputEvent* : *ALT_MASK*, *CONTROL_MASK*, *SHIFT_MASK* et *META_MASK*.
Cet exemple montre que l'on peut changer le fond de couleur d'un panneau à l'aide de l'association des touches : « Ctrl » + « Alt » + « R » (rouge) ou « B » (bleu) ou « J » (jaune).

```
import java.awt.*; import java.awt.event.*;
import javax.swing.*;  import javax.swing.event.*;
public class EcouteClavier2 {
   public static void main (String []args) {
      MaFenetre fen = new MaFenetre();
      fen.setVisible(true);
      fen.setDefaultCloseOperation(3);
   }
}
class MaFenetre extends JFrame implements KeyListener {
   private JPanel pan;
   public MaFenetre() {
      setTitle("Colorations");
      setBounds(100,100,300,150);
      Container co = getContentPane();
      pan = new JPanel();
      co.add(pan);
      addKeyListener(this);
   }
   public void keyPressed(KeyEvent ev) {
      if(ev.isControlDown() && ev.isAltDown()) {
         int touche = ev.getKeyCode();
         switch (touche) {
            case KeyEvent.VK_R : pan.setBackground (Color.red); break;
            case KeyEvent.VK_V : pan.setBackground (Color.green); break;
            case KeyEvent.VK_J : pan.setBackground (Color.yellow); break;
         }
      }
   }
   public void keyReleased(KeyEvent ev) {}
   public void keyTyped(KeyEvent ev) {}
}
```

Pour éviter d'écrire les deux dernières lignes, il est possible de remplacer le code ci-dessus par :

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
public class EcouteClavier3 {
   public static void main (String []args) {
      MaFenetre fen = new MaFenetre();
      fen.setVisible(true);
      fen.setDefaultCloseOperation(3);
   }
}
```

```
class MaFenetre extends JFrame{
   private JPanel pan;
   public MaFenetre() {
      setTitle("Colorations");
      setBounds(100,100,300,150);
      Container co = getContentPane();
      pan = new JPanel();
      co.add(pan);
      addKeyListener(new KeyAdapter() {
         public void keyPressed(KeyEvent ev) {
            if(ev.isControlDown() && ev.isAltDown()) {
               int touche = ev.getKeyCode();
               switch (touche) {
                  case KeyEvent.VK_R : pan.setBackground (Color.red); break;
                  case KeyEvent.VK_V : pan.setBackground (Color.green); break;
                  case KeyEvent.VK_J : pan.setBackground (Color.yellow); break;
               }
            }
         }
      });
   }
}
```

10.8. **Exercice**

Créer un jeu de pendu avec 26 boutons correspondant aux 26 lettres de l'alphabet.
Un bouton de « proposer » un mot se trouvera aussi sur l'interface, si le mot introduit n'est pas correct, une erreur sera comptabilisée.
Dans le panneau, voisin en fonction des erreurs, on verra apparaître au fur et à mesure, l'échafaud, la tête, le corps, un bras, un second, une jambe, une seconde.
A cette dernière, la pendaison est effective et le jeu s'arrête.
Le premier utilisateur écrira un mot qui s'affichera en mot de passe.

11. Les boîtes de liste

11.1. Généralités

La boîte de liste est un composant qui permet de choisir une ou plusieurs valeurs dans une liste prédéfinie.
On crée une boîte de liste en fournissant un tableau de chaînes à son constructeur comme suit :

```
String couleurs = {"rouge","bleu","gris","vert","jaune","noir"} ;
JList liste = new JList (couleurs) ;
```

Initialement aucune valeur n'est sélectionnée mais on peut forcer un choix préalable à l'aide de la méthode *setSelectedIndex*.
Il existe trois sortes de boîtes de liste :

- ✓ SINGLE_SELECTION (sélection d'une seule valeur)
- ✓ SINGLE_INTERVAL_SELECTION (Sélection d'une seule plage de valeurs contiguës)
- ✓ MULTIPLE_INTERVAL_SELECTION (sélection d'un nombre quelconque de valeurs)

Par défaut, c'est cette troisième sorte qui est prise en considération.
Pour changer de sorte, il faut utiliser la méthode *setSelectionMode*.
Par défaut une boîte de liste ne possède pas de barre de défilement. On peut lui en ajouter une en introduisant la liste dans un panneau de défilement de type *JScrollPane*.
Il sera alors possible de définir le nombre de lignes visibles à l'aide de la méthode *setVisibleRowCount*. Par défaut, le nombre de lignes visibles est de 8.

11.2. Exploitation d'une boîte de liste

11.2.1. Accès aux informations sélectionnées

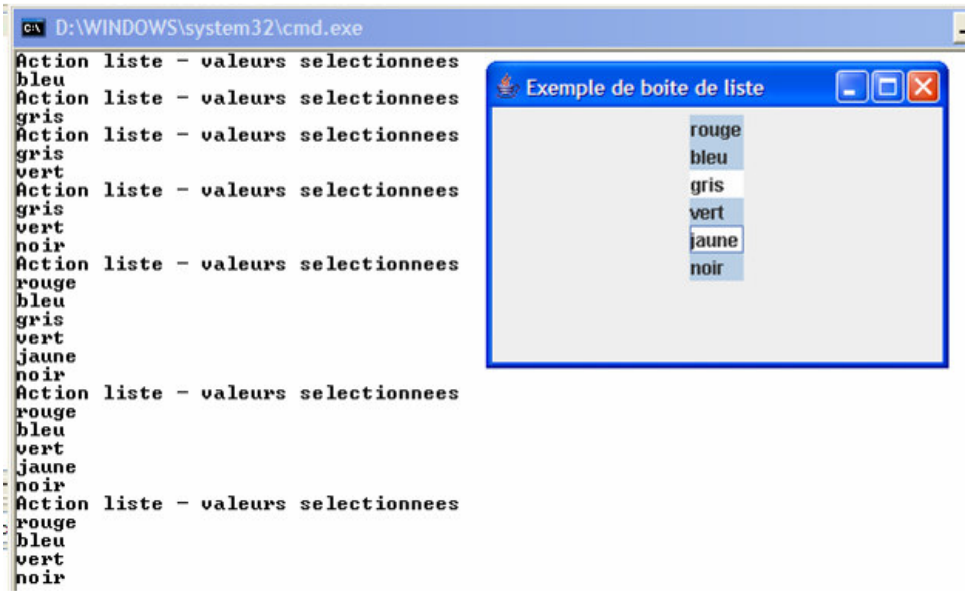
Si c'est une boîte à sélection simple, il suffit d'utiliser la méthode *getSelectedValue*. Cette méthode renvoie comme résultat un *Object* qu'il faut convertir de manière implicite en *String*.
Par contre, si c'est l'une des deux autres boîtes, il faudra utiliser la méthode *getSelectedValues* qui fournit un tableau d'*Object* qu'il faudra convertir implicitement.
Il est aussi possible de connaître la position des sélections dans la liste à l'aide des méthodes *getSelectedIndex* et *getSelectedIndices*.

11.2.2. Evénements générés par les boîtes de liste

Ce type de composant ne génère pas d'événement de type *Action*.
Dans certains cas, on pourra se contenter d'accéder à l'information sélectionnée sur une action externe à la boîte de liste (fermeture d'une boîte de dialogue, bouton servant à valider la sélection).
Dans d'autres cas, il faudra intercepter les événements générés par la liste elle-même. Ces événements font partie d'une classe *ListSelection* qui implémente l'interface *ListSelectionListener*. Cette interface possède une méthode *valueChanged*.
Ces événements sont générés plus souvent qu'on le souhaiterait. Pour palier à une redondance liée lors de l'appui sur le bouton de la souris et au relâchement de celui-ci, il faut utiliser une méthode *getValueIsAdjusting* pour savoir si on est ou pas en phase de transition.

11.3. Exemple

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
public class Boite1 {
   public static void main(String args[]){
   Boite bt = new Boite();
   bt.setVisible(true);
   bt.setDefaultCloseOperation(3);
   }
}
class Boite extends JFrame implements ListSelectionListener {
   private String [] couleurs = {"rouge" , "bleu" , gris" , "vert" , "jaune" , "noir"};
   private JList liste;
   public Boite() {
      setTitle ("Exemple de boite de liste");
      setSize(300,200);
      Container co = getContentPane();
      co.setLayout(new FlowLayout());
      liste = new JList(couleurs);
      co.add(liste);
      liste.addListSelectionListener(this);
   }
   public void valueChanged (ListSelectionEvent ev) {
      if (!ev.getValueIsAdjusting()) {
         System.out.println("Action liste - valeurs selectionnees");
         Object[] valeurs = liste.getSelectedValues();
         for (int i = 0 ; i < valeurs.length; i++)
            System.out.println((String) valeurs[i]);
      }
   }
}
```



11.4. Exercice

Reprenez 7.4 à la page 57 et faîtes en sorte que l'utilisateur choisisse parmi une liste de voiture (ALFA ROMEO, AUDI, BMW, CITROEN, FIAT, MERCEDES, PEUGEOT, RENAULT, VW). Cette fois une image de la voiture apparaîtra sur un panneau voisin.

12. Les boîtes combinées

12.1. Généralités

La boîte de liste combinée ou boîte combinée ou encore boîte combo associe un champ de texte et une boîte de liste simple. Tant que l'utilisateur n'a pas effectué de sélection, seul le champ de texte s'affiche. Par contre, dès l'instant où l'utilisateur sélectionne le champ de texte la liste s'affiche. L'utilisateur peut choisir une valeur qui s'affiche dans le champ de texte. Après validation ou perte du focus, la liste disparaît et le champ de texte s'affiche seul avec la valeur choisie.

12.2. Construction

On construit une boîte combo comme une boîte de liste mais la classe utilisée est *JComboBox*.
On peut la rendre éditable à l'aide de la méthode *setEditable*.
Elle est dotée par défaut d'un ascenseur dès que le nombre de lignes est supérieur à 8.
On peut modifier cette valeur par défaut à l'aide de la méthode *setMaximumRowCount*.
Il est aussi possible de forcer la sélection d'un élément en particulier avec la méthode *setSelectedIndex*.

12.3. Exploitation d'un boîte combo

12.3.1. Accès à l'information sélectionnée ou saisie

La méthode *getSelectedItem* fournit la valeur sélectionnée qu'il faut convertir implicitement en *String* car elle renvoie une valeur de type *Object*.
La méthode *getSelectedIndex* renvoie le rang de la valeur sélectionnée. Cette méthode a un grand intérêt lorsque la liste combo est modifiable car cette méthode fournit la valeur −1 dans le cas où l'utilisateur a introduit une information.

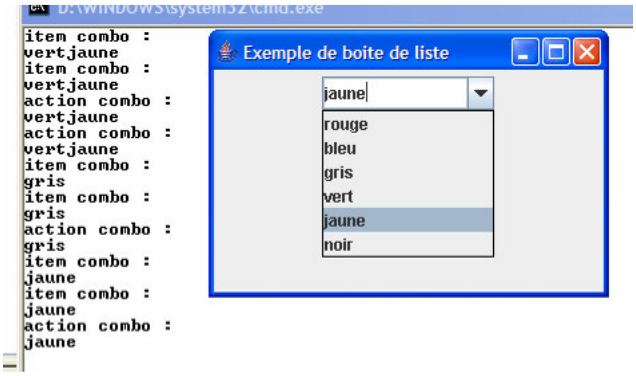
12.3.2. Les événements générés par une boîte combo

On peut seulement se contenter d'accéder à l'information sélectionnée sur une action externe, mais souvent on cherche à intercepter les événements générés par la boîte. Ce type de boîte génère des événements de type *Action* lors d'une sélection ou lors de la validation du champ de texte.
Elle génère aussi des événements de type *Item* qui implémente l'interface *ItemListener* ne comportant qu'une seule méthode *itemStateChanged*.
Comme pour la boîte de liste on obtient toujours deux événements (suppression d'une sélection et nouvelle sélection) mais dans ce cas il n'existe pas de méthode éliminant cette redondance.

12.4. Exemple

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
public class Boite2 {
   public static void main(String args[]){
      Boite bt = new Boite();
      bt.setVisible(true);
      bt.setDefaultCloseOperation(3);
   }
}
```

```
class Boite extends JFrame implements ActionListener, ItemListener {
   private String [] couleurs = {"rouge" , "bleu" , "gris" , "vert" , "jaune" , "noir"};
   private JComboBox combo;
   public Boite() {
      setTitle ("Exemple de boite de liste");
      setSize(300,200);
      Container co = getContentPane();
      co.setLayout(new FlowLayout());
      combo = new JComboBox(couleurs);
      combo.setEditable (true);
      combo.setSelectedIndex(4);
      co.add(combo);
      combo.addActionListener(this);
      combo.addItemListener(this);
   }
   public void actionPerformed (ActionEvent ev) {
      System.out.println("action combo : ");
      Object valeur = combo.getSelectedItem();
      System.out.println((String) valeur);
   }
   public void itemStateChanged (ItemEvent ev) {
      System.out.println("item combo : ");
      Object valeur = combo.getSelectedItem();
      System.out.println((String) valeur);
   }
}
```



12.5. **Evolution dynamique de la liste d'une boîte combo**

Il n'est pas aisé de faire évoluer le contenu d'une boîte de liste par contre il existe des méthodes appropriées pour la modification d'une boîte combo.

- ✓ La méthode *addItem* permet d'ajouter une nouvelle valeur à la fin de la liste.
- ✓ La méthode *addItemAt* permet d'insérer une nouvelle valeur à un rang donné.
- ✓ La méthode *removeItem* permet de supprimer une valeur existante.

12.6. **Exemple**

Même exemple que le précédent avec comme différence que toute valeur saisie par l'utilisateur est ajoutée à la liste. On utilise la méthode *getSelectedIndex*. Le programme ne teste pas si la valeur est déjà présente dans la liste. Si tel en devait être le cas, la valeur sera présente deux fois.


```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
public class Boite3 {
   public static void main(String args[]){
      Boite bt = new Boite();
      bt.setVisible(true);
      bt.setDefaultCloseOperation(3);
   }
}
class Boite extends JFrame implements ActionListener {
   private String [] couleurs = {"rouge" , "bleu" , "gris" , "vert" , "jaune" , "noir"};
   private JComboBox combo;
   public Boite() {
      setTitle ("Exemple de boite de liste");
      setSize(300,200);
      Container co = getContentPane();
      co.setLayout(new FlowLayout());
      combo = new JComboBox(couleurs);
```

```
        combo.setEditable (true);
        combo.setMaximumRowCount(6);
        co.add(combo);
        combo.addActionListener(this);
    }
    public void actionPerformed (ActionEvent ev) {
        System.out.println("action combo : ");
        Object valeur = combo.getSelectedItem();
        int rang = combo.getSelectedIndex();
        if (rang == -1) {
            System.out.println("saisie nouvelle valeur : " + valeur);
            combo.addItem(valeur);
        }
        else
        System.out.println("selection valeur existante : " + valeur + " de rang : " + rang);
    }
}
```

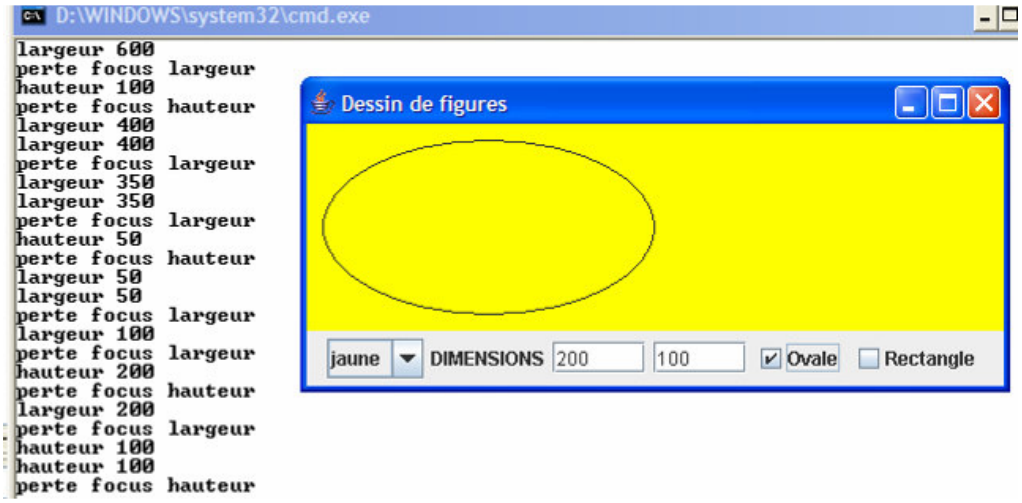
12.7. **Application**

Cette application doit permettre à l'utilisateur de choisir les formes qu'il souhaite dessiner dans une fenêtre, leurs dimensions et la couleur de fond.
Pour limiter le code, l'utilisateur n'aura le choix que de l'ovale et du rectangle.
Les dimensions seront saisies dans deux champs de texte et sont communes aux deux formes.
La couleur de fond sera choisie dans une boîte combo limitée à 4 couleurs.
Les valeurs introduites pour les dimensions sont de type *String*. Pour les convertir en entier, il faut utiliser la méthode *Integer.parseInt*.
De même pour les couleurs, les valeurs sont de types *String*. Pour les convertir en type *Color*, on utilise deux tableaux en parallèle.
La communication entre l'objet fenêtre et l'objet panneau de dessin se fait à l'aide de méthodes de modification *setHauteur*, *setLargeur*, *setOvale* et *setRectangle*.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
public class Boite4 {
    public static void main(String args[]){
        Boite bt = new Boite();
        bt.setVisible(true);
        bt.setDefaultCloseOperation(3);
    }
}
class Boite extends JFrame implements ActionListener, ItemListener, FocusListener {
    static public final String [] couleurs = {"rouge" , "vert" , "jaune" , "bleu"};
    static public final Color [] colors = {Color.red, Color.green, Color.yellow, Color.blue};
    private PaneauDessin panDes;
    private JPanel panCom;
    private JComboBox comboCoulFond;
    private JTextField txtLargeur, txtHauteur;
    private JCheckBox cOvale, cRectangle;
    public Boite() {
        setTitle ("Dessin de figures");
        setSize(450,200);
        Container co = getContentPane();
        panDes = new PaneauDessin();
        co.add(panDes);
        panDes.setBackground(Color.cyan);
        panCom = new JPanel();
        co.add(panCom, "South");
        comboCoulFond = new JComboBox(couleurs);
```

```
        panCom.add(comboCoulFond);
        comboCoulFond.addItemListener(this);
        JLabel dim = new JLabel("DIMENSIONS");
        panCom.add(dim);
        txtLargeur = new JTextField ("50",5);
        txtLargeur.addActionListener(this);
        txtLargeur.addFocusListener(this);
        panCom.add(txtLargeur);
        txtHauteur = new JTextField ("20",5);
        txtHauteur.addActionListener(this);
        txtHauteur.addFocusListener(this);
        panCom.add(txtHauteur);
        cOvale = new JCheckBox("Ovale");
        panCom.add(cOvale);
        cOvale.addActionListener(this);
        cRectangle = new JCheckBox("Rectangle");
        panCom.add(cRectangle);
        cRectangle.addActionListener(this);
    }
    public void actionPerformed (ActionEvent ev) {
        if (ev.getSource() == txtLargeur) setLargeur();
        if (ev.getSource() == txtHauteur) setHauteur();
        if (ev.getSource() == cOvale) panDes.setOvale(cOvale.isSelected());
        if (ev.getSource() == cRectangle) panDes.setRectangle(cRectangle.isSelected());
        panDes.repaint();
    }
    public void focusLost (FocusEvent ev) {
        if (ev.getSource() == txtLargeur) {
            setLargeur();
            System.out.println("perte focus largeur");
            panDes.repaint();
        }
        if (ev.getSource() == txtHauteur) {
            setHauteur();
            System.out.println("perte focus hauteur");
            panDes.repaint();
        }
    }
    public void focusGained (FocusEvent ev) {}
    public void itemStateChanged (ItemEvent ev) {
        String couleur = (String)comboCoulFond.getSelectedItem();
        panDes.setCouleur(couleur);
    }
    private void setLargeur() {
        String ch = txtLargeur.getText();
        System.out.println("largeur " + ch);
        panDes.setLargeur (Integer.parseInt(ch));
    }
    private void setHauteur() {
        String ch = txtHauteur.getText();
        System.out.println("hauteur " + ch);
        panDes.setHauteur (Integer.parseInt(ch));
    }
    class PaneauDessin extends JPanel {
        private boolean rectangle = false, ovale = false;
        private int largeur = 50, hauteur = 50;
        public void paintComponent (Graphics g) {
            super.paintComponent(g);
            if (ovale) g.drawOval (10,10,10 + largeur, 10 + hauteur);
            if (rectangle) g.drawRect (10,10,10 + largeur, 10 + hauteur);
        }
        public void setRectangle (boolean b) {
            rectangle = b;
        }
        public void setOvale (boolean b) {
            ovale = b;
        }
```

```
    public void setLargeur (int l) {
        largeur = l;
    }
    public void setHauteur (int h) {
        hauteur = h;
    }
    public void setCouleur (String c) {
        for (int i = 0 ; i<Boite.couleurs.length ; i++)
            if(c == Boite.couleurs[i]) setBackground (Boite.colors[i]);
    }
  }
}
```



12.8. **Exercice**

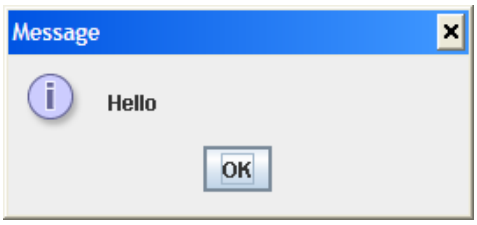
Créer une application qui en fonction des choix de l'utilisateur (couleur du panneau de représentation : jaune, vert, bleu, noir, rouge et orange, de la couleur du trait, de la dimension, de la forme : cercle, rectangle) dessine celui-ci dans un panneau.
Les choix de l'utilisateur se font dans un autre panneau dans lequel, il y aura des informations supplémentaires après la validation à l'aide d'un bouton.
On affichera l'aire formée par les figures ainsi que leur circonférence ou leur périmètre.
On considère 1 pixel = 1 mm.

13. Les boîtes de message

Java dispose de méthodes standard permettant de fournir à l'utilisateur un message qui reste affiché tant que le bouton "OK" n'est pas actionné. Cette classe le permettant s'appelle *JOptionPane* et la méthode faisant cela est *showMessageDialog*.
Nous allons voir la boîte usuelle et quelques variantes.

13.1. La boîte de message usuelle

Si l'on suppose *mes* de type *JFrame*, l'appel est : JOptionPane.showMessageDialog(mes,"Hello") ;
Le message qui apparaît est le suivant :



Si l'on place, "null" au lieu de *mes* qui est l'objet lié à cette méthode alors une boîte de dialogue apparaît indépendamment de toute fenêtre.

13.2. Autres possibilités

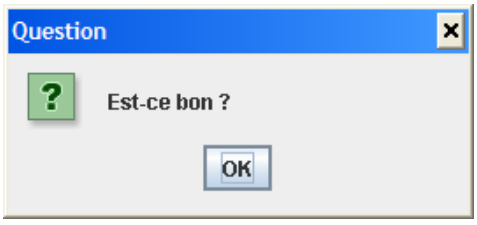
Comme on peut le voir dans l'exemple précédent, cette boîte de dialogue message possède un titre fixe (message), une icône "i" qui indique information. Il existe une variante de la méthode *showMessageDialog* qui permet de choisir :

- ✓ le contenu du message
- ✓ le titre de la boîte
- ✓ le type d'icône : erreur : *JOptionPane.ERROR_MESSAGE*
  information : *JOptionPane.INFORMATION_MESSAGE*
  avertissement : *JOptionPane.WARNING_MESSAGE*
  question : *JOptionPane.QUESTION_MESSAGE*
  aucune icône : *JOptionPane.PLAIN_MESSAGE*

Pour cela, il faut ajouter comme paramètres le titre et ensuite le type d'icône.

```
JOptionPane.showMessageDialog(mes,"Hello","Message d'avertissement",JOptionPane.ERROR_MESSAGE) ;
```


```
JOptionPane.showMessageDialog(mes,"Est-ce bon ?","Question",JOptionPane.QUESTION_MESSAGE);
```



14. Les boîtes de confirmation

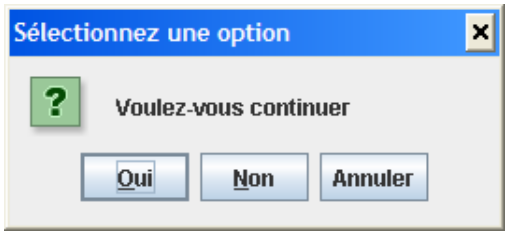
Java permet d'afficher des boîtes dites de confirmation offrant à l'utilisateur un choix de type Oui/Non. Pour cela, il faut utiliser une méthode *showConfirmDialog* de la classe *JOptionPane*.
Nous allons voir la boîte usuelle et quelques variantes.

14.1. La boîte de confirmation usuelle

Si l'on suppose *mes* de type *JFrame*, l'appel est :

```
JOptionPane.showConfirmDialog(mes,"Voulez-vous continuer") ;
```

Le message qui apparaît est le suivant :



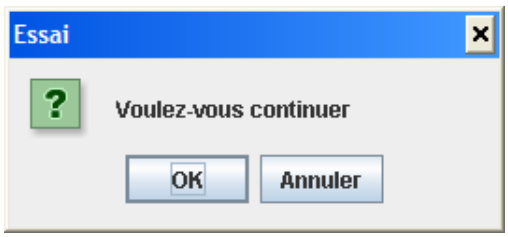
De nouveau, on peut écrire "null" pour le premier paramètre, ce qui fait que cette boîte de message est indépendante de toute fenêtre.
La valeur de retour est un entier selon l'endroit cliquer sur cette boîte.
0 pour le bouton "Oui" (YES_OPTION), 1 pour le bouton "Non" (NO_OPTION), 2 pour le bouton "Annuler" (CANCEL_OPTION) et −1 pour la croix de fermeture (CLOSED_OPTION).

14.2. Variantes

Il existe une variante qui permet de choisir le titre, ainsi que les boutons apparaissant dans cette boîte qui sont aussi associer à des valeurs entières.

- ✓ *JOptionoPane.DEFAULT_OPTION* est la boîte usuelle
- ✓ *JOptionPane.YES_NO_OPTION* est une boîte ne possédant que les boutons "OUI" - "NON"
- ✓ *JOptionPane.OK_CANCEL_OPTION* est une boîte avec "OK" et "ANNULER"

```
JOptionPane.showConfirmDialog(mes,"Voulez-vous continuer","Essai",JOptionPane.OK_CANCEL_OPTION) ;
```



Les valeurs de retour sont dans les différents cas :
0 pour les boutons "OK" et "Oui", 1 pour le bouton "Non", 2 pour le bouton "Annuler" et −1 pour la fermeture de la boîte.

15. Les boîtes de saisie

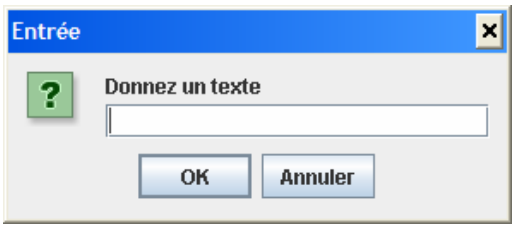
La boîte de saisie permet à l'utilisateur de fournir une information sous la forme d'une chaîne de caractères. La méthode est *showInputDialog* de la classe *JOptionPane*.
Il existe aussi plusieurs variantes, nous verrons l'usuelle et ensuite les autres.

15.1. La boîte de saisie usuelle

Si l'on suppose *mes* de type *JFrame*, l'appel est :

```
JOptionPane.showConfirmDialog(mes,"Donnez un texte") ;
```

Le message qui apparaît est le suivant :

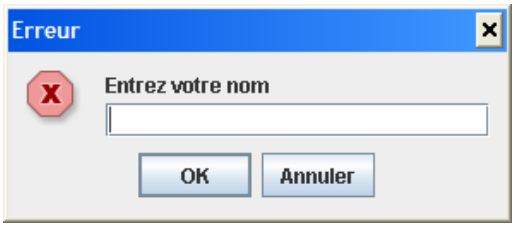


Le résultat en retour est un objet de type *String* ou la valeur *Null* si l'utilisateur a fermé la boîte ou appuyé sur `"Annuler"`.

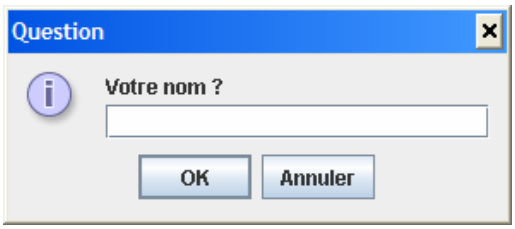
15.2. Variantes

Il est aussi possible d'ajouter un titre et de changer l'icône de cette boîte de saisie comme pour la boîte de message.
Les paramètres sont identiques.

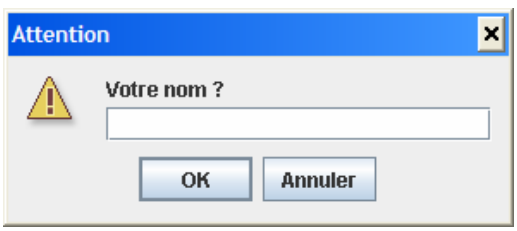
```
JOptionPane.showInputDialog(mes,"Donnez votre nom","Erreur",JOptionPane.ERROR_MESSAGE) ;
```




```
JOptionPane.showInputDialog(mes,"Votre nom ?","Question",JOptionPane.INFORMATION_MESSAGE);
```




```
JOptionPane.showInputDialog(mes,"Votre nom ?","Attention",JOptionPane.WARNING_MESSAGE);
```

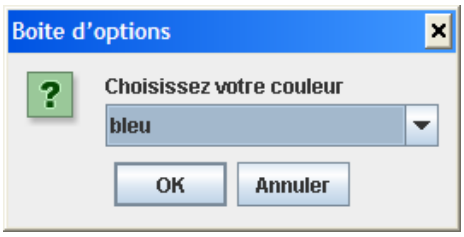


16. Les boîtes d'options

Java permet par l'intermédiaire d'une boîte combo d'afficher une boîte d'options.
Reprenons l'exemple du choix des couleurs.
Si l'on suppose *mes* de type *JFrame*, l'appel est :

```
JOptionPane.showInputDialog (mes, "Choisissez votre couleur","Boite d'options",JOptionPane.QUESTION_MESSAGE, null, couleurs, couleurs[2]) ;
```

Le troisième paramètre permet d'écrire un titre, le quatrième de choisir le type de boîte de message, le cinquième d'ajouter une icône supplémentaire, le sixième le nom du tableau et le dernier paramètre le rang par défaut qui apparaîtra à l'écran.

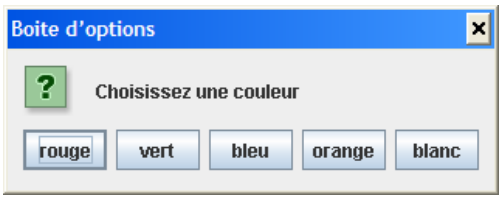


Dans notre cas *couleurs* est un tableau de type *String* initialisé comme suit :
rouge, vert, bleu, jaune, orange et blanc.
La valeur de retour est soit un *Object* sinon *null* si clic sur `"Annuler"` ou sur la croix de fermeture.

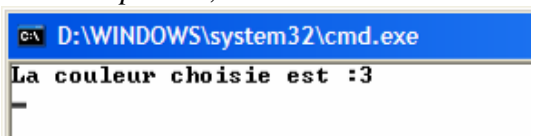
Il existe aussi une méthode nommée *showOptionDialog* qui permet d'afficher les différents sous forme de boutons.
La valeur de retour est un entier correspondant au rang de l'option choisie.

```
int rang = JOptionPane.showOptionDialog(this,"Choisissez une couleur","Boite d'options",JOptionPane.DEFAULT_OPTION, JOptionPane.QUESTION_MESSAGE, null, couleurs, couleurs[0]) ;
```

L'instruction précédente donne à l'écran :



A l'aide de l'instruction *System.out.println*, le résultat est



Exercice

Reprenez un exercice au choix en y incorporant une ou plusieurs types de boîte.

17. Les boîtes de dialogue personalisées

Nous avons utilisé des méthodes pour créer des boîtes de dialogues standard mais Java dispose d'une classe *JDialog* qui permet de créer des boîtes de dialogue personalisées.

17.1. Conctruction et affichage d'une boîte de dialogue

17.1.1. Construction

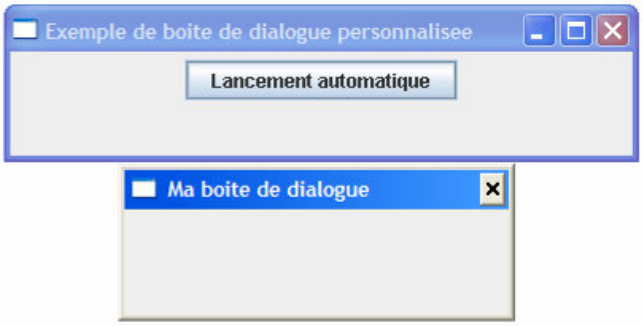
La façon la plus usuelle de construire un objet boîte de dialogue est la suivante :

```
JDialog bd = new JDialog(mes, "Ma boite de dialogue", true) ;
```

Le premier argument désigne la fenêtre parent, le second donne le texte qui apparaît dans la barre de titre et le troisième précise si la fenêtre est modale ou non. Si la boîte est modale, cela permet de ne pas agir sur d'autres composants tant que la boîte de dialogue est active.

17.1.2. Affichage

Comme pour une fenêtre, il est nécessaire d'attribuer une taille par la méthode *setSize* et de provoquer l'affichage à l'aide de *setVisible*.



17.1.3. Exemple

```
import java.awt.*; import java.awt.event.*; import javax.swing.*;
public class Dialogue1 {
   public static void main(String args[]){
      Message mes = new Message();
      mes.setVisible(true);
      mes.setDefaultCloseOperation(3);
   }
}
class Message extends JFrame implements ActionListener {
   public Message() {
      setTitle ("Exemple de boite de dialogue personnalisee");
      setSize(400,100);
      Container co = getContentPane();
      co.setLayout(new FlowLayout());
      JButton lance = new JButton("Lancement automatique");
      co.add(lance);
      lance.addActionListener(this);
   }
   public void actionPerformed (ActionEvent ev) {
      JDialog bd = new JDialog(this,"Ma boite de dialogue",true);
      bd.setSize(250,100);
      bd.setVisible(true);
   }
}
```

17.1.4. Application d'une classe dérivée de *JDialog*

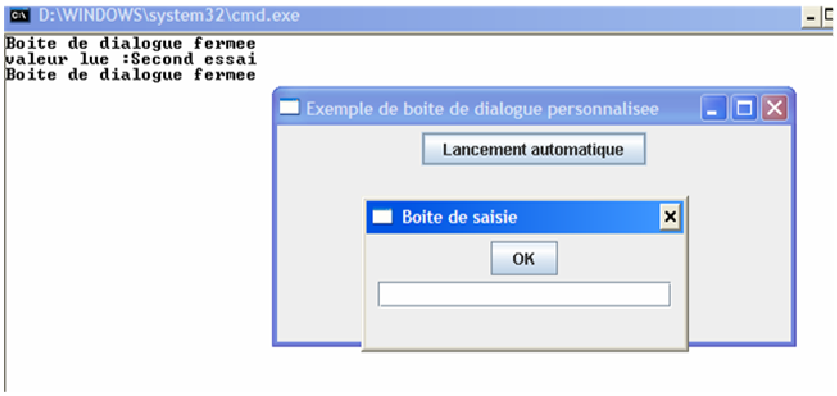
En pratique, on crée une classe qui est une classe dérivée, comme l'est la classe dérivée de *JFrame*, pour que l'on puisse y incorporer des boutons, des champs de texte, … et finalement transmettre des résultats à la fenêtre de qui elle dépend.
Ainsi tout ce qui a été dit pour les fenêtres est valable pour ces boîtes de dialogues.

17.2. Exemple d'utilisation d'une boîte de dialogue

L'introduction des éléments se fait par le biais d'un container pour lequel les composants sont ajoutés.
Il faut gérer la fermeture de la boîte de dialogue et cela se fait par le biais d'un bouton "OK"

qui fermera cette boîte en utilisant la méthode *setVisible*. Par ce biais, l'élément est caché mais toujours en mémoire, il faut donc utiliser une méthode *dispose* qui libère la boîte de dialogue et tous les objets liés.
Pour récupérer les données, il faut prévoir une méthode qui renvoie un élément de type *String*. Voici un exemple qui permet d'ouvrir une boîte de dialogue dans laquelle, un champ de texte est présent ainsi qu'un bouton "OK". Lors de la fermeture, il faut écrire le contenu du champ de texte à la fenêtre console, par exemple.

```
import java.awt.*; import java.awt.event.*; import javax.swing.*;
public class Dialogue2 {
   public static void main(String args[]){
      Message mes = new Message();
      mes.setVisible(true);
      mes.setDefaultCloseOperation(3);
   }
}
class Message extends JFrame implements ActionListener {
   private JButton lance;
   private String texte;
   public Message() {
      setTitle ("Exemple de boite de dialogue personnalisee");
      setSize(400,200);
      Container co = getContentPane();
      co.setLayout(new FlowLayout());
      lance = new JButton("Lancement automatique");
      co.add(lance);
      lance.addActionListener(this);
   }
   public void actionPerformed (ActionEvent ev) {
      MonDialog bd = new MonDialog(this);
      texte = bd.lanceDialogue();
      if(texte != null) System.out.println("valeur lue :" + texte);
         else System.out.println("Boite de dialogue fermee");
      bd.dispose();
   }
}
class MonDialog extends JDialog implements ActionListener {
   private boolean ok;
   private JButton okBouton;
   private JTextField champTexte;
   public MonDialog(JFrame fen) {
      super (fen,"Boite de saisie",true);
      setSize(250,120);
      okBouton = new JButton("OK");
      okBouton.addActionListener(this);
      champTexte = new JTextField(20);
      Container cont = getContentPane();
      cont.setLayout(new FlowLayout());
      cont.add(okBouton);
      cont.add(champTexte);
   }
   public void actionPerformed(ActionEvent ev) {
      if(ev.getSource() == okBouton) {
         ok = true;
         setVisible(false);
      }
   }
   public String lanceDialogue() {
      ok = false;
      setVisible(true);
      if (ok) return champTexte.getText();
         else return null;
   }
}
```

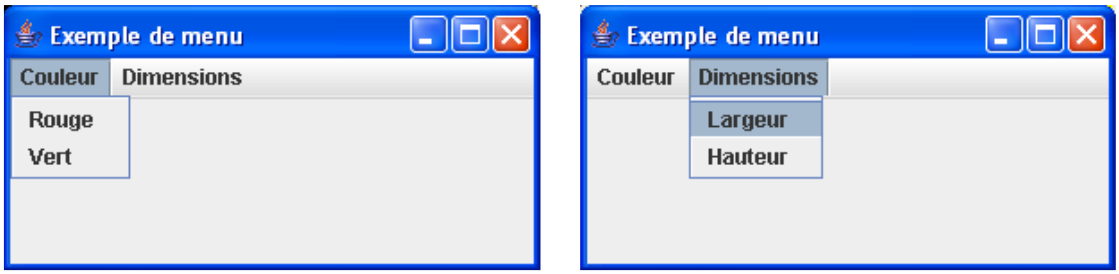


18. Les menus

Les menus dans une fenêtre graphique permettent d'accéder à des commandes à tout moment sans que celle-ci soit présente par une multitude de boutons. Java dispose de deux possibilités, la création d'une barre de menu et la création d'un menu contextuel ou surgissant.
Il est possible d'insérer des raccourcis claviers comme toute application commerciale.

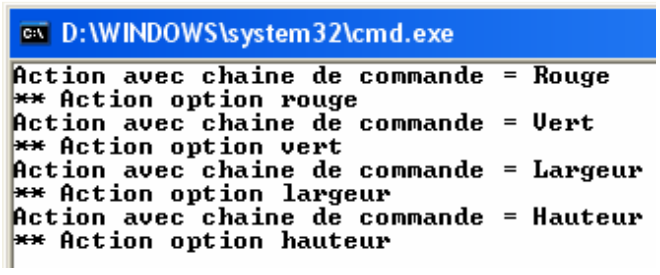
18.1. Menus déroulants

Pour créer un menu déroulant attaché à la barre menu, il faut utiliser trois classes : *JMenurBar, JMenu, JMenuItem*. La première classe sert pour un objet barre de menus, la seconde pour les menus visibles dans la barre et la troisième, pour chaque menu, les différentes options. Nous allons créer cette situation suivante :



La méthode *setMenuBar(Component c)* permet d'ajouter la barre de menu à la fenêtre. La méthode *add(Component c)* instanciée à un élément permet d'ajouter à cet élément le composant voulu.
Pour un élément de type *JMenuBar*, il n'y a pas besoin de paramètre alors que pour *JMenu* et *JMenuItem* le paramètre est le texte qui apparaît au menu.
Pour qu'une action se fasse, il faut ajouter un écouteur aux différents éléments qui doivent produire quelque chose.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
public class Menu1 {
   public static void main (String []args) {
      FenMenu fen = new FenMenu();
      fen.setVisible(true);
      fen.setDefaultCloseOperation(3);
   }
}
class FenMenu extends JFrame implements ActionListener {
   private JMenuBar barreMenus;
   private JMenu couleur, dimensions;
   private JMenuItem rouge, vert, largeur, hauteur;
   public FenMenu() {
      setTitle("Exemple de menu");
      setBounds(100,100,300,150);
      barreMenus = new JMenuBar();
      couleur = new JMenu("Couleur");
      rouge = new JMenuItem("Rouge");
      vert = new JMenuItem("Vert");
      dimensions = new JMenu("Dimensions");
      largeur = new JMenuItem("Largeur");
      hauteur = new JMenuItem("Hauteur");
      setJMenuBar(barreMenus);
      barreMenus.add(couleur);
```



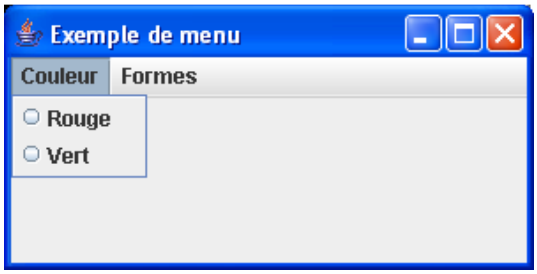
```
        couleur.add(rouge);
        couleur.add(vert);
        barreMenus.add(dimensions);
        dimensions.add(largeur);
        dimensions.add(hauteur);
        rouge.addActionListener(this);
        vert.addActionListener(this);
        largeur.addActionListener(this);
        hauteur.addActionListener(this);
    }
    public void actionPerformed(ActionEvent ev) {
        System.out.println("Action avec chaine de commande = "+ev.getActionCommand());
        if (ev.getSource() == rouge) System.out.println("** Action option rouge");
            else if (ev.getSource() == vert) System.out.println("** Action option vert");
                else if (ev.getSource() == largeur) System.out.println("** Action option largeur");
                    else System.out.println("** Action option hauteur");
    }
}
```

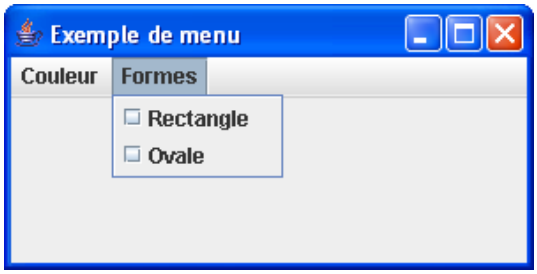
Il est possible d'ajouter une barre séparatrice entre deux options avec la méthode *addSeparator* instanciée au composant du type *JMenu* voulu : dimensions.addSeparator() ;
Nous avons utilisé l'interface *ActionListener* mais il existe une interface qui écoute les éléments d'un menu, elle s'appelle *MenuListener*. Elle possède trois méthodes : *menuSelected*, *menuDeselected* et *menuCanceled*.
Il est aussi possible d'ajouter une icône à coté du texte à l'aide du constructeur de *JMenuItem*. Elle doit être sous le format *.gif*.

18.2. Les différentes sortes d'options

Il est possible de placer les boutons radio ou des cases à cocher au lieu des menus habituels. Les boutons peuvent être placés dans un groupe pour garantir l'unicité à l'intérieur du groupe.

```
import java.awt.*; import java.awt.event.*; import javax.swing.*; import javax.swing.event.*;
public class Menu2 {
    public static void main (String []args) {
        FenMenu fen = new FenMenu();
        fen.setVisible(true);
        fen.setDefaultCloseOperation(3);
    }
}
class FenMenu extends JFrame implements
ActionListener,ItemListener {
    private JMenuBar barreMenus;
    private JMenu couleur, formes;
    private JRadioButtonMenuItem rouge, vert;
    private JCheckBoxMenuItem rectangle, ovale;
    public FenMenu() {
        setTitle("Exemple de menu");
        setBounds(100,100,300,150);
        barreMenus = new JMenuBar();
        couleur = new JMenu("Couleur");
        rouge = new JRadioButtonMenuItem("Rouge");
        vert = new JRadioButtonMenuItem("Vert");
        formes = new JMenu("Formes");
        rectangle = new JCheckBoxMenuItem("Rectangle");
        ovale = new JCheckBoxMenuItem("Ovale");
        setJMenuBar(barreMenus);
        barreMenus.add(couleur);
        couleur.add(rouge);
        couleur.add(vert);
        barreMenus.add(formes);
        formes.add(rectangle);
        formes.add(ovale);
```





```
        ButtonGroup groupe = new ButtonGroup();
        groupe.add(rouge);
        groupe.add(vert);
        rouge.addActionListener(this);
        vert.addActionListener(this);
        rectangle.addActionListener(this);
        ovale.addActionListener(this);
        rouge.addItemListener(this);
        vert.addItemListener(this);
        rectangle.addItemListener(this);
        ovale.addItemListener(this);
    }
    public void actionPerformed(ActionEvent ev) {
        if (ev.getSource() == rouge) System.out.println("** Action option rouge");
            else if (ev.getSource() == vert) System.out.println("** Action option vert");
                else if (ev.getSource() == rectangle) System.out.println("** Action option rectangle");
                    else System.out.println("** Action option ovale");
    }
    public void itemStateChanged(ItemEvent ev) {
        if (ev.getSource() == rouge) System.out.println("**Item option rouge");
            else if (ev.getSource() == vert) System.out.println("**Item option vert");
                else if (ev.getSource() == rectangle) System.out.println("**Item option rectangle");
                    else System.out.println("**Item option ovale");
        System.out.print("Option Selectionnee :");
        if (rouge.isSelected()) System.out.println(" rouge");
            else if (vert.isSelected()) System.out.println(" vert");
                else if (rectangle.isSelected()) System.out.println(" rectangle");
                    else  System.out.println(" ovale");
    }
}
```

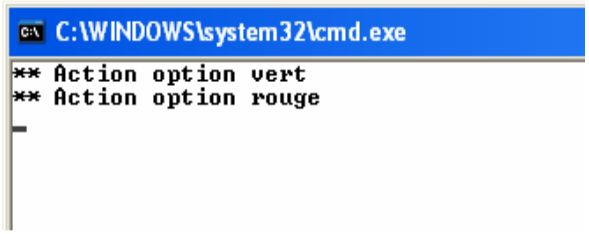
```
D:\WINDOWS\system32\cmd.exe
**Item option rouge
Option Selectionnee : rouge
** Action option rouge
**Item option rouge
Option Selectionnee : ovale
**Item option vert
Option Selectionnee : vert
** Action option vert
**Item option rectangle
Option Selectionnee : vert
** Action option rectangle
**Item option ovale
Option Selectionnee : vert
** Action option ovale
```

18.3. **Les menus surgissants**

Après les menus rattachés à une barre de menus, voyons le cas d'un menu apparaissant dont la liste d'options dépend de l'action de l'utilisateur (en général clic bouton droit).
Pour créer ce type de menu, il faut utiliser la classe *JPopupMenu* auquel il faut rattacher des objets de type *JMenuItem*.
Il est donc aussi possible d'utiliser des boutons radio et/ou cases à cocher dans ces menus.
Nous prenons comme exemple le fait que le menu apparaisse au clic droit de la souris avec le choix « rouge » ou « vert ».
Pour que le menu apparaisse, il faut utiliser la méthode *show* associée à la fenêtre par exemple et aux coordonnées auxquelles on souhaite faire apparaître le menu.
Par le clic bouton droit de la souris, il faut que ce réalise cette apparition, donc il faut utiliser un écouteur de souris qui heureusement possède une méthode *isPopupTrigger* qui renvoie la valeur *true* lorsque le bouton droit est relâché. Il faut donc utiliser la méthode *mouseReleased* dans la classe *MouseEvent* pour l'interface *MouseListener*.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
public class Menu3 {
    public static void main (String []args) {
        FenMenu fen = new FenMenu();
        fen.setVisible(true);
        fen.setDefaultCloseOperation(3);
    }
}
class FenMenu extends JFrame implements ActionListener {
    private JPopupMenu couleur;
    private JMenuItem rouge,vert;
    public FenMenu() {
```

```
        setTitle("Exemple de menu");
        setBounds(100,100,300,150);
        couleur = new JPopupMenu();
        rouge = new JMenuItem("Rouge");
        vert = new JMenuItem("Vert");
        couleur.add(rouge);
        couleur.add(vert);
        rouge.addActionListener(this);
        vert.addActionListener(this);
        addMouseListener(new MouseAdapter() {
           public void mouseReleased(MouseEvent ev) {
              if(ev.isPopupTrigger()) couleur.show(ev.getComponent(),ev.getX(),ev.getY());
           }
        });
    }
    public void actionPerformed(ActionEvent ev) {
        if (ev.getSource() == rouge) System.out.println("** Action option rouge");
            else System.out.println("** Action option vert");
    }
}
```



18.4. Les raccourcis clavier

Dans nombreuses applications commerciales, il est possible d'accéder directement à une option d'un menu ou à un menu en frappant une touche ou une combinaison de touche.
Ce sont les raccourcis clavier. Il en existe de deux types : les caractères mnémoniques et les accélérateurs.

18.4.1. Les caractères mnémoniques

Le caractère mnémonique est un caractère unique souligné dans un menu ou dans une option.
On agit sur un menu en frappant la combinaison « Alt » + lettre, par contre sur une option si le menu est déjà affiché, avec la lettre uniquement.
La méthode permettant d'associer une lettre ou une touche est *setMnemonic(char c) ;*.

18.4.2. Les accélérateurs

Il s'agit d'une combinaison de touches qu'on associe à une option (jamais à un menu) et qui s'affiche à droite de son nom. Il suffit que l'utilisateur frappe cette combinaison de touches pour provoquer la sélection de l'option.
La méthode à utiliser est *setAccelerator(KeyStroke key) ;*.
Un objet de la classe *KeyStroke* est utilisé pour connaître la combinaison de touche.
Pour cela, on utilise KeyStroke.getKeyStroke(KeyEvent.VK_R,InputEvent.CTRL_MASK) ; qui permet de savoir si « CTRL » + « R » a été appuyé.

18.4.3. Exemple

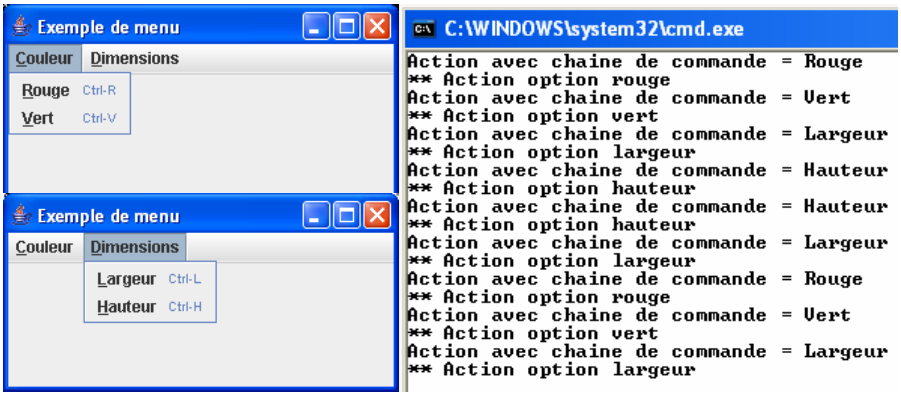
Nous allons reprendre l'exemple 1 (la classe Menu1) dans laquelle, on y ajoute les raccourcis :

- « C » comme touche mnémonique pour atteindre le menu « Couleur »
- « R » comme touche mnémonique pour atteindre l'option « Rouge »
- « V » comme touche mnémonique pour atteindre l'option « Vert »
- « D » comme touche mnémonique pour atteindre le menu « Dimensions »
- « L » comme touche mnémonique pour atteindre l'option « Largeur »
- « H » comme touche mnémonique pour atteindre l'option « Hauteur »
- « CTRL » + « R » comme accélérateur pour atteindre l'option « Rouge »
- « CTRL » + « V » comme accélérateur pour atteindre l'option « Vert »

– « CTRL » + « L » comme accélérateur pour atteindre l'option « Largeur »
– « CTRL » + « H » comme accélérateur pour atteindre l'option « Hauteur »

Voici les lignes de code à ajouter dans le constructeur de la fenêtre.

```
couleur.setMnemonic('C');
dimensions.setMnemonic('D');
rouge.setMnemonic('R');
vert.setMnemonic('V');
largeur.setMnemonic('L');
hauteur.setMnemonic('H');
rouge.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_R,InputEvent.CTRL_MASK));
vert.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_V,InputEvent.CTRL_MASK));
largeur.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_L,InputEvent.CTRL_MASK));
hauteur.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_H,InputEvent.CTRL_MASK));
```

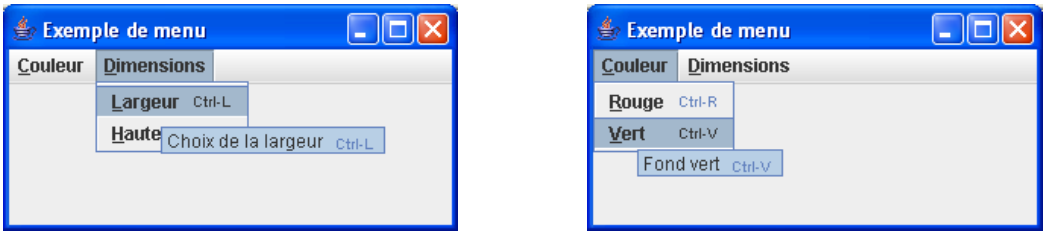


18.5. Les bulles d'aide

Dans les applications commerciales, un petit rectangle, nommé tooltip en anglais, contient un bref texte explicatif lorsque la souris reste un instant sur certains composants.
Pour avoir ces effets, il faut utiliser la méthode *setToolTipText(String texte) ;* au composant voulu.

Reprenons l'exemple précédent, auquel nous ajoutons les infos d'aide pour chaque option.

```
rouge.setToolTipText("Fond rouge");
vert.setToolTipText("Fond vert");
largeur.setToolTipText("Choix de la largeur");
hauteur.setToolTipText("Choix de la hauteur");
```



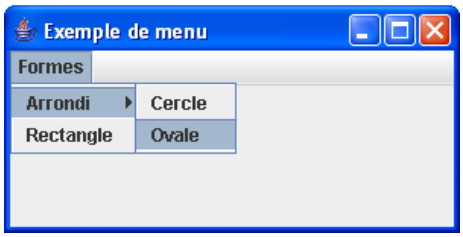
18.6. Compositions des options

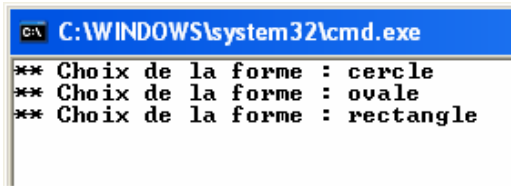
Il est possible que votre choix d'option ne soit pas direct mais une option permet une liste de sous options, il faut utiliser la classe *JMenu* à la place de *JMenuItem* et ce n'est qu'à cette dernière qu'on ajoute des *JMenuItem*.

Supposons que nous voulions choisir dans le menu « Forme », les options « Arrondi » et « Rectangle ». Mais pour l'option « Arrondi », nous avons le choix de « Cercle » ou « Ovale ».
Voici un code source qui le permet :

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
public class Menu6 {
   public static void main (String []args) {
      FenMenu fen = new FenMenu();
      fen.setVisible(true);
      fen.setDefaultCloseOperation(3);
   }
}

class FenMenu extends JFrame implements ActionListener {
   private JMenuBar barreMenus;
   private JMenu forme,arrondi;
   private JMenuItem rectangle, cercle, ovale;
   public FenMenu() {
      setTitle("Exemple de menu");
      setBounds(100,100,300,150);
      barreMenus = new JMenuBar();
      forme = new JMenu("Formes");
      arrondi = new JMenu("Arrondi");
      rectangle = new JMenuItem("Rectangle");
      cercle = new JMenuItem("Cercle");
      ovale = new JMenuItem("Ovale");
      setJMenuBar(barreMenus);
      barreMenus.add(forme);
      forme.add(arrondi);
      forme.add(rectangle);
      arrondi.add(cercle);
      arrondi.add(ovale);
      rectangle.addActionListener(this);
      cercle.addActionListener(this);
      ovale.addActionListener(this);
   }
   public void actionPerformed(ActionEvent ev) {
      if (ev.getSource() == rectangle) System.out.println("** Choix de la forme : rectangle");
         else if (ev.getSource() == cercle) System.out.println("** Choix de la forme : cercle");
            else System.out.println("** Choix de la forme : ovale");
   }
}
```



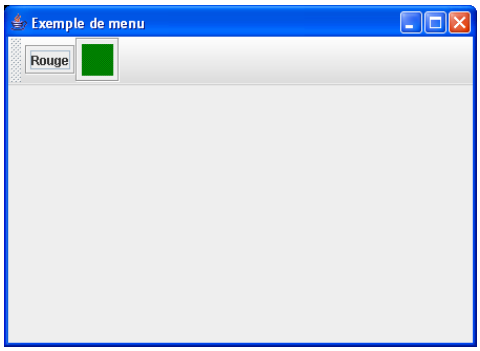


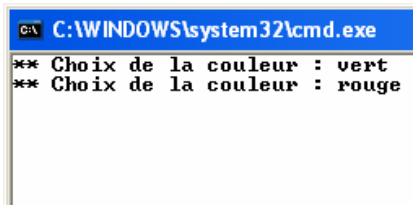
18.7. **Les barres d'outils**

Voyons maintenant comment créer des barres d'outils qui peuvent être déplacée au gré de l'utilisateur. La classe utilisée est *JToolBar*. Il y place des boutons qui sont des objets de type *JButton* avec lesquels il est possible de créer une classe abstraite d'écoute pour la différencier dans l'interface *ActionListener* pour la méthode *actionPerformed*.
On peut y ajouter dans ces barres d'outils des icônes que l'on a ajoutés sur les boutons.
Voici un exemple dans lequel on crée une barre d'outils formée de deux boutons, l'un avec texte et l'autre avec une image (un carré vert) qui affiche un message (annonce de la couleur) lorsqu'ils sont actionnés.

```
import java.awt.*; import java.awt.event.*;
import javax.swing.*; import javax.swing.event.*;
public class Menu7 {
   public static void main (String []args) {
```

```
        FenMenu fen = new FenMenu();
        fen.setVisible(true);
        fen.setDefaultCloseOperation(3);
    }
}
class FenMenu extends JFrame implements ActionListener {
    private JToolBar barreOutils;
    private JButton rouge,vert;
    public FenMenu() {
        setTitle("Exemple de menu");
        setBounds(100,100,300,150);
        Container co = getContentPane();
        co.setLayout(new BorderLayout());
        barreOutils = new JToolBar();
        rouge = new JButton("Rouge");
        vert = new JButton(new ImageIcon("vert.gif"));
        barreOutils.add(rouge);
        barreOutils.add(vert);
        co.add(barreOutils,"North");
        rouge.addActionListener(this);
        vert.addActionListener(this);
    }
    public void actionPerformed(ActionEvent ev) {
        if (ev.getSource() == rouge) System.out.println("** Choix de la couleur : rouge");
            else System.out.println("** Choix de la couleur : vert");
    }
}
```



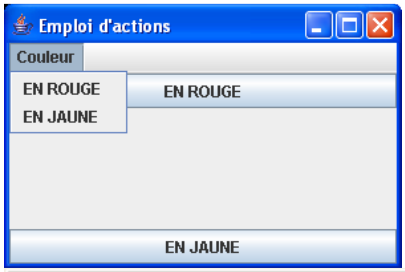


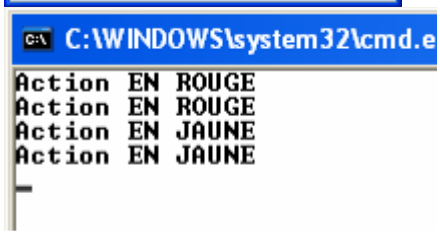
Il est possible d'interdire la barre d'outils de bouger à l'aide de l'instruction :
barreOutils.setFloatable(false) ;

18.8. **Les actions**

Dans une application de grande taille, il existe souvent plusieurs manières de déclencher une même action. Si l'on souhaite créer des logiciels de qualités, il est préférable que le traitement d'une action donnée ne soit réalisé qu'en un seul point du code.
Java permet à l'aide de la classe *AbstractAction* de réaliser ce point.
Il est clair que l'on doit créer une classe qui étend la classe *AbstractAction* pour qu'elle puisse agir à sa guise. Ce principe est le même que celui utilisé pour les *JPanel*.
Voyons un exemple qui permet d'associer une même action à plusieurs éléments différents (*JButton*, *JMenuItem*).
Dans cet exemple, on traite le cas particulier du bouton, pour lequel on doit utiliser la méthode *getValue* de la classe *AbstractAction*.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
public class Menu8 {
    public static void main (String []args) {
        FenMenu fen = new FenMenu();
        fen.setVisible(true);
        fen.setDefaultCloseOperation(3);
    }
}
class FenMenu extends JFrame {
    private JMenuBar menu;
    private JMenu couleur;
    private JButton boutonRouge,boutonJaune;
    private ActionCouleur actionRouge,actionJaune;
    public FenMenu() {
        setTitle("Emploi d'actions");
```





```
        setBounds(100,100,300,200);
        menu = new JMenuBar();
        setJMenuBar(menu);
        couleur = new JMenu("Couleur");
        menu.add(couleur);
        actionRouge = new ActionCouleur("EN ROUGE",Color.red);
        actionJaune = new ActionCouleur("EN JAUNE",Color.yellow);
        couleur.add(actionRouge);
        couleur.add(actionJaune);
        boutonRouge = new JButton((String)actionRouge.getValue(Action.NAME));
        boutonJaune = new JButton((String)actionJaune.getValue(Action.NAME));
        getContentPane().add(boutonRouge,"North");
        getContentPane().add(boutonJaune,"South");
        boutonRouge.addActionListener(actionRouge);
        boutonJaune.addActionListener(actionJaune);
    }
}
class ActionCouleur extends AbstractAction {
    private Color couleur;
    public ActionCouleur (String nom, Color couleur) {
        super(nom);
        this.couleur = couleur;
    }
    public void actionPerformed (ActionEvent ev) {
        System.out.println("Action " + ev.getActionCommand());
    }
}
```

18.9. **Exemple complet**

Voici un exemple complet qui permette d'intégrer les dernières notions vues aux premières.
L'application comporte trois choix de menu (couleur, formes, dimensions).
Pour les couleurs, quatre choix sont possibles : rouge, jaune, vert, bleu. (avec les actions)
Pour les formes, deux choix : rectangle ou ovale. (cases à cocher)
Pour les dimensions, deux boîtes de dialogue distinctes apparaissent et permettent de donner la valeur que l'on désire.
Il y a aussi une barre d'outils pour le choix des couleurs qui sont des boutons avec des icônes représentant les couleurs désirées.
Un menu surgissant apparaît lors d'un clic droit de la souris qui permet de choisir la couleur désirée.
Finalement, on ajoute pour chaque choix, un bulle d'aide.

```
import java.awt.*; import java.awt.event.*;
import javax.swing.*; import javax.swing.event.*;
public class Menu9 {
    public static void main (String []args) {
        FenMenu fen = new FenMenu();
        fen.setVisible(true);
        fen.setDefaultCloseOperation(3);
    }
}
class FenMenu extends JFrame implements ActionListener {
    private JMenuBar barreMenus;
    private JMenu couleur, formes, dimensions;
    private JMenuItem largeur, hauteur;
    private JCheckBoxMenuItem rectangle, ovale;
    private JPopupMenu couleurSurgissant;
    private ActionCouleur[] actions;
    private JToolBar barreCouleurs;
    private JButton [] boutonCouleurs;
    private Panneau pan;
    static public final String [] nomCouleurs = {"Rouge", "Vert", "Jaune", "Bleu"};
```

```
static public final Color [] couleurs = {Color.red, Color.green, Color.yellow, Color.blue};
static public final String [] nomIcone = {"rouge.gif", "vert.gif", "jaune.gif", "bleu.gif"};
public FenMenu() {
   setTitle("Figures avec menus et barres d'outils");
   setBounds(100,100,600,450);
   Container co = getContentPane();
   co.setLayout(new BorderLayout());
   pan = new Panneau();
   co.add(pan);
   pan.setBackground(Color.cyan);
   int nbCouleurs = nomCouleurs.length;
   actions = new ActionCouleur[nbCouleurs];
   barreCouleurs = new JToolBar();
   boutonCouleurs = new JButton[nbCouleurs];
   couleur = new JMenu("Couleur");
   couleurSurgissant = new JPopupMenu();
   for (int i=0; i<nbCouleurs ;i++){
      actions[i] = new ActionCouleur(nomCouleurs[i], couleurs[i], nomIcone[i],pan);
      couleur.add(actions[i]);
      couleurSurgissant.add(actions[i]);
      JButton boutonCourant = barreCouleurs.add(actions[i]);
      boutonCourant.setText(null);
      boutonCourant.setToolTipText((String)actions[i].getValue(Action.SHORT_DESCRIPTION));
   }
   co.add(barreCouleurs,"North");
   barreMenus = new JMenuBar();
   addMouseListener(new MouseAdapter() {
      public void mouseReleased(MouseEvent ev) {
         if(ev.isPopupTrigger()) couleurSurgissant.show(ev.getComponent(),ev.getX(),ev.getY());
      }
   });
   formes = new JMenu("Formes");
   rectangle = new JCheckBoxMenuItem("Rectangle");
   ovale = new JCheckBoxMenuItem("Ovale");
   dimensions = new JMenu("Dimensions");
   largeur = new JMenuItem("Largeur");
   hauteur = new JMenuItem("Hauteur");
   setJMenuBar(barreMenus);
   barreMenus.add(couleur);
   barreMenus.add(formes);
   barreMenus.add(dimensions);
   formes.add(rectangle);
   formes.add(ovale);
   dimensions.add(largeur);
   dimensions.add(hauteur);
   couleur.setMnemonic('C');
   dimensions.setMnemonic('D');
   formes.setMnemonic('F');
   rectangle.setMnemonic('R');
   ovale.setMnemonic('O');
   largeur.setMnemonic('L');
   hauteur.setMnemonic('H');
   largeur.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_L,InputEvent.CTRL_MASK));
   hauteur.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_H,InputEvent.CTRL_MASK));
   rectangle.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_R,InputEvent.CTRL_MASK));
   ovale.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_O,InputEvent.CTRL_MASK));
   largeur.addActionListener(this);
   hauteur.addActionListener(this);
   rectangle.addActionListener(this);
   ovale.addActionListener(this);
}
public void actionPerformed(ActionEvent ev) {
   if(ev.getSource() == largeur) {
      String ch = JOptionPane.showInputDialog(this,"Largeur");
      pan.setLargeur (Integer.parseInt(ch));
   }
   else if(ev.getSource() == hauteur) {
```

```
            String ch = JOptionPane.showInputDialog(this,"Hauteur");
            pan.setHauteur (Integer.parseInt(ch));
        }
        if(ev.getSource() == ovale) {
            pan.setOvale(ovale.isSelected());
        }
        if(ev.getSource() == rectangle) {
            pan.setRectangle(rectangle.isSelected());
        }
        pan.repaint();
    }
}
class Panneau extends JPanel {
    private boolean rectangle = false, ovale = false;
    private int largeur = 50, hauteur = 50;
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        if (ovale) g.drawOval(10,10,10+largeur,10+hauteur);
        if (rectangle) g.drawRect(10,10,10+largeur,10+hauteur);
    }
    public void setRectangle(boolean trace) {
        rectangle = trace;
    }
    public void setOvale(boolean trace) {
        ovale = trace;
    }
    public void setLargeur(int largeur) {
        this.largeur = largeur;
    }
    public void setHauteur(int hauteur) {
        this.hauteur = hauteur;
    }
    public void setCouleur(Color c) {
        setBackground(c);
    }
}
class ActionCouleur extends AbstractAction {
    private Color couleur;
    private Panneau pan;
    static ActionCouleur actionInactive;
    public ActionCouleur (String nom, Color couleur, String nomIcone, Panneau pan) {
        putValue(Action.NAME, nom);
        putValue(Action.SMALL_ICON, new ImageIcon(nomIcone));
        putValue(Action.SHORT_DESCRIPTION,"Fond "+nom);
        this.couleur = couleur;
        this.pan = pan;
    }
    public void actionPerformed (ActionEvent ev) {
        pan.setCouleur(couleur);
        pan.repaint();
        setEnabled(false);
        if (actionInactive != null) actionInactive.setEnabled(true);
        actionInactive= this;
    }
}
```

18.10. **Exercice**

Reprenez le jeu du pendu et améliorez-le avec les boîtes et les menus.

19. Textes et graphiques

Nous avons vu qu'il est possible de créer des lignes, rectangles ou ovales. Nous allons voir maintenant qu'il est possible de « dessiner » du texte et avec cela le choix de police, de couleur. Ces deux derniers cas peuvent être intégrés pour n'importe quel composant.

19.1. Position du texte

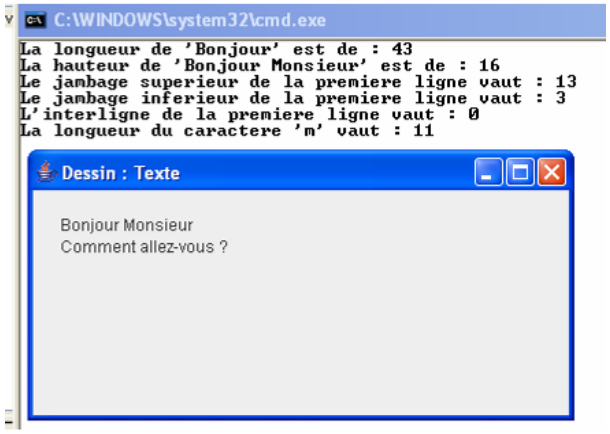
Pour « dessiner » du texte, la méthode est *drawString* avec comme paramètre le texte à dessiner, et la position, largeur et hauteur : g.drawString("Bonjour",50,100) ;.
Il existe d'autres méthodes qui permettent d'afficher du texte sur une même ligne, sur deux lignes consécutives :

- ✓ FontMetrics fm = g.getFontMetrics() ; : permet d'obtenir les caractéristiques de la fonte courante ;
- ✓ int lg = fm.stringWidth("Bonjour") ; : permet d'obtenir la longueur en pixels du texte introduit ;
- ✓ int hg = fm.getHeight() ; : permet de connaître la hauteur avec l'interligne du dessin précédent ;
- ✓ int ja = fm.getAscent() ; : permet de connaître la hauteur du jambage haut ;
- ✓ int jd = fm.getDescent() ; : permet de connaître la hauteur du jambage bas ;
- ✓ int i = fm.getLeading() ; : permet de connaître la hauteur de l'interligne ;
- ✓ int lc = fm.charWidth('a') ; : permet de connaître la longueur de la lettre a.

Exemple :

Dans cet exemple, nous placerons les mots « bonjour » et « monsieur », l'un à la suite de l'autre, suivi en dessous de « Comment allez-vous ? ».
Nous afficherons à la fenêtre console, les différentes valeurs des méthodes citées.

```
import javax.swing.*; import java.awt.*;
public class Dessin1 {
   public static void main (String [] args) {
      MaFenetre fen = new MaFenetre();
      fen.setVisible(true); fen.setDefaultCloseOperation(3);
   }
}
class MaFenetre extends JFrame {
   Panneau pan;
   public MaFenetre() {
      setTitle("Dessin : Texte"); setBounds(100,100,400,200);
      pan = new Panneau();
      getContentPane().add(pan);
   }
}
class Panneau extends JPanel {
   public void paintComponent(Graphics g) {
      super.paintComponent(g);
      int x = 20, y = 30;
      String ch1 = "Bonjour"; String ch2 = " Monsieur";String ch3 = "Comment allez-vous ?";
      g.drawString(ch1,x,y);
      FontMetrics fm = g.getFontMetrics();
      x += fm.stringWidth(ch1);
      g.drawString(ch2,x,y);
      y += fm.getHeight();
      g.drawString(ch3,(x - fm.stringWidth(ch1))),y);
      System.out.println("La longueur de 'Bonjour' est de : "+fm.stringWidth(ch1));
      System.out.println("La hauteur de 'Bonjour Monsieur' est de : "+fm.getHeight());
      System.out.println("Le jambage superieur de la premiere ligne vaut : "+fm.getAscent());
      System.out.println("Le jambage inferieur de la premiere ligne vaut : "+fm.getDescent());
      System.out.println("L'interligne de la premiere ligne vaut : "+fm.getLeading());
      System.out.println("La longueur du caractere 'm' vaut : "+fm.charWidth('m'));
   }
}
```



19.2. **Choix de la police**

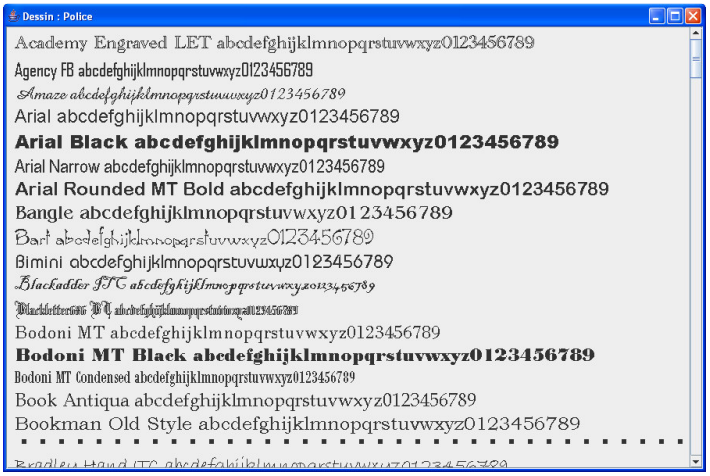
Il existe deux types de police : les fontes logiques et les fontes physiques.
Les fontes logiques sont au nombre de 5 : SansSerif, Serif, Monospacde, Dialog et DialogInput avec 4 styles possibles : normal (*Font.PLAIN*), gras (*Font.BOLD*), italique (*Font.ITALIC*) et gras-italique (*Font.BOLD+Font.ITALIC*).
Les fontes physiques dépendent du nombre de police « ttf » que vous possédez dans votre environnement.
La construction est identique pour les deux types : `Font fl = new Font("le nom de la police", style, taille) ;`.
Le nom de la police ets une chaîne de caractère, le style est une valeur numérique entière de 0 à 3 ou une des constantes citées et la taille de la police.
Pour attribuer une fonte à un élément contenant du texte, on utilise la méthode *setFont*.
Il existe des méthodes intéressantes pour les fontes :

- ✓ *getFontName* : fournit le nom de la police et son style ;
- ✓ *getFamily* : fournit le nom de la famille de police ;
- ✓ *getName* : fournit le nom logique s'il existe, sinon le nom de la police.

Exemple :

Nous allons construire un programme qui affiche toutes polices de l'environnement avec le nom de la police et le texte : « abcdefghijklmnopqrstuvwxyz0123456789 ».
Pour ce programme, nous avons besoin d'un ascenseur vu que l'on ne connaît pas à l'avance le nombre de police. Pour cela, un composant spécifique : *JScrollPane*.
Pour connaître les polices de l'environnement, il faut créer un tableau de chaîne de caractères avec la méthode : `GraphicsEnvironment.getLocalGraphicsEnvironment().getAvailableFontFamilyNames() ;`.

```
import javax.swing.*; import java.awt.*;
public class Dessin2 {
   public static void main (String [] args) {
      MaFenetre fen = new MaFenetre();
      fen.setVisible(true); fen.setDefaultCloseOperation(3);
   }
}
class MaFenetre extends JFrame {
   Panneau pan;
   public MaFenetre() {
      setTitle("Dessin : Police");
      setBounds(100,100,900,600);
      pan = new Panneau();
      pan.setPreferredSize(new Dimension(850,6000));
      getContentPane().add(new JScrollPane(pan));
   }
}
class Panneau extends JPanel {
   public void paintComponent(Graphics g) {
      super.paintComponent(g);
      String [] fontes = GraphicsEnvironment.getLocalGraphicsEnvironment().getAvailableFontFamilyNames();
      int x = 10, y = 10;
      for (int i = 0 ; i < fontes.length ; i++) {
         g.setFont(new Font(fontes[i], Font.PLAIN, 24));
         y += g.getFontMetrics().getAscent();
         String ch = fontes[i] + " abcdefghijklmnopqrstuvwxyz0123456789";
         g.drawString(ch,x,y);
         y += g.getFontMetrics().getDescent() + g.getFontMetrics().getLeading();
      }
   }
}
```



19.3. **La couleur**

La couleur pour la police ou pour tout autre objet est aussi un objet de type *Color*.
Pour le construire : *Color c = new Color(int r, int g, int b) ;* r (rouge), g (vert), b (bleu), valeur entre 0 et 255.
Il existe des contantes directement utilisables :

Nom de la constante	Couleur correspondante
Color.black	noir
Color.blue	bleu
Color.cyan	cyan
Color.darkGray	gris foncé
Color.gray	gris
Color.green	vert
Color.lightGray	gris clair
Color.magenta	magenta
Color.orange	orange
Color.pink	rose
Color.red	rouge
Color.white	blanc
Color.yellow	jaune

Il est possible à partir d'une couleur de la foncer ou l'éclairir à l'aide des méthodes : *brighter* et *darker*. Dans les deux cas le facteur multiplicateur est 1/0,7 ou 0,7.
Pour les composants texte autres que dessin, pour définr la couleur, on utilise la méthode *setForeground*.
Par contre dans les dessins pour n'importe quel dessin, la méthode est *setColor*.

19.4. **Le tracé de lignes et de formes**

Nous avions déjà vu le cas des lignes : *drawLine*, des rectangles : *drawRect* et des ovales : *drawOval*, il existe encore qautre autres types de dessin : les rectangles à coins arrondis : *drawRoundRect*, les lignes brisées : *drawPolyLine*, les polygones : *drawPolygon* et les arcs d'ellipse : *drawArc*.

- ✓ drawLine : en paramètre, on écrit la position du point initial et final.
- ✓ drawRect : en paramètre, on écrit la position du coin supérieur gauche et le coin inférieur droit.
- ✓ drawOval : en paramètre, on écrit la position du coin supérieur gauche et le coin inférieur droit du rectangle circonscrit.
- ✓ drawRoundRect : en paramètre, on écrit la position du coin supérieur gauche et le coin inférieur droit ainsi que les deux valeurs des axes des ellipses servant à former les angles.
- ✓ drawPolyline : en paramètre, on écrit un tableau d'abscisses, d'ordonnées des différents points composant la ligne brisée, ainsi que le nombre de lignes.

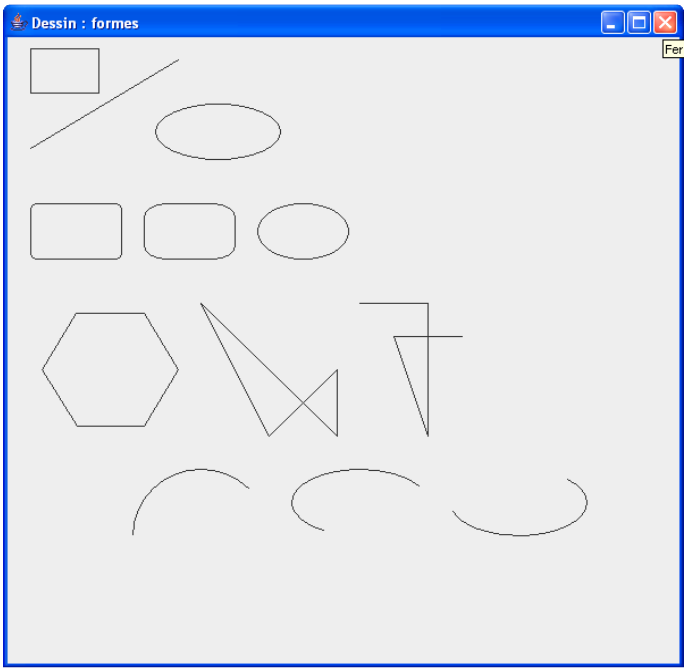
- ✓ drawPolygon : en paramètre, on écrit un tableau d'abscisses, d'ordonnées des différents points composant la ligne brisée, ainsi que le nombre de lignes.
- ✓ drawArc : en paramètre, on écrit la position du coin supérieur gauche et le coin inférieur droit du rectangle circonscrit ainsi que deux valeurs entières correspondant respectivement à l'angle de début du tracé et à la dimension angulaire de l'arc en degrés. Ces valeurs peuvent être négatives.

Il existe une méthode très intéressante *translate* qui permet de changer d'origine.

Exemple :

Nous allons dessiner sur un panneau chacun des dessins possibles.

```
import javax.swing.*;
import java.awt.*;
public class Dessin3 {
   public static void main (String [] args) {
      MaFenetre fen = new MaFenetre();
      fen.setVisible(true);
      fen.setDefaultCloseOperation(3);
   }
}
class MaFenetre extends JFrame {
   Panneau pan;
   public MaFenetre() {
      setTitle("Dessin : formes  ");
      setBounds(100,100,600,600);
      pan = new Panneau();
      getContentPane().add(pan);
   }
}
class Panneau extends JPanel {
   public void paintComponent(Graphics g) {
      super.paintComponent(g);
      g.drawRect(20,10,60,40);
      g.drawLine(20,100,150,20);
      g.drawOval(130,60,110,50);
      int larg = 80, haut = 50;
      g.translate(20,150);
      g.drawRoundRect(0,0,larg,haut,10,10);
      g.translate(100,0);
      g.drawRoundRect(0,0,larg,haut,40,25);
      g.translate(100,0);
      g.drawRoundRect(0,0,larg,haut,larg,haut);
      g.translate(-200,80);
      int r = 60;
      g.translate(10+r,10+r);
      int[] x = new int[6];
      int[] y = new int[6];
      for (int i = 0 ; i < 6 ; i++) {
         x[i] = (int) (r*Math.cos(i*Math.PI/3));
         y[i] = (int) (r*Math.sin(i*Math.PI/3));
      }
      g.drawPolygon(x,y,6);
      g.translate(2*r+20,0);
      x = new int [4];
      y = new int [4];
      x[0] = y[0] = -r;
      x[1] = y[1] = r;
      x[2] = y[3] = r;
      x[3] = y[2] = 0;
      g.drawPolygon(x,y,4);
      g.translate(r+20,-r);
      x = new int [5];
      y = new int [5];
      x[0] = y[0] = y[1] = 0;
      x[1] = x[2] = r;
      y[2] = 2*r;
      x[3] = y[3] = y[4] = r/2;
      x[4] = 3*r/2;
      g.drawPolyline(x,y,5);
      g.translate(-200,150);
      g.drawArc(0,0,2*r,2*r,45,135);
      g.translate(2*r+20,0);
      g.drawArc(0,0,2*r,r,30,210);
      g.translate(2*r+20,0);
      g.drawArc(0,0,2*r,r,45,-210);
   }
}
```

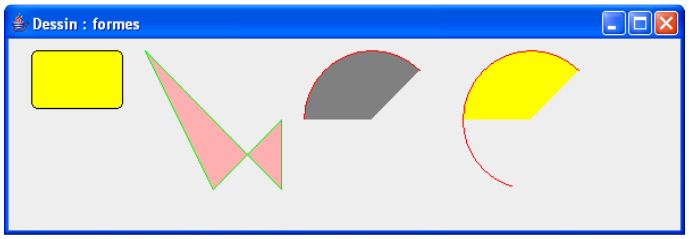


19.5. **Remplissage de formes**

Pour remplir des formes par une couleur, il suffit de choisir les méthodes homologues à celles qui dessinent des formes en changeant le mot *draw* par *fill*.
Il est possible maintenant de jongler avec les deux pour mettre une bordure d'une autre couleur en dessinant la forme pleine et ensuite la forme sans contenu mais avec une autre couleur.

Exemple :

Nous allons dessiner sur un panneau un rectangle arrondi jaune avec un bord noir, un polygone rose avec un bord vert, un arc gris avec un bord rouge et un autre arc jaune avec un bord plus grand rouge.




```
import javax.swing.*; import java.awt.*;
public class Dessin4 {
   public static void main (String [] args) {
      MaFenetre fen = new MaFenetre();
      fen.setVisible(true);
      fen.setDefaultCloseOperation(3);
   }
}
class MaFenetre extends JFrame {
   Panneau pan;
   public MaFenetre() {
      setTitle("Dessin : formes  ");
      setBounds(100,100,600,200);
      pan = new Panneau();
      getContentPane().add(pan);
   }
}
class Panneau extends JPanel {
   public void paintComponent(Graphics g) {
      super.paintComponent(g);
      int larg = 80, haut = 50;
      g.translate(20,10);
      g.setColor(Color.yellow);
      g.fillRoundRect(0,0,larg,haut,10,10);
      g.setColor(Color.black);
      g.drawRoundRect(0,0,larg,haut,10,10);
      int r = 60;
      int [] x = new int [4];
      int [] y = new int [4];
      g.translate(larg+r+20,r);
      x[0] = y[0] = -r;
      x[1] = y[1] = x[2] = y[3] = r;
      x[3] = y[2] = 0;
      g.setColor(Color.pink);
      g.fillPolygon(x,y,4);
      g.setColor(Color.green);
      g.drawPolygon(x,y,4);
      g.translate(r+20,-r);
      g.setColor(Color.gray);
      g.fillArc(0,0,2*r,2*r,45,135);
      g.setColor(Color.red);
      g.drawArc(0,0,2*r,2*r,45,135);
      g.translate(2*r+20,0);
      g.setColor(Color.yellow);
      g.fillArc(0,0,2*r,2*r,45,135);
      g.setColor(Color.red);
      g.drawArc(0,0,2*r,2*r,45,210);
   }
}
```

19.6. Le mode dessin

Même si ce dernier exemple laisse imaginer qu'il est possible superposer des dessins, il n'en est rien. Car si nous inversons le dessin plein et la ligne, on ne verrait plus que le dessin plein. Heureusement, il existe un autre mode de dessin *XOR* que l'on invoque par la méthode *setXORMode* avec comme paramètre un objet de type *Color* qui est la couleur de fond du composant.

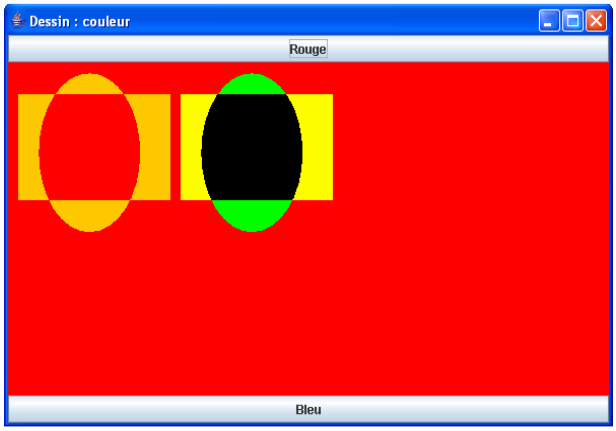
- ✓ L'affichage sur une zone ayant la couleur de fond est fait avec la couleur courante ;
- ✓ l'affichage sur une zone yanat la couleur courante est fait avec la couleur de fond ;
- ✓ l'affichage sur une zone ayant une couleur différente de celle du fond et de la couleur courante ets fait dans une couleur différente des deux autres.

Pour revenir au mode classique de dessin, la méthode est *setPaintMode*.

Exemple :

Nous allons dessiner sur un panneau un rectangle et un ovale de couleur différent, nous allons les superposer et à l'aide de bouton changer les couleurs de fond.

```
import javax.swing.*; import java.awt.*;
import javax.swing.event.*; import java.awt.event.*;
public class Dessin5 {
   public static void main (String [] args) {
      MaFenetre fen = new MaFenetre();
      fen.setVisible(true);
      fen.setDefaultCloseOperation(3);
   }
}
class MaFenetre extends JFrame implements ActionListener {
   Panneau pan;
   JButton rouge, bleu;
   public MaFenetre() {
      setTitle("Dessin : couleur  ");
      setBounds(100,100,600,400);
      pan = new Panneau();
      getContentPane().add(pan);
      rouge = new JButton("Rouge");
      bleu = new JButton("Bleu");
      getContentPane().add(rouge,"North");
      getContentPane().add(bleu,"South");
      rouge.addActionListener(this);
      bleu.addActionListener(this);
   }
   public void actionPerformed(ActionEvent ae) {
      if(ae.getSource() == rouge) pan.setBackground(Color.red);
         else pan.setBackground(Color.cyan);
   }
}
class Panneau extends JPanel {
   public void paintComponent(Graphics g) {
      super.paintComponent(g);
      g.setXORMode(getBackground());
      g.setColor(Color.orange);
      g.fillRect(10,30,150,100);
      g.fillOval(30,10,100,150);
      g.setColor(Color.yellow);
      g.fillRect(170,30,150,100);
      g.setColor(Color.green);
      g.fillOval(190,10,100,150);
   }
}
```

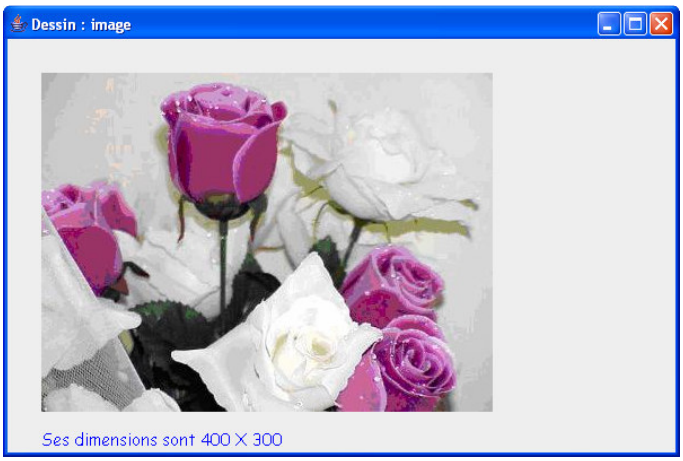


19.7. **Affichage d'images**

Il est possible en Java de faire apparaître des images dans les étiquettes mais aussi comme dessin. Pour cela il faut avant tout créer un objet de type *ImageIcon* et ensuite invoquer la méthode *drawImage*. Les seuls formats d'image acceptés sont JPG et GIF.
La procédure :

```
ImageIcon imIc new ImageIcon("rose.jpg") ;
Image im = imIc.getImage() ;
g.drawImage(im,x,y,null) ;
```

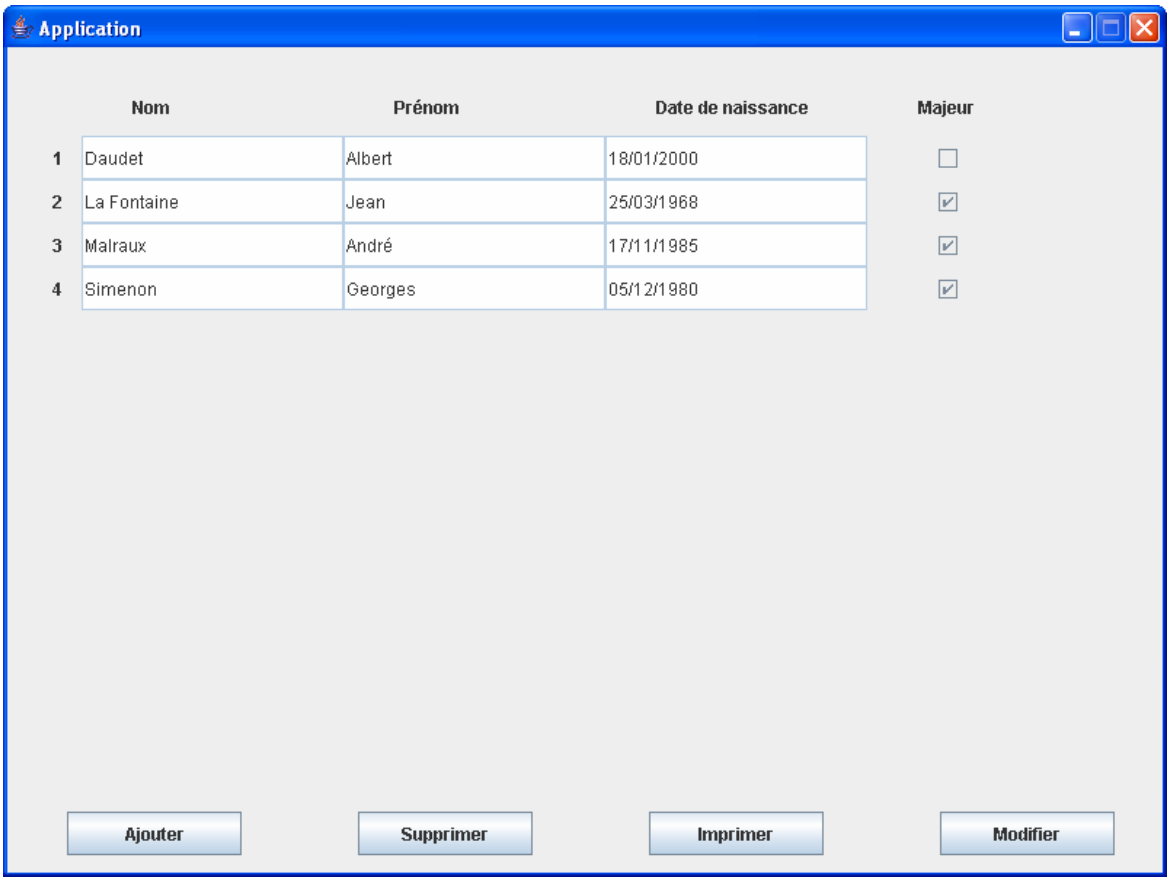
x et y précisent les coordonnées du point où sera placé le coin supérieur gauche de l'image et le quatrième paramètre obligatoire à *null*.
Les méthodes *getIconHeight* et *getIconWidth* permettent de connaître la taille en pixels de l'image.




```
import javax.swing.*;
import java.awt.*;
public class Dessin6 {
   public static void main (String [] args) {
      MaFenetre fen = new MaFenetre();
      fen.setVisible(true);
      fen.setDefaultCloseOperation(3);
   }
}
class MaFenetre extends JFrame {
   Panneau pan;
   public MaFenetre() {
      setTitle("Dessin : image  ");
      setBounds(100,100,600,400);
      pan = new Panneau();
      getContentPane().add(pan);
   }
}
class Panneau extends JPanel {
   private ImageIcon rose;
   private int hauteur, largeur;
   public Panneau() {
      rose = new ImageIcon("P1000447.JPG");
      //rose = new ImageIcon("rose.jpg");
      hauteur = rose.getIconHeight();
      largeur = rose.getIconWidth();
   }
   public void paintComponent(Graphics g) {
      super.paintComponent(g);
      int rapportY = (int) (largeur/600)+1 ;
      int rapportX = (int) (hauteur/360)+1;
      System.out.println(rapportX +" " +rapportY);
      int proportion = rapportX < rapportY ? rapportY : rapportX;
      int haut = (int) hauteur / proportion;
      int larg = (int) largeur / proportion;
      System.out.println(larg + " "+ haut);
      g.drawImage(rose.getImage(),20,20,larg,haut ,null);
      String texte = "Ses dimensions sont "+ largeur + " X " + hauteur;
      g.setFont(new Font("Comic Sans Ms",0,15));
      g.setColor(Color.blue);
      g.drawString(texte,20,360);
   }
}
```

19.8. **Exercice**

Création d'une interface similaire à ci-dessous :



Action des boutons :

« Ajouter » : Ouvrir une boîte de dialogue qui permette dintroduire un nouvel utilisateur et ensuite l'afficher à la suite de la fenêtre.

« Supprimer » : Ouvrir un boîte de saisie pour que l'utilisateur puisse introduire le numéro de la ligne à supprimer. La ligne est supprimée et les enregistrements se recollent.

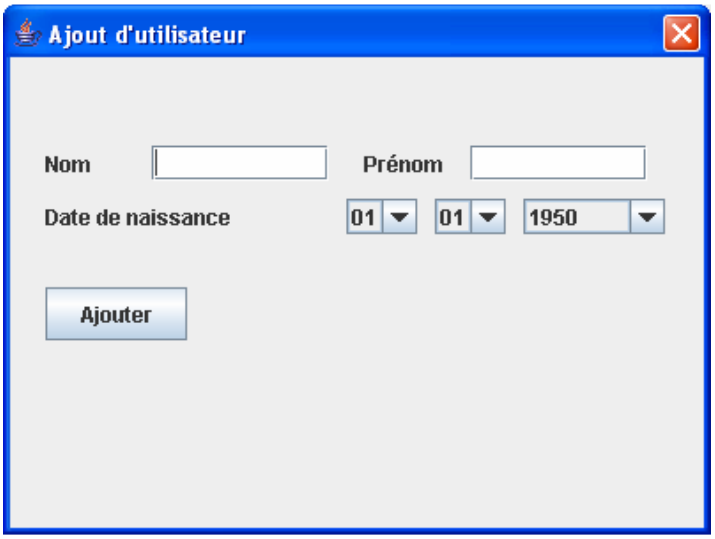
« Imprimer » : Afficher à la fenêtre console les infos de la fenêtre graphique

« Modifier » : Ouvrir un boîte de saisie pour que l'utilisateur puisse introduire le numéro de la ligne à modifier. Alors une boîte de dialogue apparaît pour les nouvelles données. Ces données remplaceront les anciennes.

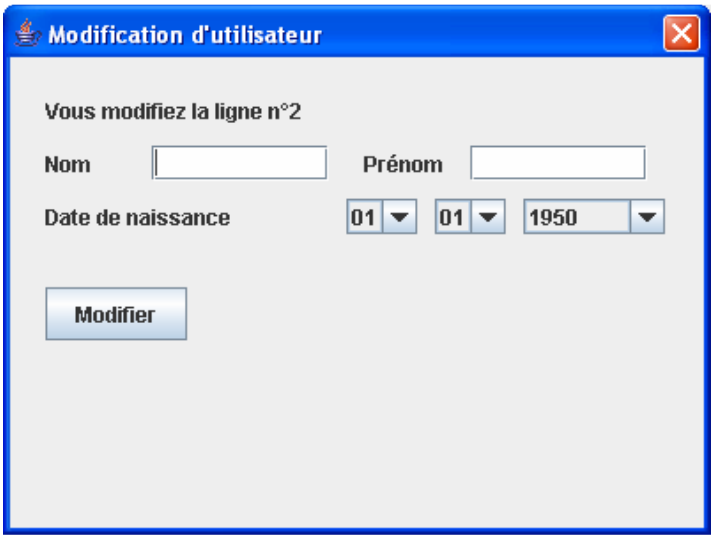
Les éléments de la fenêtre ne peuvent être modifiés par la fenêtre.

La case à cocher « Majorité » est cochée si l'utilisateur est agé de plus de 18 ans à la date système.
Lorsqu'il n'y a plus d'utilisateur, les boutons « supprimer », « imprimer » et « modifier » sont désactivés.

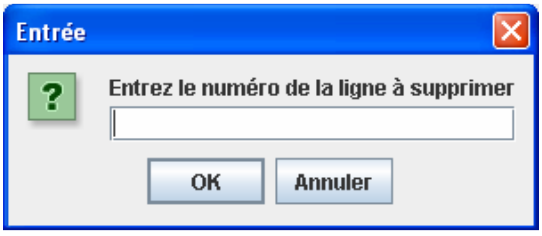
La boîte de dialogue « Ajouter » :



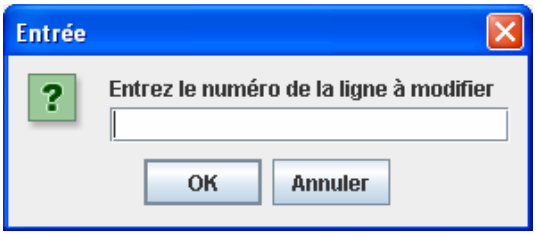
La boîte de dialogue « Modifier » :



La boîte de saisie « Supprimer » :

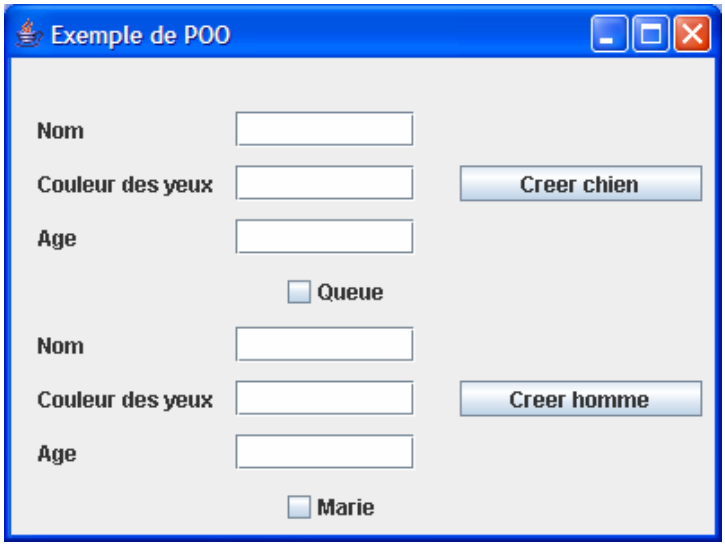


La boîte de saisie « Modifier » :



Après avoir vu, les prémices de la programmation orientée objet (chapitre II) et de la programmation graphique et événementielle, nous allons développer une application qui crée deux objets (un chien et un homme) et montrer tous les mécanismes de la POO à partir de cet exemple. Cet exemple va utiliser des classes graphiques qui permettent d'utiliser une interface plus agréable.

1. Création de l'interface graphique



Créer une interface graphique qui soit similaire à ce qui précède à l'aide des différents objets vus jusqu'à maintenant.
Il faut utiliser une classe *XYLayout* qui permet de positionner à un endroit voulu de la fenêtre ou du panneau.
Pour l'utilisation, il faut créé pour l'objet de type *Container* ou de type *JPanel*, un objet de type *XYLayout* dans la méthode *setLayout* instanciée sur l'objet.

co.setLayout(new XYLayout(int hauteur,int largeur)); ou pan.setLayout(new XYLayout(int hauteur,int largeur));

Pour ajouter des composants à un endroit spécifique, il faut écrire :
co.add(String, composant); L'élément String équivaut à "xposition, yposition, longueur, hauteur"

Tous les composants font 20 pixels de haut et une longueur de 100 exceptés les boutons qui ont une longueur de 135 pixels.
Les étiquettes sont positionnées à 15 pixels du bord gauche, les champs de texte à 125, les cases à cocher à 150 et les boutons à 250.
Quant à la position en hauteur, ils sont positionnés de 30 en 30 pixels.

Deux actions uniquement donnent quelque chose, les clics sur les boutons.
Le clic sur `"Créer chien"` permet de créer un objet de type *ClasseChien* qui possède 4 champs, deux de type *String* (nom et couleurYeux), un de type *int* (age) et un de type *boolean* (queue) et d'attribuer à *nom* : Snoopy, à *couleurYeux* : marron, à *age* : 2 et à *queue* : true.
Le clic sur `"Créer homme"` permet de créer un objet de type *ClasseHomme* qui possède 4 champs, deux de type *String* (nom et couleurYeux), un de type *int* (age) et un de type *boolean* (marie) et d'attribuer à *nom* : Steven, à *couleurYeux* : bleu, à *age* : 35 et à *marie* : true.

```
import java.awt.*; import java.awt.event.*; import javax.swing.*; import javax.swing.event.*;
public class ExemplePOO {
   public static void main(String args[]){
   Fen fenetre = new Fen();
   fenetre.setVisible(true);
   fenetre.setDefaultCloseOperation(3);
   fenetre.setResizable(false);
   }
}
class Fen extends JFrame implements ActionListener {
   JTextField champs [] = new JTextField [7];
   JButton creerChien,creerHomme;
   JCheckBox aUneQueue, estMarie;
   ClasseChien chien;
   ClasseHomme homme;
   public Fen() {
      setTitle ("Exemple POO");
      setBounds(300,75,400,310);
      Container co = getContentPane();
      co.setLayout(new FlowLayout());
      JPanel pan = new JPanel();                    //On peut aussi faire sans panneau et tout placer sur la fenêtre, alors
      pan.setPreferredSize(new Dimension(400,270)); //pan est remplacé par co dans la suite et l'instruction co.add(pan)
      pan.setLayout(new XYLayout(270,400));         //disparaît
      String etiquette[] = {"Nom","Couleur des yeux","Age","","Nom","Couleur des yeux","Age"};
      JLabel lab [] = new JLabel [7];
      for (int i = 0 ; i < 7 ; i++ ) {
         lab[i] = new JLabel(etiquette[i]);
         champs[i] = new JTextField();
         String texte1 = "15,30,100,20";
         String texte2 = "125,30,100,20";
         texte1 = texte1.replaceFirst("30",String.valueOf(30*(i+1)));
         texte2 = texte2.replaceFirst("30",String.valueOf(30*(i+1)));
         pan.add(texte1,lab[i]);
         pan.add(texte2,champs[i]);
      }
      champs[3].setVisible(false); //Ceci cache une zone de texte mais permet de garder le même alignement
      creerChien = new JButton("Créer chien");
      creerHomme = new JButton("Créer homme");
      aUneQueue = new JCheckBox("Queue");
      estMarie = new JCheckBox("Marié");
      pan.add("250,60,135,20",creerChien);
      pan.add("250,180,135,20",creerHomme);
      pan.add("150,120,100,20",aUneQueue);
      pan.add("150,240,100,20",estMarie);
      co.add(pan);

      creerChien.addActionListener(this);
      creerHomme.addActionListener(this);
   }
   public void actionPerformed (ActionEvent ev) {
      if (ev.getSource() == creerChien) chien = new ClasseChien();
         else homme = new ClasseHomme();
   }
}
```

Les classes *ClasseHomme et ClasseChien :*

```
public class ClasseHomme {
   String nom, couleurYeux;
   int age;
   boolean isMarie;
   public ClasseHomme() {
      nom = "Steven";
      couleurYeux = "marron";
      age = 35;
      isMarie = true;
   }
}
```

```
public class ClasseChien {
   String nom, couleurYeux;
   int age;
   boolean hasQueue;
   public ClasseChien() {
      nom = "Snoopy";
      couleurYeux = "marron";
      age = 2;
      hasQueue = true;
   }
}
```

2. Héritage

Les objets chien et homme créés ont de nombreuses ressemblances. Un des avantages de la POO est la possibilité de gérer de telles similitudes, comme dans une hiérarchie. Cette possibilité s'appelle *héritage*. Quand une classe hérite d'une autre classe, la classe enfant hérite automatiquement de toutes les caractéristiques (variables) et du comportement (méthodes) de la classe parent.
L'héritage est toujours additif ; il n'est pas possible d'hériter d'une classe et de recevoir moins que ce que possède la classe parent.
Dans Java, l'héritage est géré avec le mot clé *extends*. Quand une classe hérite d'une autre classe, la classe enfant étend la classe parent.
Exemple : `public class ClasseChien extends ClasseMammifere { … }`

Les éléments communs aux hommes et aux chiens sont communs à tous les mammifères, ce qui permet de créer une classe *ClasseMammifere* pour gérer ces similitudes.
Nous pouvons ensuite supprimer les déclarations des éléments communs des classes *ClasseChien* et *ClasseHomme*, les déclarer dans la classe *ClasseMammifere*. Ensuite déclarer les sous-classes *ClasseChien* et *ClasseHomme* à partir de la classe *ClasseMammifere*.

Modifiez, en ce sens, le code source de l'application précédente.
Pour déclarer les sous-classes, lors du constructeur, il ne sera pas nécessaire de faire appel aux constructeur parent car celui-ci est directement instancié. Si un autre constructeur spécifique est créé (surcharge), et que l'on a besoin du constructeur parent, on l'appelle alors à l'aide de *super()*.

```
public class ClasseMammifere {
      String nom, couleurYeux;
      int age;
      public ClasseMammifere() {
         nom="";
         couleurYeux="Marron";
         age=0;
      }
}
```

Voilà donc le code source pour la classe parent *ClasseMammifere* et voyons ce que donne les classes *ClasseChien* et *ClasseHomme* dans ce cas.

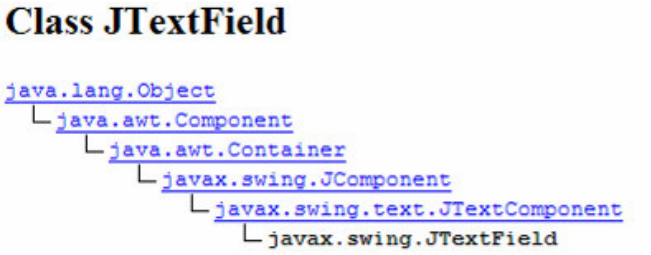
```
public class ClasseChien extends ClasseMammifere {
   boolean hasQueue;
   public ClasseChien() {
      nom="Snoopy";
      age=2;
      hasQueue=true;
   }
}
```

```
public class ClasseHomme extends Classe Mammifere {
   boolean isMarie;
   public ClasseHomme() {
      nom="Steven";
      couleurYeux="bleu";
      age=35;
      isMarie=true;
   }
}
```

Dans la classe *ClasseChien*, il n'est plus nécessaire d'attribuer une valeur à la variable *couleurYeux* car le chien possède la valeur définie par le constructeur parent. Ce qui n'est pas le cas pour l'homme. Il faut donc changer la valeur en lui attribuant la valeur voulue.
Dès qu'une classe étend une autre classe, elle hérite de toutes ces variables et de toutes ces méthodes.
Il est à savoir que toutes les classes héritent les variables et les méthodes de la classe *Object* lorsque aucune extension n'est prévue.

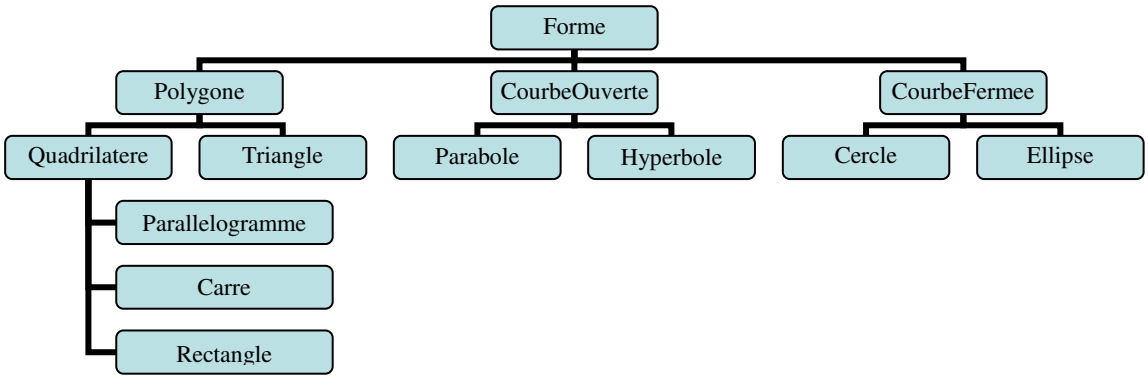
Dans Java, les classes ne peuvent hériter que d'une seule classe à la fois. Mais l'héritage peut être utilisé autant de fois que l'on veut dans la hiérarchie.
Ainsi une classe *ClasseChien*, héritera d'une classe *ClasseMammifere* qui elle-même hériterait d'une classe *ClasseAnimaux* qui pourrait aussi hériter d'une classe *ClasseEspecesVivantes* qui hériterait enfin d'une classe *ClasseObjet* qui elle-même dépend de la classe *Object*.

Voici une autre hiérarchie existante :



Cette hiérarchie peut encore se modifier, en y ajoutant soi-même une classe personnelle qui soit une extension de la classe *JTextField*.

Un autre cas :



Cet autre cas peut encore se complexifier ou se préciser.

Revenons à notre cas avec les mammifères, chiens et hommes.
La *ClasseMammifere* possède un constructeur qui définit des valeurs par défaut pratiques et appropriées. Il est intéressant que les sous-classes accèdent à ce constructeur.
Il existe deux façons pour y accéder.

- Si vous n'appelez pas explicitement le constructeur de la classe parent, Java appelle automatiquement le constructeur par défaut de la classe parent comme première ligne du constructeur de la classe enfant. La seule façon d'éviter ce comportement consiste à appeler un des constructeur de la classe parent comme première ligne du constructeur de la classe enfant.
  Les appels au constructeur sont toujours chaînés de cette façon et il n'est pas possible de contourner le mécanisme. Si vous désirez appeler le constructeur de la classe parent, le mot clé Java défini est *super()* qui appelle le constructeur sans paramètre de la classe parent.
- Si vous ne désirez pas « utiliser » les constructeurs de la classe parent, il faudra en première ligne du constructeur de la classe enfant en définir un et l'appeler. En réalité, l'objet est construit par le constructeur par défaut de la classe parent et ensuite redéfini par le constructeur de la classe enfant. C'est ce que l'on appelle la surcharge.

Après les modifications effectuées, il n'y a aucun changement dans le fonctionnement de l'application. De fait, le concept de base est changé pour une plus grande efficacité lors de mise à niveau ou amélioration apportées.

3. Modificateurs d'accès

Il est important de comprendre à quel moment les membres (variables et méthodes) de la classe sont accessibles. Dans Java, plusieurs options permettent de personnaliser l'accès aux membres.
De façon générale, il vaut mieux limiter autant que possible la portée des éléments d'un programme, y compris celle des membres des classes. Moins un élément est accessible, moins il risque d'être mal utilisé.
Il existe en Java, quatre modificateurs d'accès différents pour les membres des classes : *private, protected, public* et *default* (absence de modificateur).
Le fait que les classes d'un même paquet disposent d'accès différents par rapport aux classes situées en dehors du paquet ajoute un peu de complexité.
Voici deux tableaux montrant l'accessibilité et l'héritage des classes et variables membre depuis l'intérieur du même paquet et depuis l'extérieur du paquet.

Accès depuis l'intérieur du paquet d'une classe			Accès depuis l'extérieur du paquet d'une classe		
Modificateur d'accès	Héritage	Accessible	Modificateur d'accès	Héritage	Accessible
pas de modificateur	OUI	OUI	pas de modificateur	NON	NON
public	OUI	OUI	public	OUI	OUI
protected	OUI	OUI	protected	OUI	NON
private	NON	NON	private	NON	NON

Toutes variables ou méthodes devant rester interne à la classe auront comme accès *private*.
Par contre, les constructeurs doivent avoir un accès *public*.
En programmation orientée objet, il est recommandé de cacher les informations à l'intérieur de la classe en rendant privées toutes les variables membres de la classe, et en y accédant au moyen de méthodes, de format spécifique, appelés méthodes d'accès.

4. Mode d'accès

Les méthodes d'accès (parfois appelées getter et setter) fournissent l'interface publique de la classe accessible depuis l'extérieur, tout en conservant au stockage des données de la classe son caractère privé.
C'est une bonne idée puisque vous pourrez ensuite à tout moment modifier la représentation interne des données dans la classe sans toucher aux méthodes qui définissent réellement ces valeurs internes. Tant que vous ne modifiez pas l'interface publique d'accès à la classe, vous ne détruisez aucun code qui repose sur cette classe et ses méthodes publiques.
Dans Java, les méthodes d'accès sont fournies par paires : une pour obtenir la valeur interne (*get*) et une autre pour définir la valeur interne (*set*).
Par convention, la méthode *get* utilise le nom de la variable interne privée associée au préfixe ″get″. La méthode *set* fait de même avec le préfixe ″set″.
En général, lorsque la valeur est de type booléenne, on utilise ″is″ ou ″has″ à la place de ″get″.
Adaptez le code source de telle façon que les variables des classes *ClasseChien*, *ClasseHomme* et *ClasseMammifere* soient *private* mais que l'on puisse obtenir et changer leur valeur.

```
public class ClasseChien extends ClasseMammifere {
   public boolean hasQueue() {
      return queue;
   }
   public void setQueue(boolean valeur) {
      queue = valeur;
   }
   public ClasseChien() {
      setNom("Snoopy");
      setAge(2);
      setQueue(true);
   }
   private boolean queue;
}

public class ClasseHomme extends ClasseMammifere{
   public boolean isMarie() {
      return marie;
   }
   public void setMarie(boolean valeur) {
      marie = valeur;
   }
   public ClasseHomme() {
      setNom("Steven");
      setCouleurYeux("bleu");
      setAge(35);
      setMarie(true);
   }
   private boolean marie;
}

public class ClasseMammifere {
   public String getNom () {
      return nom;
   }
   public String getCouleurYeux() {
      return couleurYeux;
   }
   public String getSon() {
      return son;
   }
```

```
    public int getAge() {
        return age;
    }
    public void setNom(String valeur) {
        nom = valeur;
    }
    public void setCouleurYeux(String valeur) {
        couleurYeux = valeur;
    }
    public void setAge(int valeur) {
        age = (valeur > 0 ? valeur : 0);
    }
    public void setSon(String valeur) {
        son = valeur;
    }
    public ClasseMammifere() {
        nom="Le nom";
        couleurYeux="Marron";
        age=0;
    }
    private String nom, couleurYeux, son;
    private int age;
}
```

Il convient de déplacer les variables *private* en fin de classe car comme elles ne peuvent être utilisées ailleurs que dans la classe donc peuvent être omise pour la compréhension du code.

5. Classes abstraites

Dans une classe, il est possible de déclarer une méthode comme étant *abstract*, ce qui signifie qu'il n'y a pas d'implémentation de la méthode dans cette classe, mais que toutes les classes qui étendent celles-ci, doivent fournir une implémentation.
Par exemple, supposons que tous les mammifères doivent préciser leur vitesse de course maximale et que chaque mammifère puisse indiquer une vitesse différente.
Il faut créer une méthode abstraite *vitesse()* dans la classe *ClasseMammifere*.
Lorsqu'une classe contient une méthode abstraite, la totalité de la classe doit être déclarée comme abstraite. Ainsi cette classe ne pourra plus avoir de méthodes qui sont instanciée.
A présent, il faut implémenter la méthode *vitesse()* à toutes les classes étendant la classe *ClasseMammifere*.
Faite de sorte qu'il y ait une boîte de dialogue qui donne lors de la création de l'objet la vitesse de l'objet.

```
abstract public class ClasseMammifere {
   public String getNom () {
      return nom;
   }
   public String getCouleurYeux() {
      return couleurYeux;
   }
   public String getSon() {
      return son;
   }
   public int getAge() {
      return age;
   }
   public void setNom(String valeur) {
      nom = valeur;
   }
   public void setCouleurYeux(String valeur) {
      couleurYeux = valeur;
   }
   public void setAge(int valeur) {
      age = (valeur > 0 ? valeur : 0);
   }
   public void setSon(String valeur) {
      son = valeur;
   }
   public ClasseMammifere() {
      nom="Le nom";
      couleurYeux="Marron";
      age=0;
   }
   abstract public void vitesse();
   private String nom, couleurYeux, son;
   private int age;
}

import javax.swing.*;
public class ClasseHomme extends ClasseMammifere{
   public boolean isMarie() {
      return marie;
   }
   public void setMarie(boolean valeur) {
      marie = valeur;
   }
   public ClasseHomme() {
      setNom("Steven");
      setCouleurYeux("bleu");
```

```
        setAge(35);
        setMarie(true);
    }
    public void vitesse() {
        JOptionPane.showMessageDialog(null,"30 km/h","Vitesse de l'homme",1);
    }
    private boolean marie;
}

import javax.swing.*;
public class ClasseChien extends ClasseMammifere {
    public boolean hasQueue() {
        return queue;
    }
    public void setQueue(boolean valeur) {
        queue = valeur;
    }
    public ClasseChien() {
        setNom("Snoopy");
        setAge(2);
        setQueue(true);
    }
    public void vitesse() {
        JOptionPane.showMessageDialog(null,"50 km/h","Vitesse du chien",1);
    }
    private boolean queue;
}
```

6. Polymorphisme

Le polymorphisme est la possibilité pour deux classes séparées, mais reliées, de recevoir le même message mais d'agir dessus de différentes façons. Cela permet de manipuler des objets sans en connaître tout à fait le type. Ainsi deux classes différentes peuvent avoir la même méthode mais l'implémenter de façon différente.
A l'aide de cette notion, le développeur s'occupe des généralités en laissant l'environnement d'exécution s'occuper des spécificités.

Dans notre exemple, étant difficile d'intégrer tel quel le mécanisme de polymorphisme, voici un exemple extrême pour permettre de mieux comprendre ce mécanisme.

En reprenant l'arbre de la page 66, que va imprimer le programme qui suit ?

```
public class Hierarchie {
   public static void main (String [] args) {
      Parallelogrammes p = new Parallelogrammes();
      Rectangles r = new Rectangles();
      Carres c = new Carres();
      Quadrilateres q = new Quadrilateres();
      Triangles t = new Triangles();
      Polygones g = new Polygones();
      Formes f = new Formes();
      p.seDefinir();
      r.seDefinir();
      c.seDefinir();
      q.seDefinir();
      t.seDefinir();
      g.seDefinir();
      f.seDefinir();
      System.out.println();
      f = c;
      g = r;
      q = p;
      f.seDefinir();
      g.seDefinir();
      q.seDefinir();
   }
}

class Formes {
   public void seDefinir() {
      System.out.println("Je suis une forme.");
   }
}

class Polygones extends Formes {
   public void seDefinir() {
      super.seDefinir();
      System.out.println("Je suis aussi un polygone.");
   }
}

class Triangles extends Polygones {
}

class Quadrilateres extends Polygones {
}

class Carres extends Quadrilateres {
   public void seDefinir() {
      System.out.println("Je suis un carre.");
   }
}
```

```
class Rectangles extends Quadrilateres {
   public void seDefinir() {
      super.seDefinir();
      System.out.println("Je suis aussi un rectangle.");
   }
}

class Parallelogrammes extends Quadrilateres {
}
```

Résultats :

p.seDefinir(); : .....
r.seDefinir(); : .....
c.seDefinir(); : .....
q.seDefinir(); : .....
t.seDefinir(); : .....
g.seDefinir(); : .....
f.seDefinir(); : .....
f.seDefinir(); : .....
g.seDefinir(); : .....
q.seDefinir(); : .....

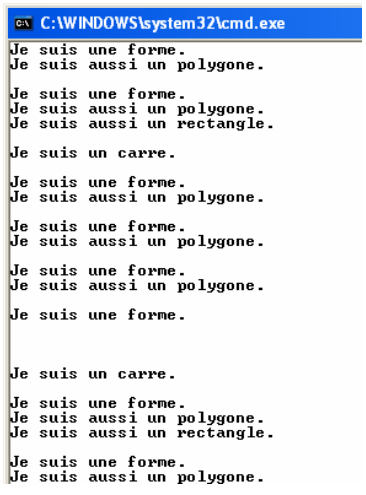
Voici un exemple extrême qui combine le polymorphisme et l'héritage.
L'importance de la signature de la méthode et de la création de l'objet.

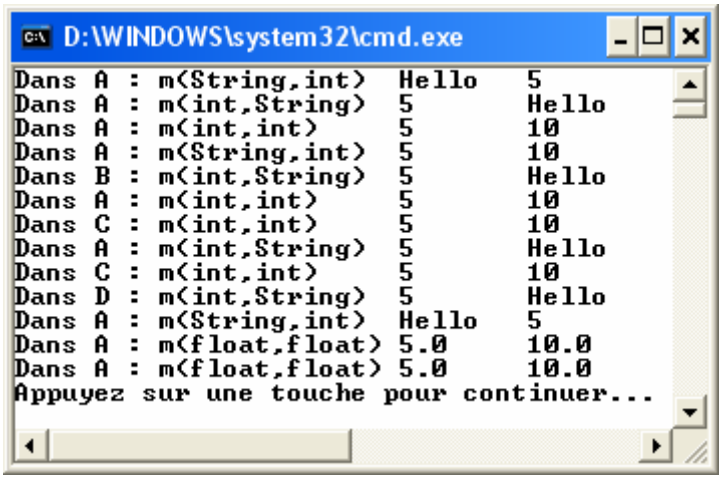
```
public class Poly {
   public static void main (String [] args) {
      A a = new A();
      A b = new B();
      A c1 = new C();
      C c2 = new C();
      B d1 = new D();
      A d2 = new D();
      a.m("Hello",5);
      a.m(5,"Hello");
      a.m(5,10);
      a.m("5",10);
      b.m(5,"Hello");
      b.m(5,10);
      c1.m(5,10);
      c1.m(5,"Hello");
      c2.m(5,10);
      d1.m(5,"Hello");
      d2.m("Hello",5);
      d2.m(5,10.f);
      d2.m(5.f,10);
   }
}
class A {
   void m(String s, int i) {
      System.out.println("Dans A : m(String,int)\t" + s + "\t" + i);
   }
   void m(int i, String s) {
      System.out.println("Dans A : m(int,String)\t" + i + "\t" + s);
   }
   void m(int i1, int i2) {
      System.out.println("Dans A : m(int,int)\t" + i1 + "\t" + i2);
   }
```

```
    void m(float f1, float f2) {
        System.out.println("Dans A : m(float,float)\t" + f1 + "\t" + f2);
    }
}
class B extends A {
    void m(int i, String s) {
        System.out.println("Dans B : m(int,String)\t" + i + "\t" + s);
    }
    void m(float f, int i) {
        System.out.println("Dans B : m(float,int)\t" + f + "\t" + i);
    }
}
class C extends A {
    void m(int i1, int i2) {
        System.out.println("Dans C : m(int,int)\t" + i1 + "\t" + i2);
    }
}
class D extends B {
    void m(int i1, int i2) {
        System.out.println("Dans D : m(int,int)\t" + i1 + "\t" + i2);
    }
    void m(int i, String s) {
        System.out.println("Dans D : m(int,String)\t" + i + "\t" + s);
    }
}
```

Résultats :

a.m("Hello",5); : .....
a.m(5,"Hello"); : .....
a.m(5,10); : .....
a.m("5",10); : .....
b.m(5,"Hello"); : .....
b.m(5,10); : .....
c1.m(5,10); ..... :
c1.m(5,"Hello"); ..... :
c2.m(5,10); : .....
d1.m(5,"Hello"); : .....
d2.m("Hello",5); : .....
d2.m(5,10.f); : .....
d2.m(5.f,10); : .....





7. Utilisation des interfaces

Une interface ressemble à une classe abstraite mais avec un différence importante : une interface ne peut inclure de code. Dans Java, le mécanisme d'interface est un moyen destiné à remplacer l'héritage multiple qui n'existe pas.
Une interface est une déclaration de classe spécialisée qui peut déclarer des constantes, des déclarations de méthodes, mais par leur implémentation.
Le gros avantage se trouve dans le fait qu'une classe peut implémenter plusieurs interfaces.
Lorsqu'une interface est implémentée par une classe, cette classe **DOIT** définir les méthodes de l'interface.
Dans notre exemple de départ, nous pouvons créer une interface *InterfaceSon* qui possède une méthode *dit()*.
Le code s'écrit comme suit :

```
public interface InterfaceSon {
   public void dit() {
   }
}
```

Alors la classe *ClasseMammifere* qui implémente cette interface doit posséder la définition de la méthode *dit()*.
Par exemple :

```
public void dit() {
   JOptionPane.showMessageDialog(null,this.getSon(),this.getNom() + ″Dit″, 1) ;
}
```

Il faut aussi ajouter une ligne dans les classes *ClasseChien* et *ClasseHomme* pour que le tout donne le résultat escompté.
Dans la classe *ClasseChien*, dans la méthode de création : `setSon(″Wouah, Wouah !″) ;` et dans la classe *ClasseHomme*, dans la méthode de création : `setSon(″Bonjour à tous ! Je suis ″ + this.getNom() + ″.″) ;`

8. Exercice

Créer une interface permettant de choisir un marque, un style, une couleur et si nécessaire le nombre de portes, si le choix est citadine ou familiale.
Dans les autres cas : sportive = 3 et mono volume = 5.

On utilisera des listes, boutons radio et un champ texte pour les différents éléments.

Le véhicule ainsi créé peut donner plusieurs caractéristiques : vitesse maxi pour les sportives, nombre de place pour les mono volumes, la couleur, la marque et le style pour toutes.
C'est qui doit être affiché dans le panneau de droite en même temps qu'une photo représentant la marque et le style de la voiture.

Les informations manquantes sont données par des boîtes de dialogues avec liste déroulante.

Pour bien voir que vous avez utilisé les principes de la POO (polymorphisme et héritage), il faut que dans chacune des classes fils, un texte apparaît à la fenêtre console qui donne les mêmes caractéristiques que le panneau droit.

Votre projet est sauvegardé sous le nom « ChoixVehicule.java », pour les autres aucune imposition n'est faite.

Le résultat désiré dans le cas de figure de la fenêtre console :

Délai de réalisation : vendredi 20 mai 2005.

NB :
La taille de la fenêtre est de 600 X 450, le panneau est de 400 X 420.
La taille des photos est prise en fonction de leur état initial.
La taille de la police est 20.
La fenêtre ne peut être redimensionnée par l'utilisateur.
Les marques sont : BMW, Fiat, Mazda, Mercedes, Peugeot, Renault, Toyota et VW.
Les styles sont : citadine, familiale, sportive, mono volume.
Les couleurs : rouge, bleu, noir.
Les vitesses maxi pour chaque marque : 280.3, 282.1, 275.0, 259.8, 267.4, 258.2, 264.3 et 249.0
Les places disponibles pour chaque marque : 5, 6, 5, 5, 8, 8, 7 et 8.
Les images, vous pouvez les télécharger ou d'autres, des sites officiels des constructeurs.
Ces données peuvent être changées par les vraies valeurs si vous avez l'occasion de chercher.

Nous avons construit jusqu'à maintenant des applications autonomes, nous allons voir maintenant comment construire des applets. Une applet est téléchargée sur une machine donnée à partir d'une machine distante qui en fournit le code. Ce chargement est toujours provoqué par l'analyse d'un fichier contenant des commandes HTML.
Tout ce qui a été vu jusqu'à présent est utilisable et il faut tenir compte de certaines particularités inhérentes à la manière dont une applet est lancée, ainsi que d'éventuelles restrictions d'accès.

1. Première applet

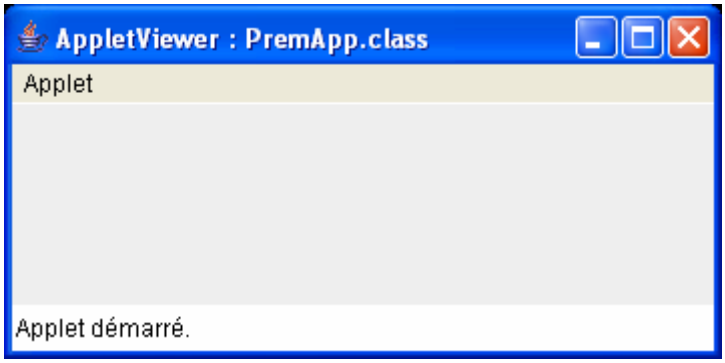
Une applet a beaucoup de points communs avec une application qui crée une fenêtre graphique.
Une applet est obligatoirement constituée d'une classe dérivée de *JApplet*, laquelle est un conteneur de plus niveau. Au lancement d'une applet, on dispose automatiquement d'une fenêtre graphique dont les dimensions sont définies par les commandes ou balises HTML du fichier lançant l'applet.
Voici les paramètres nécessaires dans un fichier HTML pour qu'une applet puisse être lancée :

```
<HTML>
   <BODY>
      <APPLET CODE = ″PremApp.class″ WIDTH = 350 HEIGHT = 100>
      </APPLET>
   </BODY>
</HTML>
```

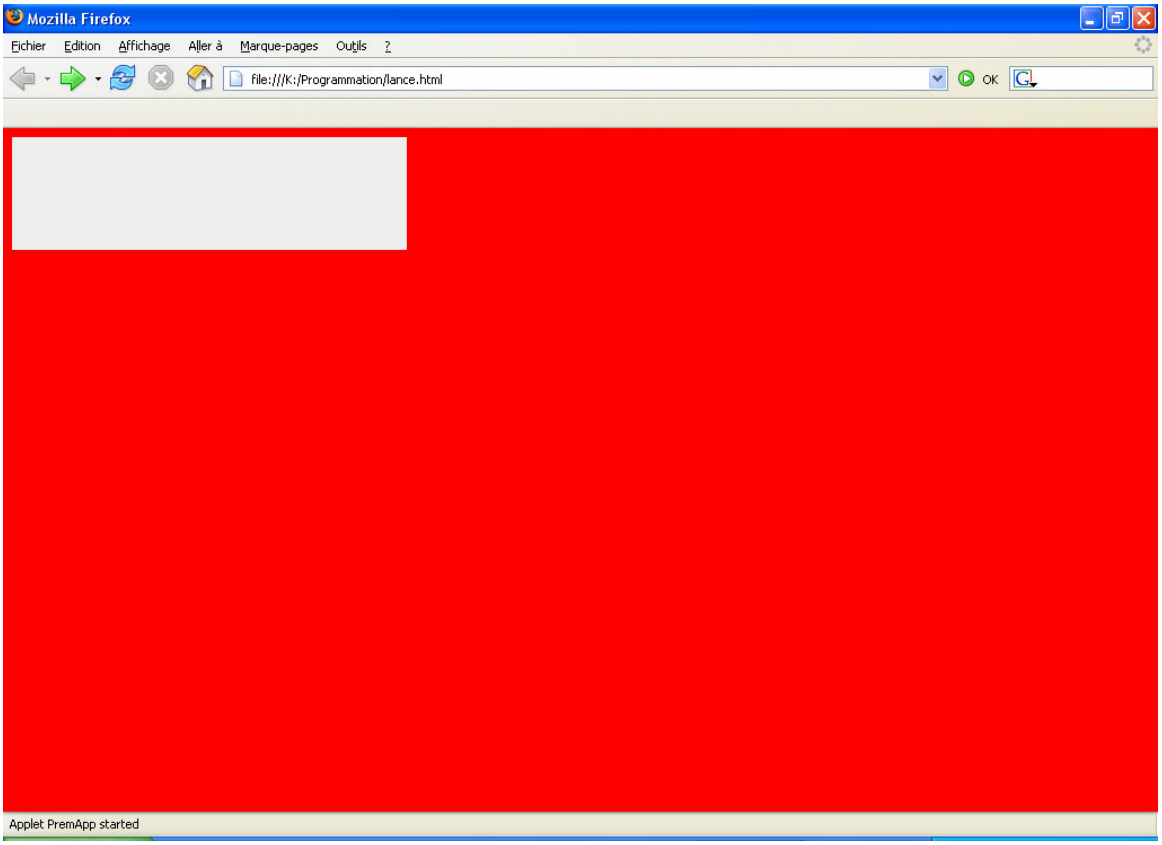
Quant à l'applet elle-même voici un exemple très simple d'une fenêtre sans rien :

```
import javax.swing.* ;
public class PremApp extends JApplet {
   public PremApp () {
   }
}
```

Le résultat obtenu est à l'aide d'une application qui permet de visualiser des applets :



Dans une page WEB, le résultat est le suivant :



2. La méthode init

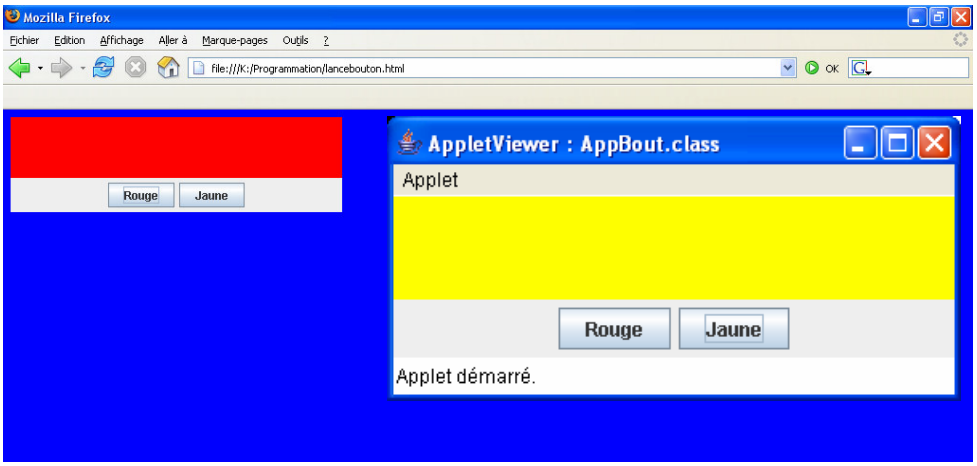
Notre premier exemple n'a fait que créer un conteneur vide, il va de soi qu'on peut associer à ce conteneur des écouteurs d'événements et y introduire des composants.
La méthode *init* est exécutée automatiquement lors du lancement de l'applet.
Cette méthode est l'équivalente de la méthode *main* dans les applications Java.
Contrairement à la méthode *main* pour laquelle, très peu de chose était écrite, on construit le maximum d'objets dans la méthode *init*.
Cette démarche peut se révéler indispensable pour certaines opérations telles que la récupération au sein de l'applet d'informations en provenance du fichier HTML.

Voici un exemple qui permet à l'aide de deux boutons de changer la couleur du panneau en rouge ou en jaune.

Le code pour le fichier HTML :

```
<HTML>
   <BODY>
      <APPLET CODE = ″AppBout.class″ WIDTH = 350 HEIGHT = 100>
      </APPLET>
   </BODY>
</HTML>
```

Le code pour le fichier Java :




```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
public class AppBout extends JApplet
implements ActionListener {
   private JPanel pan1,pan2;
   private JButton rouge,jaune;
   public void init() {
   pan1 = new JPanel();
   pan2 = new JPanel();
   Container co = getContentPane();
   rouge = new JButton("Rouge");
   jaune = new JButton("Jaune");
   rouge.addActionListener(this);
   jaune.addActionListener(this);
   pan2.add(rouge);
   pan2.add(jaune);
   co.add(pan1);
   co.add(pan2,BorderLayout.SOUTH);
   }
   public void actionPerformed (ActionEvent e) {
      if(e.getSource() == rouge) pan1.setBackground(Color.RED);
         else pan1.setBackground(Color.YELLOW);
   }
}
```

3. Différents stades de la vie d'une applet

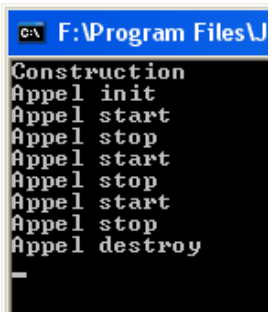
La méthode *init* est appelée après la création de l'objet applet. Il existe une méthode nommée *destroy* qui est appelée au moment où l'applet se termine, c'est-à-dire lorsque l'utilisateur quitte le navigateur ou lorsqu'il ferme la fenêtre correspondante lorsque l'applet a été lancée par un visualisateur d'applets.
Il se peut que l'utilisateur rende inactive la fenêtre correspondante à l'applet en naviguant sur la page Web, il est donc intéressant lors d'un dessin évolutif que l'applet cesse lorsque l'on ne la voit plus et reprenne par la suite.
C'est pourquoi Java prévoit deux autres méthodes : *start* et *stop* pour rendre à nouveau visible une applet ou la rendre invisible.

init : après la création de l'objet applet ;
start : après *init*, puis chaque fois que l'applet redevient visible ;
stop : chaque fois que l'applet n'est plus visible, ainsi qu'avant *destroy* ;
destroy : à la fin de l'exécution de l'applet.

Il faut que le programmeur redéfinisse les méthodes *start* et *stop* pour qu'elles fassent exactement ce qu'il veut.

```
import javax.swing.*;
public class Etats extends JApplet {
   public Etats () {
      System.out.println("Construction");
   }
   public void init() {
      System.out.println("Appel init");
   }
   public void start() {
      System.out.println("Appel start");
   }
   public void stop() {
      System.out.println("Appel stop");
   }
   public void destroy() {
      System.out.println("Appel destroy");
   }
}
```

```
<HTML>
<BODY>
<APPLET CODE = "Etats.class" WIDTH = 350 HEIGHT = 100>
</APPLET>
</BODY>
</HTML>
```



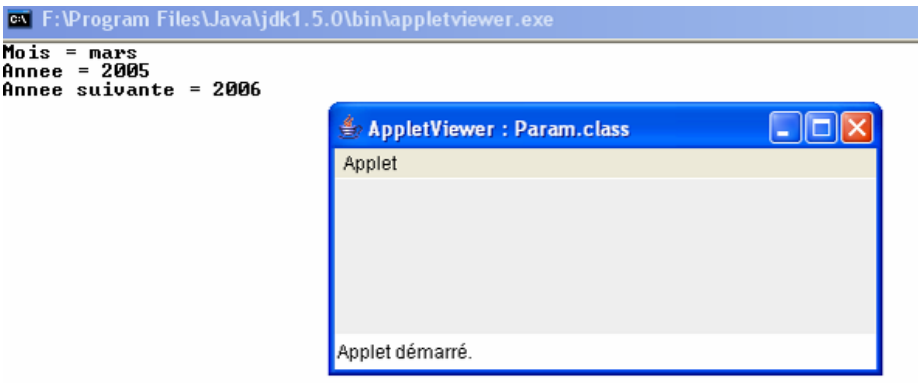
4. Transmission d'information à une applet

Il est possible de transmettre des informations à une applet par le biais de commande appropriées dans le fichier HTML. Chaque information est identifiée par un nom et une valeur :
<PARAM NAME = ″mois″ VALUE = ″mars″>
Ainsi dans cet exemple, on attribue au paramètre de nom *mois*, la valeur *mars*.
Dans l'applet, il faut utiliser la méthode *getParameter(String s)* qui récupère la valeur du nom qui est écrit comme paramètre de la méthode.
Le résultat est toujours un objet de type *String*.
Les valeurs fournies en paramètre WIDTH et HEIGHT sont aussi récupérables comme n'importe quelles autres informations.

Voici un exemple de récupération de données et de transformation :

```
<HTML>
<BODY bgcolor="RED">
<APPLET CODE = "Param.class" WIDTH = 350 HEIGHT = 100>
<PARAM NAME = "mois" VALUE = "mars">
<PARAM NAME = "annee" VALUE = "2005">
</APPLET>
</BODY>
</HTML>
```

```
import javax.swing.*;
public class Param extends JApplet {
   public void init() {
      String nomMois = getParameter("mois");
      String nomAnnee = getParameter("annee");
      System.out.println("Mois = " + nomMois);
      System.out.println("Annee = " + nomAnnee);
      System.out.println("Annee suivante = " + (Integer.parseInt(nomAnnee)+1));
   }
}
```



5. Restrictions imposées aux applets

Les applets ont été conçues à l'origine pour être exécutées sur un site distant de celui qui en fournit le code. Dans ces conditions, les concepteurs de Java avaient prévu des restrictions nécessaires pour assurer une sécurité absolue sur la machine d'exécution.
En particulier, la machine virtuelle interdisait à une applet :

- d'accéder aux fichiers locaux ;
- de lancer un programme exécutable local ;
- d'obtenir des informations relatives au système local.

Toute tentative d'opération de type provoquait une exception *SecurityException*.
Avec le temps et avec la généralisation de l'utilisation des applets au sein de ce que l'on nomme *intranet*, ces restrictions sont apparues trop sévères. Aujourd'hui, la notion de *gestionnaire de sécurité* permet à un environnement de définir les opérations qu'il autorise les applets à faire. Par ailleurs, la notion d'*applet certifiée* permet d'accorder des permissions supplémentaires à une applet pour laquelle on dispose d'une garantie d'origine déterminée.

6. Création d'une applet (horloge)

Cahier des charges :

Objet : Création d'une horloge analogique et affichage du jour, la date et l'heure en numérique fonctionnant sur une page Web.

Equipe de travail : Classe et professeur de la 6ème informatique.

Evaluation d'un temps de travail : 5 heures de cours.

Chargé des tests et maintenance future : le professeur.

Divers : Le produit doit pouvoir afficher la date, l'heure du jour système.
En cas de changement de date ou heure système, le produit pourra être redémarré si nécessaire. On n'attend pas qu'il se réajuste automatiquement. (possible lors d'une seconde phase)
Par contre, lors de l'affichage des heures, minutes et secondes en numérique, celles-ci devront apparaître toujours sous la forme de 2 caractères et sous le format 24 heures.
Concernant le fonctionnement, il faudra soit que les utilisateurs aient la version Java JRE 1.5, soit que le projet sera compilé en version 1.4.2 version standard maintenant.

Fonctionnalité : Toutes les secondes, les aiguilles doivent bouger en fonction de l'heure système.

Schéma de travail :
1°) Création de l'interface et de la liaison avec la page Web.
2°) Définition des méthodes *init*, *start* et *stop* de la classe étendant *JApplet*.
3°) Création des panneaux *Cadran* et *Aiguilles* qui servent de fond (présentation) et de fonctionnement.
4°) Définition de la méthode *paint* dans chacune des classes prcédentes pour obtenir ce que l'on désire.
5°) Création de méthodes permettant d'attribuer des valeurs spécifiques ne se transmettant pas entre les classes *Horloge, Cadran* et *Aiguilles*.

Problématiques :
1°) Connaître le jour, la date et l'heure système et transmettre ces informations à la classe *Aguilles*.
2°) Afficher les heures numériques dans le format 24 heures et avec 2 caractères.
3°) Travailler avec des segments que l'on définit seconde par seconde ou que l'on applique une rotation.
4°) Effectuer la mise à jour, toutes les secondes sans utiliser des *Thread* (recherche de classes permettant ce système)
5°) Différencier l'épaisseur des traits
6°) Superposer les deux panneaux mais surtout que le second ne cache pas le premier.

6.1. Création de l'interface, de la liaison avec la page Web et mise en route

6.1.1. La liaison avec la page Web

Pour créer le fichier qui permet de lancer l'applet sur une page Web, il suffit d'insérer à l'endroit voulu le code suivant :

```
<APPLET CODE = "Horloge.class" WIDTH = 400 HEIGHT = 400></APPLET>
```

« code » permet de définir le fichier qui est lancé, « width » et « height » permettent de définir la taille de la fenêtre qui lance l'applet.

6.1.2. Définition des méthodes *init*, *start* et *stop* de la classe étendant *JApplet*.

La classe *Horloge.java* est conçue comme suit :

```
public class Horloge extends JApplet {
   public void init() { }
   public void start() { }
   public void stop () { }
}
```

Il faut définir ce que doit faire *init*, *start* et *stop* et définir les champs dont nous avons besoin.

a) On va utiliser dans cette classe des objets particuliers : *Timer* qui provient de *Java.util* et la variable associée s'appelle dans notre cas, minuterie.
   Cette classe permet de définir une tâche à effectuer à l'aide de la méthode
   `minuterie.scheduleAtFixedRate(TimerTask task, Date firstTime, long period);`
   Si la valeur de *fistTime* est 0, cela veut dire que la tâche commence tout de suite.
   Dans notre cas, pour la variable *period*, nous désirons que la mise à jour se fasse toutes les secondes, nous écrivons donc la valeur 1000 qui correspond à 1000 millisecondes qui est 1 seconde. Il faut donc créer un objet de type *TimerTask* avec la redéfinition de la méthode *run*.

b) Les champs d'entier *sec*, *min*, *h*, *periode*, *journee*, *jour*, *mois*, *annee* qui servent à donner à un instant donné, les secondes, les minutes, les heures (au format 12 heures), les heures (au format 24 heures), le jour de la semaine (valeur entière à convertir), le jour du mois, le mois et l'année.
   Aiguilles aiguilles et *Cadran montre* déclarent deux objets de type *Aiguilles* et *Cadran* qui sont des classes étendant la classe *JPanel* pour la représentation de l'horloge.

c) La méthode *init()* :

```
Dimension size = getSize();
int diametreCadran = Math.min(size.width,size.height*5/6)*9/10;
montre = new Cadran(diametreCadran);
getContentPane().add(montre);
aiguilles = new Aiguilles(diametreCadran);
setGlassPane(aiguilles);
aiguilles.setVisible(true);
```

   Il est nécessaire de connaître la taille de la fenêtre définie dans le fichier HTML, c'est ce que fait la méthode *getSize()*.
   Il faut définir une taille du cercle qui sert d'horloge, on prend le minimum de la longueur et hauteur*5/6 définie et ensuite les 9/10 pour la présentation.
   On crée un panneau *montre* de type *Cadran* avec cette valeur minimale, l'on l'ajoute au containeur de la fenêtre.
   On crée finalement le panneau *aiguilles* de type *Aiguilles* avec la valeur minimale et on utilise la méthode *setGlassPane* qui permet de superposer les deux panneaux.

d) La méthode *start()* :
Dans cette méthode, on a besoin d'un objet de type *Calendar* pour obtenir heure et date système avec la méthode *getInstance()*.
C'est dans cette méthode que l'on définit les valeurs des champs d'entier et cela à l'aide des champs définis dans un objet de type *Calendar*.

```
journee = now.get(now.DAY_OF_WEEK);
periode = now.get(now.HOUR_OF_DAY);
jour = now.get(now.DATE);
mois = now.get(now.MONTH);
annee = now.get(now.YEAR);
sec = now.get(now.SECOND);
min = now.get(now.MINUTE);
h = now.get(now.HOUR);
```

C'est aussi dans cette méthode que l'on utilise l'objet de type *Timer* définit dans la classe. Il faut alors définir la méthode *run()* :

```
increment();
aiguilles.setSeconde(sec);
aiguilles.setMinute(min);
aiguilles.setHeure(h);
aiguilles.setJour(jour);
aiguilles.setMois(mois);
aiguilles.setAnnee(annee);
aiguilles.setPeriode(periode);
aiguilles.setJournee(journee);
aiguilles.repaint();
```

On crée des méthodes qui permettent d'incrémenter d'une unité pour avoir le défilement. Par contre, les autres méthodes permettent d'attribuer aux champs de l'objet *aiguilles* les valeurs correspondantes de l'objet de la classe *Horloge*.
Quant à la dernière, elle permet de redessiner le tout.

e) La méthode *Stop()*
Cette méthode ne comporte q'une instruction, l'instanciation de l'objet *aiguilles* à la méthode *cancel()* qui permet d'arrêter la tâche définie.

```
minuterie.cancel();
```

6.1.3. Création des panneaux *Cadran* et *Aiguilles*

Ces panneaux sont des panneaux personnalisés qui étendent la classe *JPanel* et qui avec la méthode *paint(Graphics g)* redéfinie donnera notre interface.

La classe *Cadran* qui sert de fond :

```
int diametre;
Ellipse2D.Double face;
Line2D.Double marqueHoraire;
BasicStroke traitEpais = new BasicStroke(3.0f,BasicStroke.CAP_ROUND,BasicStroke.JOIN_MITER);
BasicStroke traitFin = new BasicStroke(1.0f,BasicStroke.CAP_ROUND,BasicStroke.JOIN_MITER);
final Color CLEAR = new Color(0,0,0,0);
public Cadran(int diametre) {
      this.diametre = diametre;
      face = new Ellipse2D.Double();
      marqueHoraire = new Line2D.Double(0, -diametre*0.4,0,-diametre*0.5);
      setOpaque(false);
}
```

On utilise des objets des classes *Ellipse2D.Double*, *Line2D.Double* pour avoir la possibilité de choisir les épaisseurs des traits à l'aide de la classe *BasicStroke*.

Avec des objets de type *Graphics*, il n'est pas possible de définir une épaisseur mais bien pour des objets de type *Graphics2D*.
La méthode *setOpaque(false)* permet de rendre transparent le panneau pour qu'il soit superposable.

La classe *Aiguilles* qui sert pour les aiguilles et est donc l'avant plan :

```
Line2D.Double aiguillesDesHeures;
Line2D.Double aiguillesDesMinutes;
Line2D.Double aiguillesDesSecondes;
Ellipse2D.Double centre;
int seconde,minute,heure,periode,jour,mois,annee;
String journee;
final Color CLEAR = new Color(0,0,0,0);
BasicStroke traitEpais = new BasicStroke(3.0f,BasicStroke.CAP_ROUND,BasicStroke.JOIN_MITER);
BasicStroke traitFin = new BasicStroke(1.0f,BasicStroke.CAP_ROUND,BasicStroke.JOIN_MITER);
public Aiguilles(int diametre) {
      centre = new Ellipse2D.Double(-3,-3,6,6);
      aiguillesDesHeures = new Line2D.Double(0,6,0,-diametre*0.25);
      aiguillesDesMinutes = new Line2D.Double(0,8,0,-diametre*0.3);
      aiguillesDesSecondes = new Line2D.Double(0,14,0,-diametre*0.35);
      setOpaque(false);
}
```

Cette classe est similaire à la classe *Cadran* mais les champs sont différents.

6.1.4. Définition de la méthode *paint*

Pour la classe *Cadran* :

```
public void paint(Graphics g) {
   Dimension size = getSize();
   face.setFrame((size.width-diametre)/2,(size.height-diametre)/2,diametre,diametre);
   Graphics2D g2D = (Graphics2D)g;
   g2D.setPaint(CLEAR);
   g2D.fillRect(0,0,size.width,size.height);
   g2D.setPaint(Color.lightGray);
   g2D.fill(face);
   g2D.setPaint(Color.darkGray);
   g2D.setStroke(traitEpais);
   g2D.draw(face);
   g2D.translate(size.width/2,size.height/2);
   for (int i = 0 ; i < 12 ; i++) {
      if(i%3 == 0) g2D.setStroke(traitEpais);
         else g2D.setStroke(traitFin);
      g2D.draw(marqueHoraire);
      g2D.rotate(2.0*Math.PI/12.0);
   }
}
```

Pour dimensionner le panneau, on a besoin de connaître la longueur et hauteur de la fenêtre définie dans le fichier HTML.
Ensuite on crée l'objet *g2D* de type *Graphics2D* et on dessine les éléments voulus dans un ordre spécifique pour les couleurs et les épaisseurs des traits.
La méthode *rotate(double valeur)* permet d'ajouter (concaténer) un élément en effectuant une rotation de l'objet qui l'instancie.

Pour la classe *Aiguilles* :

```
public void paint(Graphics g) {
    double angleDesSecondes = seconde * 2.0*Math.PI / 60;
    double angleDesMinutes = (angleDesSecondes + minute * 2.0*Math.PI)/60;
    double angleDesHeures = (angleDesMinutes + heure * 2.0*Math.PI)/12;
    Dimension size = getSize();
    Graphics2D g2D = (Graphics2D)g;
    g2D.setPaint(CLEAR);
    g2D.fillRect(0,0,size.width,size.height);
    g2D.setPaint(Color.darkGray);
    g2D.translate(size.width/2,size.height/2);
    g2D.setFont(new Font("Dialog", 1, 18));
    Color couleur = new Color(0,0,255);
    g2D.setPaint(couleur.darker().darker());
    String heureAffiche = (periode < 10 ?  "0"+Integer.toString(periode) : Integer.toString(periode));
    String minuteAffiche = (minute < 10 ?  "0"+Integer.toString(minute) : Integer.toString(minute));
    String secondeAffiche = (seconde < 10 ?  "0"+Integer.toString(seconde) : Integer.toString(seconde));
    String heureAfficheFinale =journee+" "+jour+"/"+mois+"/"+annee+" : "+heureAffiche+" H "+minuteAffiche+" M " + secondeAffiche+" S";
    g2D.drawString(heureAfficheFinale,-size.width/2.45f,size.height*7/16);
    g2D.setPaint(Color.darkGray);
    AffineTransform transform = g2D.getTransform();
    g2D.setStroke(traitEpais);
    g2D.rotate(angleDesHeures);
    g2D.draw(aiguillesDesHeures);
    g2D.setTransform(transform);
    g2D.rotate(angleDesMinutes);
    g2D.draw(aiguillesDesMinutes);
    g2D.setStroke(traitFin);
    g2D.setTransform(transform);
    g2D.rotate(angleDesSecondes);
    g2D.draw(aiguillesDesSecondes);
    g2D.setPaint(Color.white);
    g2D.draw(centre);
}
```

On fonctionne dans le même principe pour cette méthode mais à la différence que l'on a besoin d'une copie pour effectuer les rotations, c'est pourquoi on utilise une classe *AffineTransform* qui permet d'obtenir une copie de l'objet *g2D*.
Dans cette méthode, il faut aussi prévoir l'écriture du jour, de la date et de l'heure en numérique.
Ainsi des variables de type *String* sont créées avec un test pour savoir si la valeur est strictement plus petite que 10 alors, on ajoute un 0.
Pour « dessiner » du texte avec les nombres, il faut les convertir, c'est ce que fait la méthode *toString(int valeur)*.

6.1.5. Création de méthodes « setter » dans la classe *Aiguilles*

C'est à l'aide de ces méthodes, que la méthode *paint(Graphics g)* pourra connaître exactement les différentes valeurs dont elle a besoin. Ces valeurs provenant de la classe *Horloge* lors de la mise ne route par la méthode *start()*.

```
public void setSeconde(int sec) { this.seconde = sec; }
public void setMinute(int min) { this.minute = min; }
public void setHeure(int h) { this.heure = h; }
public void setPeriode(int periode) { this.periode = periode; }
public void setJour(int jour) { this.jour = jour; }
public void setMois(int mois) { this.mois = mois; }
public void setAnnee(int annee) { this.annee = annee; }
```

```
public void setJournee(int journee) {
    switch (journee) {
        case 1 : this.journee = "Dimanche";
        case 2 : this.journee = "Lundi";
        case 3 : this.journee = "Mardi";
        case 4 : this.journee = "Mercredi";
        case 5 : this.journee = "Jeudi";
        case 6 : this.journee = "Vendredi";
        case 7 : this.journee = "Samedi";
    }
}
```

Chapitre VIII: Java : Gestion des fichiers

Nous abordons ici un chapitre très important lors de la programmation de façon générale. En effet un programme quelconque devient intéressant lorsque l'on peut reprendre des informations stockées ou sauvegarder des données. Pour ce faire, il faut savoir lire et écrire dans des fichiers.
En Java, il y a deux notions différentes : les flux et les fichiers.
Pour comprendre très rapidement, lorsque l'on utilisait la commande *System.out.println*, en fait on utilisait la notion de flux et plus exactement un flux de sortie.
Ces flux de sortie désignent un canal susceptible de recevoir de recevoir de l'information sous forme d'une suite d'octets. Il peut s'agir d'un périphérique d'affichage mais aussi d'un fichier ou encore d'une connexion à un site distant, voire d'un emplacement en mémoire centrale.

De façon comparable, il existe des flux d'entrée, c'est-à-dire des canaux délivrant de l'information sous forme d'une suite d'octets. Il peut s'agir d'un périphérique de saisie (clavier), d'un fichier, d'une connexion ou d'un emplacement en mémoire centrale.

Dans les flux, deux catégories existent : les flux binaires et les flux texte. Dans le premier cas, l'information est transmise sans modification de la mémoire au flux ou du flux à la mémoire. Dans le second cas, en revanche, l'information subit une transformation, nommée *formatage*, de manière que le flux reçoive ou transmette une suite de caractères. La méthode *println* réalise un tel formatage.

Comme dans la plupart des langages, on peut accéder à un fichier binaire, soit de façon séquentielle, soit de façon directe. Dans le premier cas, on traite les informations dans l'ordre qu'elles apparaissent dans le fichier. Par contre dans le second cas, on se place directement sur l'information voulue, sans avoir ainsi à consulter ou créer celles qui précédent.

Voici le plan de ce chapitre long et difficile :

- Création séquentielle d'un fichier binaire ;
- Lecture séquentielle d'un fichier binaire ;
- Accès direct à un fichier binaire
- Création d'un fichier texte ;
- Lecture d'un fichier texte.

En dernier point, on pourra voir une description détaillée des principales classes flux.

1. Création séquentielle d'un fichier binaire

1.1. Objectif

A l'aide du clavier, introduire des nombres entiers qui devront être stockés dans un fichier binaire.

1.2. Programme

```
import java.io.*;
public class Fichier1 {
      public static void main (String args[]) throws IOException {
         String nomfich;
         int n;
         System.out.println("Donner le nom du fichier a creer");
         nomfich = Clavier.lireString();
         DataOutputStream sortie = new DataOutputStream(new FileOutputStream(nomfich));
         do {
            System.out.println("Donner un entier. Si 0 = FIN");
            n = Clavier.lireInt();
            if(n != 0) sortie.writeInt(n);
         }
         while (n != 0);
         sortie.close();
         System.out.println("*** fin creation fichier ***");
      }
}
```

1.3. Analyse

Analysons ce programme pour mieux cerner les commandes utilisées.
import java.io.;* sert pour pouvoir utiliser les flux.
FileOutputStream f = new FileOutputStream("test.txt") sert à associer à l'objet *f* le nom du fichier.
DataOutputStream sortie = new DataOutputStream(f)
Cependant cette classe ne peut pas grand-chose c'est pourquoi on utilise une autre classe *DataOutputStream* qui elle possède plus de méthode et est construite à partir de la classe précédente et de ce fait l'objet *sortie* est directement associé à *test.txt* par l'intermédiaire de *f.*
On peut économiser la création d'un objet en combinant les deux comme dans le programme.
Cette dernière classe possède notamment la méthode *writeInt() ;* qui permet d'écrire un entier dans fichier par l'intermédiaire d'un flux. Il existe des méthodes similaires pour écrire pour tous les types primitifs ainsi que la classe *String*.
Il ne faut pas oublier de fermer le fichier à la fin du programme.

Il est à noter la présence de *throws IOException* qui est nécessaire au bon fonctionnement car l'utilisation de la méthode *writeInt()* peut provoquer une exception qu'il faudra gérer.

Il est possible de doter un flux d'un tampon (emplacement mémoire qui sert à optimiser les échanges avec le flux). Les informations sont d'abord enregistrées dans le tampon et ce n'est que lorsque celui-ci est plein qu'il est transmis dans le flux. Pour doter un flux d'un tampon, on crée un objet de type *BufferedOutputStream* en passant le type *FileOutputStream* en argument.
Voici la commande compressée :

```
DataOutputStream sortie = new DataOutputStream(new BufferedOutputStream(new FileOutputStream(nomfich))) ;
```

2. Lecture séquentielle d'un fichier binaire

2.1. Objectif

Relire séquentiellement un fichier comme crée au point précédent.

2.2. Programme

```
import java.io.*;
public class Fichier2 {
      public static void main (String args[]) throws IOException {
         String nomfich;
         int n = 0;
         System.out.println("Donner le nom du fichier a lister");
         nomfich = Clavier.lireString();
         DataInputStream entree = new DataInputStream(new FileInputStream(nomfich));
         System.out.println("Valeurs lues dans le fichier " + nomfich + ":");
         boolean eof = false;
         while (!eof) {
            try {n = entree.readInt();}
            catch (EOFException e) {eof = true ;}
            if(!eof) System.out.println(n);
         }
         entree.close();
         System.out.println("*** fin lecture fichier ***");
      }
}
```

2.3. Analyse

On retrouve les équivalents aux classes *DataOutputStream* et *FileOutpuStream* avec les classes qui permette les entrées donc ici les lectures : *DataInputStream* et *FileInputStream*.
Les méthodes équivalentes à *writeInt()* sont *readInt()* et toutes celles associées à un type primitif et la classe *String*

Comme on ne connaît pas à l'avance le nombre de nombres à lire, on arrivera à la fin du fichier et la lecture d'un entier provoquera une exception de type *IOException* qui est gérée par les blocs *try{…} catch{…}*.
Dans pourquoi, une variable de type booléenne *eof* est déclarée au départ à *false* et dès que l'exception arrive, celle-ci devient *true* et le programme s'arrête.

L'initialisation de la variable *n* à 0 est obligatoire car le compilateur donnerait une erreur dans le cas contraire.

3. Accès direct à un fichier binaire

3.1. Introduction

Java permet de réaliser l'accès direct à un fichier binaire à l'aide de la classe *RandomAccessFile* qui dispose des fonctionnalités des deux classes *DataInputStream* et *DataOutputStream*, en particulier les méthodes *readInt*, *writeInt* et les autres liées aux types primitifs et la classe *String*.
En plus du nom de fichier associé, il faut passer en paramètre le mode d'accès : r = lecture seule, rw = lecture et écriture.
La classe *RandomAccessFile* dispose d'une méthode *seek* permettant d'agir sur le pointeur de fichier. Ce dernier correspond au rang du prochain octet à lire ou à écrire. Le premier octet portant le numéro 0. Tant que l'on n'agit pas sur ce pointeur, il se trouve incrémenté, après chaque opération, du nombre d'octets lus ou écrits.

3.2. Exemple d'accès direct à un fichier existant

```
import java.io.*;
public class Fichier3 {
      public static void main (String args[]) throws IOException {
         String nomfich;
         int n,num;
         System.out.println("Donner le nom du fichier a lister");
         nomfich = Clavier.lireString();
         RandomAccessFile entree = new RandomAccessFile(nomfich,"r");
         do {
            System.out.println("Numero de l'entier recherche :(0 = FIN)");
            num = Clavier.lireInt();
            if(num == 0) break;
            entree.seek(4*(num-1));
            n = entree.readInt();
            System.out.println("valeur = " + n);
         }
         while (num != 0);
         entree.close();
         System.out.println("*** fin consultation fichier ***");
      }
}
```

3.3. Possibilités et problème de l'accès direct

Outre les possibilités de consultation rapide qu'il procure, l'accès direct facilite et accélère les opérations de mise à jour d'un fichier. Dans ce cas, on utilise le mode d'accès « rw ».

L'accès direct permet ainsi de créer un nouveau fichier dans lequel les informations pourront être introduite à la position que l'on désire. Attention la plupart des environnements créent des « trous » en fonction de la position donnée pour l'écriture.
Lors de la lecture, il faut donc être en mesure de repérer ces éventuels trous.
Deux techniques existent :

- Avant d'exécuter le programme, on commence par initialiser tous les emplacements du fichier par une valeur conventionnelle dont on sait qu'elle ne pourra pas apparaître comme valeur effective.
- On peut aussi créer une table des trous, table qui doit alors de préférence être conservée dans le fichier lui-même.

Dernier problème de l'accès direct, c'est qu'il faut toujours donner la position du pointeur.

3.4. En cas d'erreur

3.4.1. Erreur de pointage

Dans le programme précédent, pour passer d'un élément à un autre, on a utiliser la formule : 4*(num – 1). Si au lieu de cela on utilisait 4 * num – 1, il n'y aurait pas d'erreur de compilation mais par contre on se trouverait à cheval entre le dernier octet d'un entier et le premier octet de l'entier suivant. On obtient alors des résultats fantaisistes mais le dernier nombre ne serait pas obtenu.

3.4.2. Positionnement hors fichier

Comme il faut donner la position du pointeur (rang), il se peut que la valeur introduite soit supérieure ou soit on introduit une valeur négative.
Les deux donnent des exceptions différentes :

- pour une valeur négative, *seek* lance une exception *IOException*. Si elle n'est pas traitée, on obtient le message *Negative seek offset*.
- pour une valeur supérieure, aucune exception n'est déclenchée par *seek* mais on obtient une exception *EOFexception* si on lance une opération de lecture. L'écriture est possible si le mode est « rw », puisque c'est précisément comme cela qu'on peut créer un nouveau fichier.

Dans ces conditions, lorsque l'on consulte un fichier existant en accès direct, le mieux est de se protéger explicitement d'un mauvais positionnement :

- en déterminant la taille du fichier à l'aide de la méthode *length* de la classe *RandomAccessFile* (le résultat est de type *long*).
- en s'assurant que la valeur fournie à *seek* est bien non négative et inférieur à la taille.

3.4.3. Programme modifié en ce sens

```
import java.io.*;
public class Fichier4 {
      public static void main (String args[]) throws IOException {
         String nomfich;
         int n,num;
         System.out.println("Donner le nom du fichier a lister");
         nomfich = Clavier.lireString();
         RandomAccessFile entree = new RandomAccessFile(nomfich,"r");
         long taille = entree.length();
         do {
            System.out.println("Numero de l'entier recherche :(0 = FIN)");
            num = Clavier.lireInt();
            if(num == 0) break;
            int rang = 4 * (num-1);
            if ((rang >= 0) && (rang < taille)){
                        entree.seek(rang);
                        n = entree.readInt();
                        System.out.println("valeur = " + n);
            }
            else { System.out.println("Position inexistante"); continue; }
         }
         while (num != 0);
         entree.close();
         System.out.println("*** fin consultation fichier ***");
      }
}
```

4. Les flux texte

4.1. Introduction

Avec les classes *DataInputStream* et *DataOutpuStream*, il est possible de lire ou écrire sur un flux binaire des informations de n'importe quel type primitif et des éléments de la classe *String*.
Cependant les caractères sont représentés sous la forme de deux octets (codage Unicode), ils sont véhiculés sous cette forme sur le flux.
Or dans les environnements, on est habitué à manipuler des fichiers dits de type texte (fichiers qui peuvent être lus par un éditeur de texte).
Dans ces fichiers texte, chaque caractère se trouve codé sur un seul octet et suivant un code dépendant plus ou moins de l'environnement. On y trouve généralement un caractère représentant la fin de ligne (le code hexadécimal 10 pour Unix et la suite 13 et 10 pour PC)
Pour palier à ce manque, Java dispose de deux autres familles de classes, dérivées de classes abstraites *Printer* et *Reader* qui permettent de manipuler des flux texte.
Les caractères manipulés subissent dès lors la transformation suivante :

- pour un flux en sortie : une conversion de deux octets représentant un caractère Unicode en un octet correspondant au code local de ce caractère dans l'implémentation ;
- pour un flux en entrée : une conversion d'un octet représentant dans le code local en deux octets correspondant au code Unicode de ce caractère ;

En outre les fins de lignes seront transformées en accord avec leur représentation locale.

4.2. Création d'un fichier texte

4.2.1. Objectif

Ecrire un programme qui lit des nombres entier au clavier et qui pour chacun d'eux, écrit une ligne d'un fichier texte contenant le nombre fourni accompagné de son carré, sous la forme suivante : « 12 a pour carre 144 ».

4.2.2. Programme

```
import java.io.*;
public class Fichier5 {
   public static void main (String args[]) throws IOException {
      String nomfich;
      int n;
      System.out.println("Donner le nom du fichier a creer");
      nomfich = Clavier.lireString();
      PrintWriter sortie = new PrintWriter(new FileWriter(nomfich));
      do {
         System.out.println("Donner un entier :(0 = FIN)");
         n = Clavier.lireInt();
         if (n !=0) sortie.println(n + " a pour carre " + n*n);
      }
      while (n != 0);
      sortie.close();
      System.out.println("*** fin creation fichier ***");
   }
}
```

4.2.3. Analyse

FileWriter f = new FileWriter("test.txt") ; permet d'associer le nom de fichier « test.txt » à l'objet *f*, objet qui permet de manipuler un flux texte.
Les méthodes de cette classe ne permettent pas de formater le texte, c'est pourquoi on utilise une autre classe *PrintWriter* qui elle permet le formatage du texte à l'aide de *print* et *println*.
La construction ce fait comme suit : PrintWriter sortie = new PrintWriter (f) ;.
Il est aussi possible comme pour les flux binaires d'utiliser un tampon avec la classe *BufferedWriter* mais la classe *PrintWriter* dispose déjà son propre tampon.
La fin de ligne étant dépendante de l'environnement utilisé, il est possible de créer une fin de ligne en fonction de celui-ci, comme suit : String finLigne = System.getProperty ("line.separator") ;.

4.3. **Lecture d'un fichier texte**

Il est possible de lire un fichier texte dans son entièreté mais aussi accéder aux différentes informations présentes dans la ligne. Les deux cas se résolvent de manière différente.

4.3.1. Accès aux lignes d'un fichier texte

Il n'y a pas malheureusement le symétrique de *PrintWriter*, il faut se contenter de *FileReader* mais en la couplant à *BufferedReader* qui dispose d'une méthode *readLine*, il est possible de lire chaque ligne du fichier.
La commande est : BufferedReader entree = new BufferedReader(new FileReader("test.txt")) ;.
Si la fin de fichier est atteinte, la méthode *readLine* renvoie la valeur *null*.
Deux canevas sont alors possibles :

```
do {
   ligne = entree.readLine() ;
   if (ligne != null) { //traitement d'une ligne }
}
while (ligne != null) ;
```

```
while (true) {
      ligne = entree.readLine() ;
      if (ligne != null) break ;
      //traitement d'une ligne
}
```

Une manière encore plus condensée : while((ligne = entree.readLine()) != null) //traitement d'une ligne

4.3.2. Programme

```
import java.io.*;
public class Fichier6 {
        public static void main (String args[]) throws IOException {
                String nomfich;
                String ligne;
                System.out.println("Donner le nom du fichier a lire");
                nomfich = Clavier.lireString();
                BufferedReader entree = new BufferedReader(new FileReader(nomfich));
                while((ligne = entree.readLine()) != null) System.out.println(ligne);
                entree.close();
                System.out.println("*** fin lecture fichier ***");
        }
}
```

4.3.3. La classe StringTokenizer

Pour accéder à chacune des informations d'une même ligne, Java dispose d'une classe *StringTokenizer* qui permet de couper la chaîne en différents *tokens* (sous chaînes) en se fondant sur la présence de caractères séparateurs qu'on choisit librement. Par ailleurs, on pourra appliquer à ces différents *tokens*, les possibilités de conversion d'une chaîne en un type primitif.

4.3.4. Programme 1

Il faut créer un programme qui extrait les nombres d'un fichier nommé « reels.txt », contenant des nombres flottants répartis d'une manière quelconque, c'est-à-dire que chaque ligne peut comporter plusieurs nombres séparés par au moins un espace et qui ensuite effectue une somme.
Pour ce programme, il faut construire un objet de type *StringTokenizer* comme suit :

```
StringTokenizer tok = new StringTokenizer(ligneLue," ") ;
```

Le second argument indique les différents séparateurs utilisés.
Cette classe possède deux méthodes intéressantes : *countToken* qui compte le nombre de *tokens* d'un objet du type et *nextToken* qui fournit le *token* suivant s'il existe.

```
import java.io.*; import java.util.*; // pour la classe StringTokenizer
public class Fichier7 {
   public static void main (String args[]) throws IOException {
      String nomfich;
      String ligneLue;
      double x, som = 0;
      System.out.println("Donner le nom du fichier a lire");
      nomfich = Clavier.lireString();
      BufferedReader entree = new BufferedReader(new FileReader(nomfich));
      while((ligneLue = entree.readLine()) != null) {
         StringTokenizer tok = new StringTokenizer(ligneLue," ");
         int nv = tok.countTokens();
         for (int i = 0; i < nv ; i++) {
            x = Double.parseDouble(tok.nextToken());
            som += x;
            System.out.println(x + " ");
         }
      }
      entree.close();
      System.out.println("somme des nombres = " + som);
      System.out.println("*** fin lecture fichier ***");
   }
}
```

4.3.5. Programme 2

Nous allons reprendre le ficher « test.txt » et reprendre les nombres et leur carré sans le texte « a pour carre ».

```
import java.io.*; import java.util.*;
public class Fichier8 {
   public static void main (String args[]) throws IOException {
      String nomfich, ligneLue;
      int nombre, carre;
      System.out.println("Donner le nom du fichier a lire");
      nomfich = Clavier.lireString();
      BufferedReader entree = new BufferedReader(new FileReader(nomfich));
      System.out.println("Nombres et carres contenus dans ce fichier");
      while((ligneLue = entree.readLine()) != null) {
         StringTokenizer tok = new StringTokenizer(ligneLue," ");
         nombre = Integer.parseInt(tok.nextToken());
         for(int i = 0 ; i < 3 ; i++) tok.nextToken(); // pour passer au-delà du texte « a pour carre »
         carre = Integer.parseInt(tok.nextToken());
         System.out.println(nombre + " " + carre);
      }
      entree.close();
      System.out.println("*** fin lecture fichier ***");
   }
}
```

5. La gestion des fichiers : la classe *File*

Java dispose d'une classe *File* qui offre des fonctionnalités de gestion de fichiers comparables à celles auxquelles on accède par le biais de commandes système de l'environnement.
Il s'agit bien d'opérations concernant le fichier et non son contenu. Cette classe gère aussi les dossiers.

5.1. Création d'un objet de type *File*

La création d'un objet de type *File* se fait comme suit : `File monFichier = new File("test.txt") ;`.
Cet objet est un objet créé en mémoire et non un fichier dans l'arborescence du disque.
Il existe une méthode pour créer un fichier sur le disque : `boolean ok = monFichier.createNewFile() ;`.

On peut associer à un objet *File* le chemin mais attention à l'environnement. En effet, pour un PC, le caractère de séparation des dossiers est « \ ». Par contre pour Unix ou Linux, le caractère de séparation est « / ».
Il existe aussi une méthode permettant la portabilité et cela à l'aide d'une constante *File.separator*. Il suffit d'écrire : `String s = File.separator;` et utiliser « s » au lieu du séparateur.

5.2. Utilisation d'objets de type *File*

5.2.1. Dans les constructeur de flux

Au lieu d'écrire : `DataOutputStream sortie = new DataOutputStream(new FileOutputStream(nomfich)) ;`, on peut écrire : `DataOutputStream sortie = new DataOutputStream(new FileOutputStream(new File(nomfich))) ;`.
Cette seconde écriture permet en plus de la gestion des flux la possibilité de renommer le fichier, le déplacer, le copier, …

5.2.2. Création et suppression

Il est possible de créer à l'aide d'on objet de type *File* un fichier mais aussi de le supprimer et de créer un dossier à l'aide des méthodes : *createNewFile()*, *delete()*, *mkdir()* et *mkdirs()*.

- *createNewFile()* : renvoie un booléen de valeur *true* si la création s'est bien déroulée ;
- *delete()* : renvoie un booléen de valeur *true* si la suppresion a eu lieu. Si le fichier n'existe pas, il renvoie *false*.
- *mkdir()* : renvoie un booléen de valeur *true* si la création s'est déroulée correctement. Seul le dernier niveau de répertoire peut être créé avec cette méthode.
- *mkdirs()* : renvoie un booléen de valeur *true* si la création s'est déroulée correctement. Cette méthode permet la création de niveaux intermédiaires de répertoires.

5.2.3. Test d'existence

Trois méthode permettent de connaître le fichier existe, de connaître si l'association est un fichier ou un dossier.

- *exists()* : fournit la valeur *true* si le fichier correspondant existe ;
- *isFile()* : fournit la valeur *true* si l'objet correspond à un nom de fichier ;
- *isDirectory()* : fournit la valeur *true* si l'objet correspond à un nom de dossier.

5.2.4. Informations

Méthodes permettant de connaître des informations comme la taille, le nom, les attributs.

- *length()* : fournit la longueur du fichier en octet (0 s'il n'existe pas ou il est vide) ;
- *getName()* : fournit une chaîne contenant le nom correspondant (sans le chemin) ;
- *isHidden()* : fournit la valeur *true* si l'attribut du fichier est « caché » ;
- *canRead()* : fournit la valeur *true* si l'attribut du fichier est « lecture » ;
- canWrite() : fournit la valeur *true* si l'attribut du fichier est « archive » ;

5.2.5. Accès aux membres d'un dossier

Il est possible de connaître la liste de tous les membres d'un dossier donné.
String [] list() : fournit un tableau de chaîne correspondant
File [] listFiles() : fournit les mêmes informations mais sous la forme plus pratique d'un tableau d'objets de type *File*.

Voici un exemple pour obtenir la liste de tous les sous dossiers et fichiers appartenant à un dossier dont on fournit le nom dans une variable.

```
import java.io.*;
public class Fichier9 {
   public static void main (String args[]) {
      String nomRepertoire;
      System.out.println("Donner le dossier a lister");
      nomRepertoire = Clavier.lireString();
      File repert = new File(nomRepertoire);
      String [] liste = repert.list();
      for (int i = 0 ; i < liste.length ; i++) System.out.println(liste[i]);
   }
}
```

ou encore

```
import java.io.*;
public class Fichier10 {
   public static void main (String args[]) {
      String nomRepertoire;
      System.out.println("Donner le dossier a lister");
      nomRepertoire = Clavier.lireString();
      File repert = new File(nomRepertoire);
      File [] liste = repert.listFiles();
      for (int i = 0 ; i < liste.length ; i++) System.out.println(liste[i].getName()+"\t\t"+liste[i].length()+" octets");
   }
}
```

Le premier programme permet de lister les dossiers et fichiers mais rien de plus.
Par contre le second programme, comme on utilise un tableau d'objet de type *File*, il est possible d'avoir des informations supplémentaires comme la taille, les attributs, …
Il est même possible de n'afficher que les fichiers ou que les dossiers.

6. Les flux en général

Dans ce dernier point, un rappel global des constructeurs et méthodes principales est exposé. Il faut surtout savoir que les flux permettent beaucoup plus que l'utilisation de fichiers puisque il s'agit de canal où navigue des informations.

6.1. Généralités

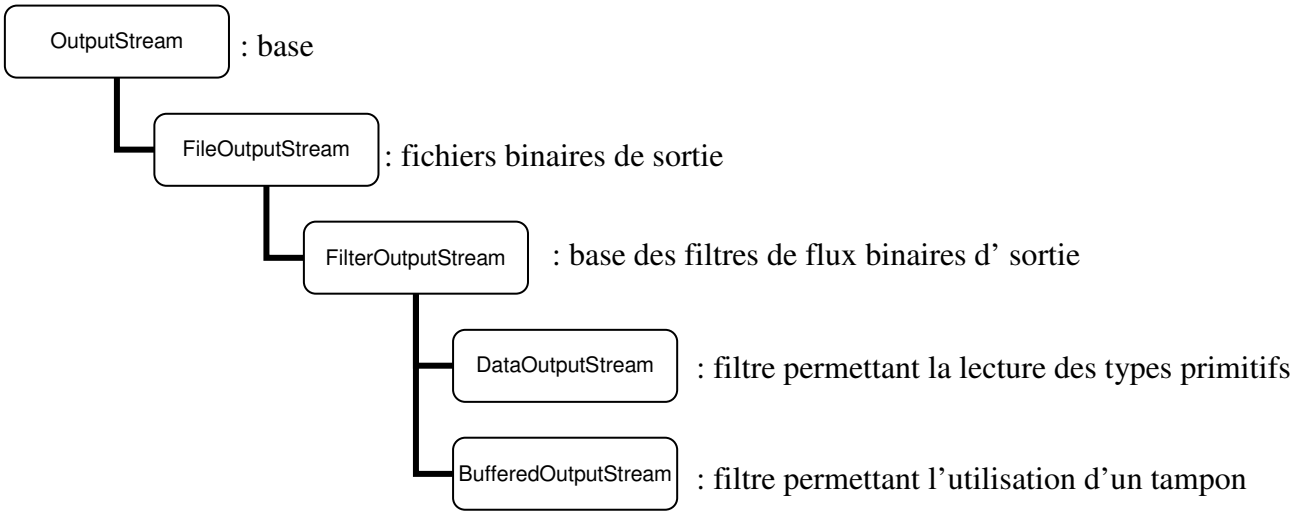
Les différentes classes de flux peuvent se répartir en cinq familles, chacune issue d'une classe de base abstraite :

- *OutputStream* : flux binaire de sortie ;
- *InputStream* : flux binaire d'entrée ;
- *RandomAccessFile* : fichiers à accès direct ;
- *Writer* : flux texte de sortie ;
- *Reader* : flux texte d'entrée.

Hormis la classe *RandomAccessFile*, réservée aux fichiers, les quatre autres classes de base n'imposent pas un flux de nature précise. Mais dans les quatre classes, on trouve des classes spécialisées pour des flux associés à un fichier.

6.2. Les flux binaires de sortie

Les classes les plus usuelles sont organisées suivant la hiérarchie suivante :



6.2.1. OutputStream

Classe abstraite, base des classes relatives à des flux binaires de sortie, dotée de fonctionnalités rudimentaires.
Les méthodes principales :

- void write(int n) : écrit l'octet de poids faible n ;
- void write(byte[]b) : écrit le tableaux d'octets b ;
- void close() : ferme le flux ;
- void flush() : vide le tampon, s'il existe.

6.2.2. FileOutputStream

Flux binaire de sortie, associé à un fichier. Cette classe est dotée des mêmes fonctionnalités rudimentaires de la précédente.
Trois possibilités pour construire un objet de cette classe :

- FileOutputStream(String nomFichier) ;
- FileOutputStream(String nomFichier, boolean b) (si b a la valeur *true*, ouverture)
- FileOutputStream(File objFichier) ;

6.2.3. FilterOutputStream

Filtres des flux binaires de sorties, c'est-à-dire de classes qui ajoutent des fonctionnalités supplémentaires à un flux binaire de sortie.

6.2.4. DataOutputStream

Cette classe permet de doter un flux binaire de sortie de possibilités d'écritures des différents types primitifs et les chaînes de caractères.

- DataOutputStream(OutputStream fluxSortie) : est la manière de construire cet objet ;
- void writeByte(byte x) : écrit la valeur x comme un byte ;
- void writeShort(short x) : écrit la valeur x comme un entier court ;
- void writeInt(int x) : écrit la valeur x comme un entier ;
- void writeLong(long x) : écrit la valeur x comme un entier long ;
- void writeFloat(float x) : écrit la valeur x comme un réel simple ;
- void writeDouble(double x) : écrit la valeur x comme un réel double ;
- void writeBoolean(boolean x) : écrit la valeur x comme un booléen ;
- void writeChar(char x) : écrit la valeur de x comme un caractère ;
- void writeChars(String chaine) : écrit les caractères de la chaîne mais l'équivalent en lecture n'existe pas.
- void writeUTF(String chaine) : écrit les caractères de la chaîne en format UTF.
  Son équivalent en lecture existe.

6.2.5. BufferedOutputStream

Cette classe permet de doter un flux binaire d'un tampon.
Les constructeurs :

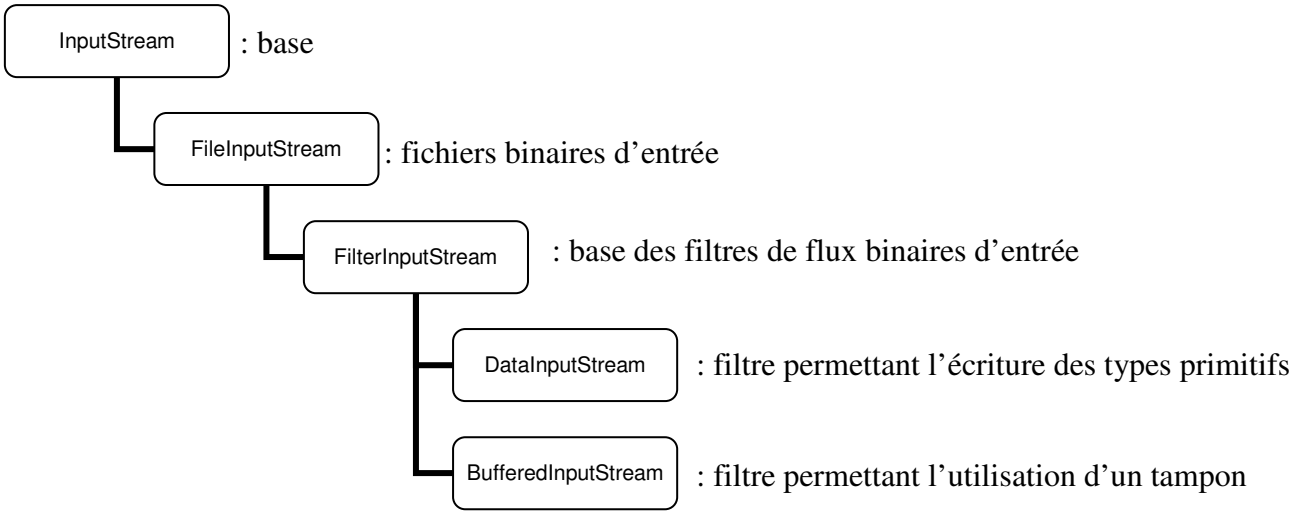
- BufferedOutputStream(OutputStream fluxSortie) ;
- BufferedOutputStream(OutputStream fluxSortie, int tailleTampon) ;

Une méthode :

- void flush() : vide le tampon.

6.3. Les flux binaires d'entrée

Les classes les plus usuelles sont organisées suivant la hiérarchie suivante :



6.3.1. InputStream

Classe abstraite, base des classes relatives à des flux binaires de sortie, dotée de fonctionnalités rudimentaires.
Les méthodes principales :

- int read(int n) : lit un octet (–1 si fin du flux atteint) ;
- int read(byte[]b) : lit une suite d'octets (au plus b.length) dans b ;
- void close() : ferme le flux ;
- long skip(long n) : saute n octets dans le flux et fournit le nombre de caractères sautés.

6.3.2. FileInputStream

Flux binaire d'entrée, associé à un fichier. Cette classe est dotée des mêmes fonctionnalités rudimentaires de la précédente.
Deux possibilités pour construire un objet de cette classe :

- FileInputStream(String nomFichier) ;
- FileInputStream(File objFichier) ;

6.3.3. FilterInputStream

Filtres des flux binaires d'entrée, c'est-à-dire de classes qui ajoutent des fonctionnalités supplémentaires à un flux binaire d'entrée.

6.3.4. DataInputStream

Cette classe permet de doter un flux binaire d'entrée de possibilités d'écritures des différents types primitifs et les chaînes de caractères.

- DataInputStream(InputStream fluxEntree) : est la manière de construire cet objet ;
- byte readByte() : lit l'information comme un byte ;

- short readShort() : lit l'information comme un entier court ;
- int readInt() : lit l'information comme un entier ;
- long readLong() : lit l'information comme un entier long ;
- float readFloat() : lit l'information comme un réel simple ;
- double readDouble() : lit l'information comme un réel double ;
- boolean readBoolean() : lit l'information comme un booléen ;
- char readChar() : lit l'information comme un caractère ;
- String readUTF() : lit l'information comme une chaîne en format UTF.

6.3.5. BufferedInputStream

Cette classe permet de doter un flux binaire d'un tampon.
Les constructeurs :

- BufferedInputStream(InputStream fluxEntree) ;
- BufferedInputStream(InputStream fluxEntree, int tailleTampon) ;

Une méthode :

- void flush() : vide le tampon.

6.4. **Les fichiers d'accès direct**

RandomAccessFile

Les constructeurs :

- RandomAccessFile(File objetFichier, String modeOuverture)
- RandomAccessFile(String nomFichier, String modeOuverture)

Le mode d'ouverture est « r » pour lecture seule et « rw » pour lecture et écriture.

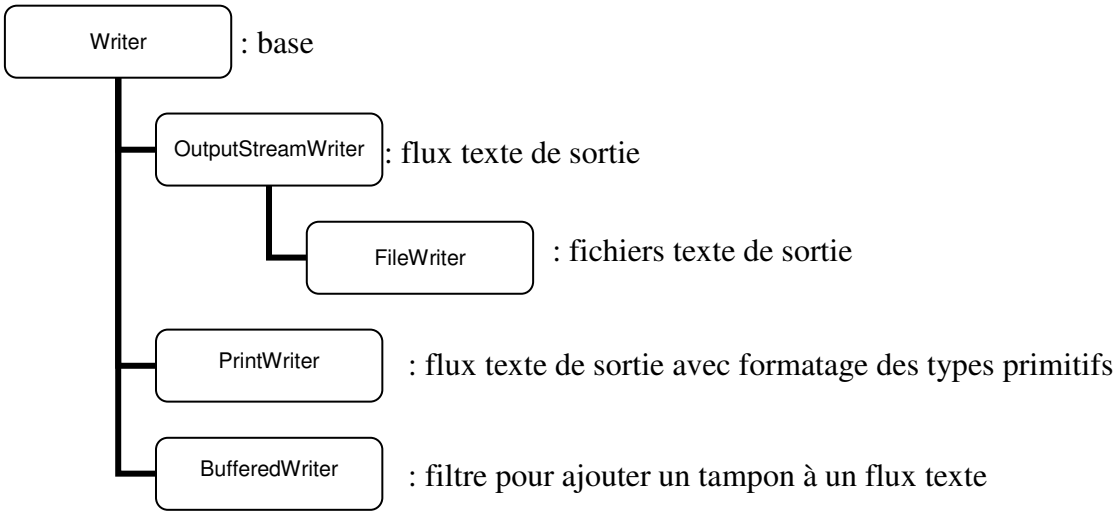
Les méthodes :

- void writeByte(byte x) : écrit la valeur x comme un byte ;
- void writeShort(short x) : écrit la valeur x comme un entier court ;
- void writeInt(int x) : écrit la valeur x comme un entier ;
- void writeLong(long x) : écrit la valeur x comme un entier long ;
- void writeFloat(float x) : écrit la valeur x comme un réel simple ;
- void writeDouble(double x) : écrit la valeur x comme un réel double ;
- void writeBoolean(boolean x) : écrit la valeur x comme un booléen ;
- void writeChar(char x) : écrit la valeur de x comme un caractère ;
- byte readByte() : lit l'information comme un byte ;
- short readShort() : lit l'information comme un entier court ;
- int readInt() : lit l'information comme un entier ;
- long readLong() : lit l'information comme un entier long ;
- float readFloat() : lit l'information comme un réel simple ;

- double readDouble() : lit l'information comme un réel double ;
- boolean readBoolean() : lit l'information comme un booléen ;
- char readChar() : lit l'information comme un caractère ;
- void seek(long position) : déplace le pointeur de la valeur de position octet ;
- long length() : donne la taille du fichier ;
- long getFilePointer() : donne la position du pointeur dans le fichier.

6.5. **Les flux texte de sortie**

Les classes les plus usuelles sont organisées suivant la hiérarchie suivante :



6.5.1. Writer

Classe de base à toutes les classes relatives à des flux texte de sortie dotée de fonctionnalités rudimentaires.
Les méthodes :

- void write(int n) : écrit le caractère n ;
- void write(char[] tabCar) : écrit le tableau de caractères tabCar ;
- void close() : ferme le flux ;
- void flush() : vide le tampon.

6.5.2. OutputStreamWriter

Flux texte de sortie. Cette classe est dotée des mêmes fonctionnalités de la précédente.
Constructeur : OutputStreamWriter(OutputStream fluxSortie) ;

6.5.3. FileWriter

Flux texte de sortie, associés à un fichier.
Les constructeurs :

- FileWriter(File objetFichier) ;
- FileWriter(String nomFichier) ;
- FileWriter(String nomFichier,boolean ext) : si ext = *true*, ouverture du fichier.

6.5.4. PrintWriter

Flux texte de sortie, dotés de possibilités de formatage avec *print* et *println*.
Les constructeurs :

- PrintWriter(OutputStream fluxSortie) ;
- PrintWriter(Writer fluxTexteSortie) ;
- PrintWriter(OutputStream fluxSortie, boolean vidageAuto) : si vidageAuto = *true*, le tampon est vidé à chaque appel de *println* ;
- PrintWriter(Writer fluxTexteSortie, boolean vidageAuto) : si vidageAuto = *true*, le tampon est vidé à chaque appel de *println*).

Les méthodes :

- void print(String texte) : écrit la chaîne de caractère texte ;
- void print(String texte) : écrit la chaîne de caractère texte avec un création d'une nouvelle ligne.

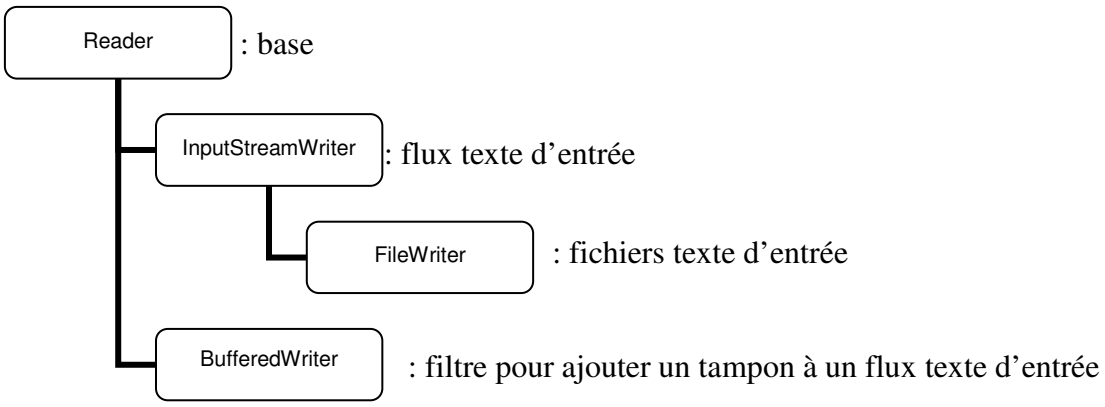
6.5.5. BufferedWriter

Cette classe permet de doter un flux texte d'un tampon
Les constructeurs :

- BufferedWriter(Writer fluxTexteSortie) ;
- BufferedWriter(Writer fluxTexteSortie, int tailleTampon).

6.6. **Les flux texte d'entrée**

Les classes les plus usuelles sont organisées suivant la hiérarchie suivante :



6.6.1. Reader

Classe de base à toutes les classes relatives à des flux texte d'entrée dotée de fonctionnalités rudimentaires.
Les méthodes :

- int read() : lit un caractère, fournit –1 si fin de flux ;
- int read(char[] tabCar) : lit une suite d'octets (au plus tabCar.length) dans tabCar ;
- void close() : ferme le flux ;
- long skip(long n) : saute n octets dans le flux et fournit le nombre de caractères sautés.

6.6.2. InputStreamWriter

Flux texte d'entrée. Cette classe est dotée des mêmes fonctionnalités de la précédente.
Constructeur : InputStreamWriter(InputStream fluxEntree) ;

6.6.3. FileReader

Flux texte d'entrée, associés à un fichier.
Les constructeurs :

- FileReader(File objetFichier) ;
- FileReader(String nomFichier) ;

6.6.4. BufferedWriter

Cette classe permet de doter un flux texte d'un tampon et de fonctionnalités de lecture globale d'une ligne
Les constructeurs :

- BufferedReader(Reader fluxTexteEntree) ;
- BufferedReader(Reader fluxTexteEntree, int tailleTampon).

Une méthode importante :

- String readLine() : lit une ligne de texte.

6.7. **Exercice**

a) Reprendre l'exercice 19.8 de la page 102 de telle sorte que les données soient sauvegardées dans un fichier et puissent être relues et modifiées par la suite.
b) Reprendre le dernier jeu du pendu et le réaliser cette fois avec un fichier contenant une série de mot. Le mot à chercher ne sera plus donnée par l'utilisateur mais par un choix aléatoire dans la liste des mots.

Table des matières

Table des matières

Table des matières

CHAPITRE V : Java : Programmation graphique

Table des matières

Table des matières

Table des matières

Table des matières