CPR Informatique

(poste 3159 ou 3164)

ECOLE NATIONALE
DES SCIENCES
GEOGRAPHIQUES

Septembre 2002

Programmer en Java

Table des matières

1.- Présentation générale du langage Java

1.1. – Introduction

Java est un langage de programmation récent (les premières versions datent de 1995) développé par Sun MicroSystems. Il est fortement inspiré des langages C et C++.

Comme C++, Java fait partie de la « grande famille » des **langages orientés objets**. Il répond donc aux trois principes fondamentaux de l'approche orientée objet (POO) : **l'encapsulation**, **le polymorphisme** et **l'héritage** Nous reviendrons bien évidemment en détails sur ces trois notions dans la suite de ce document (Cf. chapitre 2 : « Les principes de la POO »).

1.2. – Historique

Le langage Java trouve ses origines dans les années 1990. A cette époque, quelques ingénieurs (innovateurs) de SUN Microsystems commencent à parler d'un projet d'environnement indépendant du hardware pouvant facilement permettre la programmation d'appareil aussi variés que les téléviseurs, les magnétoscopes etc ... James Grosling (SUN Microsystems) développe un premier langage, Oak, permettant de programmer dans cet environnement. En 1992, tout est prêt pour envahir le marché avec cette nouvelle technologie mais la tentative se solde par un échec.

Bill Joy (co-fondateur de SUN Microsystems) sauve malgré tout le projet. Devant la montée en puissance de l'Internet, il lui semble intéressant d'insister sur l'élaboration d'un langage indépendant des plates-formes et des environnements (les principaux problèmes rencontrés sur l'Internet étant liés à l'hétérogénéité des machines et des logiciels utilisés).
Dès lors tout s'accélère. Oak est renommé (en 1995) Java et soumis à la communauté Internet grandissante. Une machine virtuelle, un compilateur ainsi que de nombreuses spécifications sont données gratuitement et Java entame une conquête fulgurante. Aujourd'hui, après de nombreuses améliorations (parfois modifications) Java n'est plus uniquement une solution liée à l'Internet. De plus en plus de sociétés (ou de particuliers) utilisent ce langage pour l'ensemble de leurs développements (applications classiques, interfaces homme-machine, applications réseaux …).

1.3. – Les principales raisons du succès de Java

Java a rapidement intéressé les développeurs pour quatre raisons principales :

- C'est un langage orienté objet dérivé du C, mais plus simple à utiliser et plus « pur » que le C++. On entend par « pur » le fait qu'en Java, on ne peut faire que de la programmation orienté objet contrairement au C++ qui reste un langage hybride, c'est-à-dire autorisant plusieurs styles de programmation. C++ est hybride pour assurer une compatibilité avec le C ;
- Il est doté, en standard, de bibliothèques de classes très riches comprenant la gestion des interfaces graphiques (fenêtres, boites de dialogue, contrôles, menus, graphisme), la programmation multi-threads (multitâches), la gestion des exceptions, les accès aux fichiers et au réseau … L'utilisation de ces bibliothèques facilitent grandement la tâche du programmeur lors de la construction d'applications complexes ;
- Il est doté, en standard, d'un mécanisme de gestions des erreurs (les exceptions) très utile et très performant. Ce mécanisme, inexistant en C, existe en C++ sous la forme d'une extension au langage beaucoup moins simple à utiliser qu'en Java ;

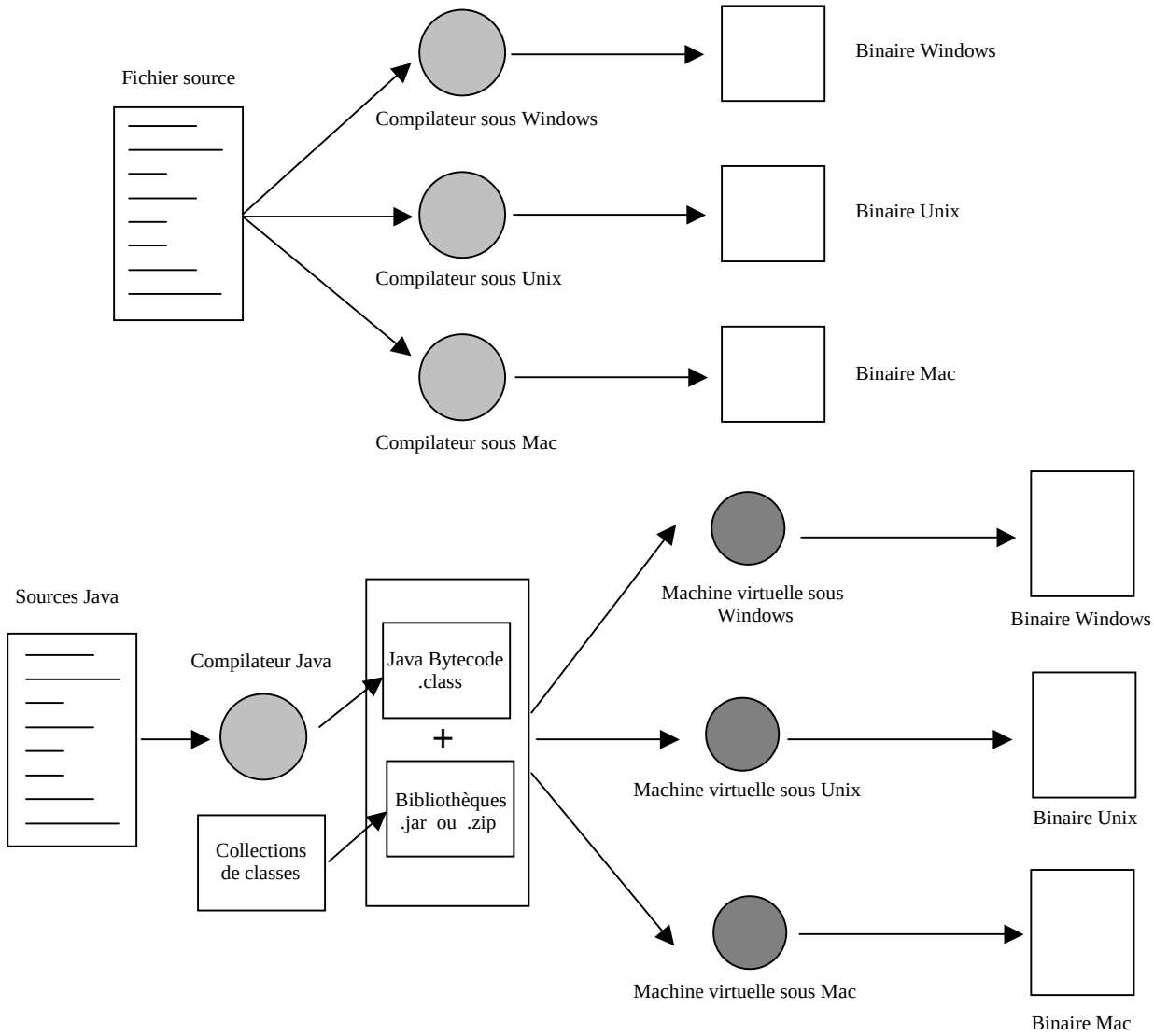
- Il est multi plates-formes : les programmes tournent sans modification sur tous les environnements où Java existe (Windows, Unix et Mac) ;

Ce dernier point a contribué à utiliser d'abord Java pour développer des applets (Cf. chapitre 6.- Les applets), qui sont des applications pouvant être téléchargées via l'Internet et exécutées dans un navigateur sur n'importe quelle plate-forme. Ainsi, une page statique HTML peut s'enrichir de programmes qui lui donneront un comportement dynamique.

1.4.- Les particularités de Java

1.4.1.- Multi plates-formes

Java est un langage qui doit être compilé et interprété. Dans une première phase on compile un programme (un ou plusieurs fichiers source *.java*) en fichiers *.class*. Le compilateur génère un fichier *.class* pour chacune des classes définies dans le(s) fichier(s) *.java*. L'ensemble des fichiers *.class* est ensuite interprété par la Machine Virtuelle Java (*Java Virtual Machine*) pour exécuter le programme (seule la JVM dépend du système). Ce principe donne à Java une portabilité maximale (Windows, Unix, MacIntosh …). Les figures ci-dessous montrent la différence de principe entre la compilation dans un langage classique (C++) et la compilation/interprétation en Java.



1.4.2.- Applets et applications

Il est possible de développer des applications isolées (*standalone applications*), fonctionnant avec l'interpréteur comme un programme habituel dans un langage classique mais aussi des "applets". Les applets sont des programmes qui peuvent être téléchargés sur l'Internet puis exécutés automatiquement quand ils sont intégrés à dans des pages HTML. Dans ce dernier cas, l'ensemble des fichiers *.class* est utilisé avec un fichier HTML qui fait appel à une des classes (Cf. chapitre 6).

Bien que les principes de programmation soient très proches, ce document traite uniquement des applications car la philosophie est plus proche des langages classiques (C/C++, Pascal, ADA …)

Néanmoins, nous consacrerons le chapitre 7.1 est consacré aux spécificités des applets.

1.5.- Questions courantes concernant Java

1.5.1. - Pourquoi ne pas interpréter directement le programme Java ?

- Les fichiers *.class* contiennent du *bytecode,* une sorte de code machine Java (comparable au code machine d'un microprocesseur). L'interpréteur exécute beaucoup plus rapidement ce bytecode que les fichiers sources *.java*.
- Seuls les fichiers *.class* sont nécessaires à l'exécution d'un programme Java. Ils contiennent du code machine et ne peuvent donc pas être lus par des tiers, protégeant ainsi le code source.
- Etant compilés, les fichiers *.class* sont de taille plus petite que le code source *Java* ; ceci est un argument important pour transférer les programmes sur l'Internet.
- Chaque fichier *.class* décrivant une classe d'objet, une même classe peut être utilisée par différents programmes sans que cette classe ne soit dupliquée dans chacun des programmes.

1.5.2. - Pourquoi ne pas compiler directement et générer un exécutable ?

Un exécutable contient du code qui n'est exécutable que sur la machine pour laquelle il est destiné et le seul moyen de rendre un langage multi plates-formes, sans avoir à recompiler le code source (comme en C/C++), est d'utiliser un interpréteur. L'autre avantage de l'interpréteur est qu'il peut être incorporé à un navigateur (Netscape/Internet Explorer), ce qui lui permet d'exécuter des programmes Java à l'intérieur de pages HTML.

1.5.3. - Qu'en est-il de la rapidité d'exécution de Java ?

Le plus gros problème d'un programme Java est son manque de rapidité d'exécution : l'interpréteur prend forcément un certain temps pour interpréter le code des fichiers *.class*, ce qui rend les programmes moins rapides que de « vrais exécutables ». Mais Java est encore jeune et Sun Microsystems et d'autres éditeurs de logiciels tentent d'optimiser les Machines Virtuelles Java. L'un des axes d'optimisation est actuellement le développement de compilateurs JIT (*Just In Time*) : quand une classe est chargée, elle est traduite dans le code machine de la plate-forme où elle est exécutée (traduction à la volée du micro-code Java vers par exemple, le code machine PowerPC ou Intel). Cette opération ralentit le départ de l'application, mais l'exécution du programme est ensuite beaucoup plus rapide !

1.5.4. - En résumé

Contrairement au C/C++, le compilateur Java ne génère pas de fichier exécutable. Il crée pour chacune des classes d'un fichier *Java* un fichier *.class* qui sera interprété par la Machine Virtuelle Java.

Compilation **: javac** [<options>] <fichiers-source- « .java »> . Exemple : **javac Test.java**

Exécution **: java** [<options>] <fichiers-class sans extension> . Exemple : **java Test**

1.6.- Ouvrages et sites de référence

Il est très difficile de sélectionner un ou deux ouvrages sur Java tant le nombre de livres sur le sujet est important depuis quelques années. Pour information, les deux seuls ouvrages consultés pour rédiger ce document est « Apprenez Java en 21 jours » de Laura Lemay et Charles Perkins (aux éditions S&SM) et l'excellent «Thinking in Java, 2nd édition » de Bruce Eckel (aux éditions MindView, Inc et traduit en français sur le site http://penserenjava.free.fr/).

Java étant très lié à l'Internet, il est plus simple et moins coûteux de consulter les nombreux sites consacrés à ce langage. Voici la liste de quelques sites très intéressants :

http://www.eteks.com/	France	Un des sites français les plus complets.
http://www.sun.fr/developpeurs/	France	Sun Developer Connection est le site d'informations en français sur Java de Sun Microsystems.
http://www.developpez.com/java/	France	Ce site propose de nombreux documents sur le développement en Java.
http://java.sun.com/	USA	Javasoft est la division de Sun Microsystems qui développe Java. Leur site est donc la référence en Java.
http://www.gamelan.com/	USA	Gamelan propose un annuaire des applets et des applications Java existantes à ce jour
http://www.borland.com/jbuilder/	USA	Le site de Borland consacré à JBuilder (environnement de développement en Java).

2.- Les principes de la POO

Avertissement : ce chapitre présente les concepts de la programmation orienté objet ; il est directement inspiré du livre de Bruce Eckel : « Thinking in Java ».

Après quelques années d'enseignement dans le domaine de la programmation, j'ai remarqué qu'il existait deux types de « programmeurs en herbe » : ceux qui ne veulent pas se lancer dans la POO sans en connaître les tenants et les aboutissants et ceux qui au contraire ne saisissent les concepts généraux qu'après avoir testé les mécanismes de mise en œuvre (en clair « apprendre par l'exemple »). Ce chapitre a donc été écrit pour les premiers. Si vous faites partie de la seconde catégorie, vous pouvez lire directement le chapitre 3 qui présente la syntaxe du langage Java. Vous pourrez y revenir plus tard pour approfondir vos connaissances.

2.1.- L'encapsulation

2.1.1. - La notion d'objet

La complexité des problèmes qu'on est capable de résoudre est directement proportionnelle au type et à la qualité de notre capacité d'abstraction. Tous les langages de programmation fournissent des abstractions. Le langage assembleur, par exemple, est une abstraction de la machine sous-jacente. Beaucoup de langages « impératifs » (Fortran, Pascal, C) sont des abstractions du langage assembleur. Ces langages sont de nettes améliorations par rapport à l'assembleur, mais leur abstraction première requiert une réflexion en termes de structure d'ordinateur et non en terme de structure du problème à résoudre.
L'approche orientée objet va plus loin en fournissant au programmeur des outils représentant des éléments dans l'espace problème. Ces éléments sont représenté dans l 'espace solution par des « **objets** ». L'idée est que le programme est autorisé à s'adapter à l'esprit du problème en ajoutant de nouveaux types d'objet, de façon à ce que, quand on lit le code décrivant la solution, on lit aussi quelque chose qui décrit le problème. Ainsi, la POO permet de décrire le problème avec les termes mêmes du problème plutôt qu'avec les termes de la machine où la solution sera mise en œuvre.

Alan Kay résume ainsi les cinq caractéristiques principales de `Smalltalk` (premier langage de programmation orienté objet et l'un des langages sur lequel est basé Java) :

- **Toute chose est un objet.** Il faut penser à un objet comme à une variable améliorée : il stocke des données, mais on peut « effectuer des requêtes » sur cet objet, lui demander de faire des opérations sur lui-même. En théorie, on peut prendre n'importe quel composant conceptuel du problème qu'on essaye de résoudre (un chien, un immeuble, un service administratif, etc...) et le représenter en tant qu'objet dans le programme ;
- **Un programme est un ensemble d'objets qui communiquent entre eux en s'envoyant des messages.** Pour qu'un objet effectue une requête, on « envoie un message » à cet objet. Plus concrètement, un message est un appel à une fonction appartenant à un objet particulier ;
- **Chaque objet a son propre espace de mémoire composé d'autres objets.** On peut ainsi cacher la complexité d'un programme par la simplicité des objets mis en œuvre ;
- **Chaque objet est d'un type précis.** Dans le jargon de la POO, chaque objet est une *instance* (ou une réalisation) d'une *class*e, où « classe » a une signification proche de « type » (Cf. chapitre suivant) ;

- **Tous les objets d'un type particulier peuvent recevoir le même message.** C'est une caractéristique lourde de signification, comme vous le verrez plus tard, parce qu'un objet de type « cercle » est aussi un objet de type « forme géométrique », un cercle se doit d'accepter les messages destinés aux formes géométriques. Cela veut dire qu'on peut écrire du code parlant aux formes géométriques qui sera accepté par tout ce qui correspond à la description d'une forme géométrique.

2.1.2. - La notion de classe

L'idée que tous les objets, tout en étant uniques, appartiennent à un ensemble d'objets qui ont des caractéristiques et des comportements communs fut utilisée directement dans le premier langage orienté objet, Simula-67.

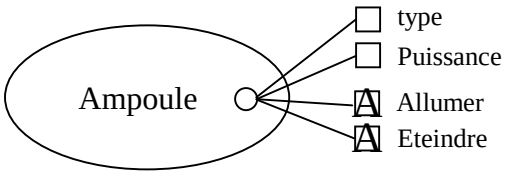
Des objets semblables, leur état durant l'exécution du programme mis à part, sont regroupés ensemble dans des « **classes d'objets** ». Le mot clef **class** est utilisé dans la plupart des langages orienté objet. On utilise les classes exactement de la même manière que les types de données prédéfinis. On peut en effet créer des variables d'une classe (appelés *objets* ou *instances*) et les manipuler comme des variables d'un type prédéfini. Les objets d'une même classe partagent des caractéristiques communes, mais chaque objet a son propre état. L'objet d'une classe est néanmoins différent d'une variable d'un type prédéfini parce qu'il est possible de lui associer des « comportements » répondant à des requêtes.

Comment utiliser un objet ?

Il faut pouvoir lui demander d'exécuter une requête, telle que terminer une transaction, dessiner quelque chose à l'écran, ou allumer un interrupteur. Chaque objet ne peut traiter que certaines requêtes. Les requêtes qu'un objet est capable de traiter sont définies par son **interface** et son type est ce qui détermine son interface.

Prenons l'exemple d'une ampoule électrique :

Interface



Utilisation

```
Ampoule amp = new Ampoule() ;
amp.Allumer() ;
```

L'interface précise quelles sont les caractéristiques d'un objet (ici type et puissance) et quelles sont les opérations que l'on peut effectuer sur un objet particulier (ici Allumer et Eteindre). L'interface ne précise pas la manière dont elles sont effectuées (c'est-à-dire le code correspondant à l'opération) : c'est le rôle de **l'implémentation**. Du point de vue de la programmation, une classe dispose de fonctions associées à chaque requête possible (appelée méthodes), et lorsqu'on effectue une requête particulière sur un objet, ces fonctions sont appelées. Ce mécanisme est souvent résumé en disant qu'on « envoie un message » à un objet, l'objet se « débrouillant » pour l'interpréter (exécution du code associé).

Dans l'exemple ci-dessus, le nom de la classe est **Ampoule**, le nom de l'objet Ampoule créé est **amp**. D'après l'interface, on peut demander à un objet Ampoule de s'allumer et de s'éteindre. Un objet Ampoule est créé en définissant une « référence » (**amp**) pour cet objet et en appelant **new** pour créer un nouvel objet de cette classe. Pour envoyer un message à cet objet, il suffit de spécifier le nom de l'objet suivi de la requête avec un point entre les deux. Du point de vue de l'utilisateur d'une classe prédéfinie, c'est tout ce qu'il est nécessaire de savoir pour utiliser les objets de la classe en question.

2.1.3. – L'implémentation cachée

Pour comprendre la notion d'implémentation cachée, il est nécessaire de différentier les programmeurs qui créent les classes d'objets et ceux qui les utilisent dans leur applications (qui peuvent d'ailleurs être eux-mêmes des créateurs de classes pour d'autres programmeurs utilisateurs etc …). Le but des programmeurs « clients » est de créer une boite à outils de classes réutilisables pour le développement rapide d'applications. Les « créateurs » de classes se concentrent sur la construction de la classe pour ne donner aux programmeurs « clients » que le strict nécessaire. L'intérêt principal d'une telle démarche est que le programmeur « créateur » peut changer des portions de code (améliorations, ajouts, suppressions …) sans que ces changements aient un impact sur l'utilisation des classes. Les portions cachées correspondent en général aux données de l'objet qui pourraient facilement être corrompues par un programmeur « clients » négligent ou mal informé.

La raison première du contrôle d'accès est donc d'empêcher les programmeurs « clients » de toucher à certaines portions auxquelles ils ne devraient pas avoir accès - les parties qui sont nécessaires pour les manipulations internes du type de données mais n'appartiennent pas à l'interface dont les utilisateurs ont besoin pour résoudre leur problème. C'est en réalité un service rendu aux utilisateurs car ils peuvent voir facilement ce qui est important pour leurs besoins et ce qu'ils peuvent ignorer.

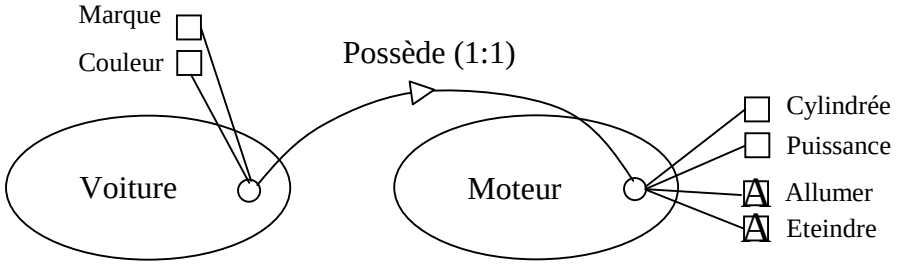
La deuxième raison d'être du contrôle d'accès est de permettre au concepteur de la bibliothèque de changer le fonctionnement interne de la classe sans se soucier des effets que cela peut avoir sur les programmeurs « clients ». Par exemple, on peut implémenter une classe particulière d'une manière simpliste afin d'accélérer le développement et se rendre compte plus tard qu'on a besoin de la réécrire afin de gagner en performances. Si l'interface et l'implémentation sont clairement séparées et protégées, cela peut être facilement réalisé.

La plupart des langages orienté objet (dont C++ et Java) utilisent trois mots clefs pour fixer des limites au sein d'une classe : **public, private et protected**. Le mot clef **public** veut dire que les définitions qui suivent sont disponibles pour tout le monde. Le mot clef **private**, au contraire, veut dire que personne, le créateur de la classe et les fonctions internes de ce type mis à part, ne peut accéder à ces définitions. Si quelqu'un tente d'accéder à un membre défini **private**, il récupère une erreur lors de la compilation. Le mot clef **protected** se comporte comme **private**, en moins restrictif. Une classe dérivée a accès aux membres **protected**, mais pas aux membres **private**. Nous reviendrons sur la notion de protected dans le chapitre sur l'héritage (Cf. chapitre 2.2).

2.1.4. – La réutilisation de l'implémentation

La réutilisation des classe créées et testées est l'un des grands avantages des langages orientés objets. Cependant créer des classes réutilisables demandent beaucoup de méthode et d'expérience.

La manière la plus simple de réutiliser une classe est d'utiliser directement un objet de cette classe. On peut également placer un objet d'une classe à l'intérieur d'une nouvelle classe : c'est ce qu'on appelle « créer un objet membre ». La nouvelle classe peut être constituée de n'importe quel nombre d'objets d'autres types, selon la combinaison nécessaire pour que la nouvelle classe puisse réaliser ce pour quoi elle a été conçue. Parce que la nouvelle classe est composée à partir de classes existantes, ce concept est appelé *composition* (*agrégatio*n). Par exemple : « une voiture possède un moteur ».



```
class Moteur
{
    float Cylindree ;
    float Puissance ;

    void Allumer () ;
    void Eteindre () ;
}

class Voiture
{
    Moteur m ;

    String Marque ;
    Color Couleur ;
}
```

Un objet de la classe Moteur est défini dans la classe Voiture

La composition s'accompagne d'une grande flexibilité : les objets membres de la nouvelle classe sont généralement privés, ce qui les rend inaccessibles aux programmeurs « clients » de la classe. Cela permet de modifier ces membres sans perturber le code des clients existants.

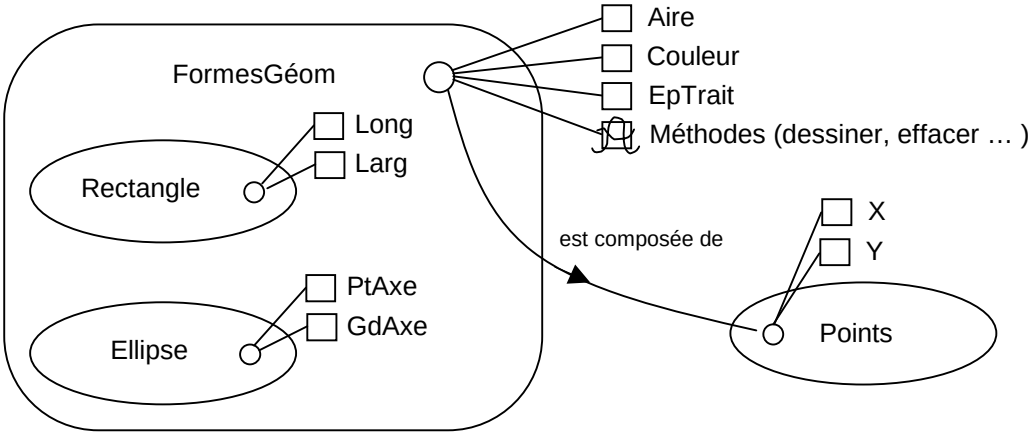
L'héritage ne dispose pas de cette flexibilité car le compilateur doit placer des restrictions lors de la compilation sur les classes créées avec héritage (Cf. chapitre suivant). Parce que la notion d'héritage est très importante au sein de la programmation orientée objet, elle est trop souvent recommandée et le nouveau programmeur pourrait croire que l'héritage doit être utilisé partout. Cela mène à des conceptions très compliquées et souvent cauchemardesques. La composition est la première approche à examiner lorsqu'on crée une nouvelle classe, car elle est plus simple et plus flexible. Avec de l'expérience, les endroits où utiliser l'héritage deviendront raisonnablement évidents.

2.2.- L'héritage

Une fois votre classe créée et testée, il serait dommage de devoir en créer une nouvelle pour implémenter des fonctionnalités similaires. Il est plus intéressant de prendre la classe existante, de la cloner et de faire les ajouts ou les modifications à ce clone. C'est ce que permet l'*héritage*, avec la restriction suivante : si la classe d'origine (appelée classe de *base* ou classe *parent*) est modifiée, le clone (appelé classe *dérivée*, *héritée*, *enfant* ou *sous-classe*) répercutera ces changements.

Une classe fait plus que décrire des contraintes sur un ensemble d'objets ; elle a aussi des relations avec d'autres classes. Deux classes peuvent avoir des caractéristiques et des comportements en commun, mais l'une des deux peut avoir plus de caractéristiques que l'autre et peut aussi réagir à plus de messages (ou y réagir de manière différente). L'héritage exprime cette similarité entre les classes en introduisant le concept de classes de base et de classes dérivées. Une classe de base contient toutes les caractéristiques et comportements partagés entre les classes dérivées. Un type de base est créé pour représenter le cœur de certains objets du système. De ce type de base, on dérive d'autres types pour exprimer les différentes manières existantes pour réaliser ce cœur.

Prenons l'exemple classique des « formes géométriques », utilisées entre autres dans les systèmes d'aide à la conception ou dans les jeux vidéos (Cf. Modèle HBDS ci-dessous). La classe de base est la « forme géométrique », et chaque forme a une taille, une couleur, une position, etc... Chaque forme peut être dessinée, effacée, déplacée, peinte, etc... A partir de cette classe de base, des classes spécifiques sont dérivées (héritées) : des ellipses, des rectangles, des cercles et autres, chacune avec des caractéristiques et des comportements additionnels (certaines figures peuvent être inversées par exemple). Certains comportements peuvent être différents, par exemple quand on veut calculer l'aire de la forme. La hiérarchie des classe révèle à la fois les similarités et les différences entre les formes.



Quand on hérite d'une classe, on crée une nouvelle classe. Cette nouvelle classe non seulement contient tous les membres d'une classe existante (bien que les membres privés soient cachés et inaccessibles), mais surtout elle duplique l'interface de la classe de base. Autrement dit, tous les messages acceptés par les objets de la classe de base seront acceptés par les objets de la classe dérivée. Comme on connaît « le type » de la classe par les messages qu'on peut lui envoyer, cela veut dire que la classe dérivée *est du même « type » que la classe de bas*e.
Dans l'exemple précédent, « un cercle est une forme ». **Cette équivalence de type via l'héritage est l'une des notions fondamentales dans la compréhension de la programmation orientée objet.**

Comme la classe de base et la classe dérivée ont toutes les deux la même interface, certaines implémentations accompagnent cette interface. C'est à dire qu'il doit y avoir du code à exécuter quand un objet reçoit un message particulier. Si on ne fait qu'hériter une classe sans rien lui rajouter, les méthodes de l'interface de la classe de base sont importées dans la classe dérivée. Cela veut dire que les objets de la classe dérivée n'ont pas seulement le même « type », ils ont aussi le même comportement, ce qui n'est pas particulièrement intéressant.

Il y a deux façons de différencier la nouvelle classe dérivée de la classe de base originale. La première est relativement directe : il suffit d'ajouter de nouvelles fonctions à la classe dérivée. Ces nouvelles fonctions ne font pas partie de la classe parent. Cela veut dire que la classe de base n'était pas assez complète pour ce qu'on voulait en faire, on a donc ajouté de nouvelles fonctions. Cet usage simple de l'héritage se révèle souvent être une solution idéale. Cependant, il faut vérifier s'il ne serait pas souhaitable d'intégrer ces fonctions dans la classe de base qui pourrait aussi en avoir l'usage. Ce processus de découverte et d'itération dans la conception est fréquent dans la programmation orientée objet.

Bien que l'héritage puisse parfois impliquer l'ajout de nouvelles fonctions à l'interface, il existe une autre manière de différencier la nouvelle classe : *changer* le comportement d'une des fonctions existantes de la superclasse. C'est ce que l'on appelle *redéfinir (ou surcharger)* cette fonction. Pour redéfinir une fonction, il suffit de créer une nouvelle définition pour la fonction dans la classe dérivée. C'est comme dire : « j'utilise la même interface ici, mais je la traite d'une manière différente dans cette nouvelle classe ».

Un débat est récurrent à propos de l'héritage : l'héritage ne devrait-il pas *seulement* redéfinir les fonctions de la classe de base et ne pas ajouter de nouvelles fonctions membres qui ne font pas partie de la superclasse ? Ceci implique que le type de la classe dérivée est *exactement* le même que celui de la classe de base (il a exactement la même interface) et donc que l'on peut exactement substituer un objet de la classe dérivée à un objet de la classe de base. On fait souvent référence à cette *substitution pure* sous le nom de *principe de substitutio*n. Dans un sens, c'est la manière idéale de traiter l'héritage puisque la relation entre la classe de base et la classe dérivée est une relation *est-*

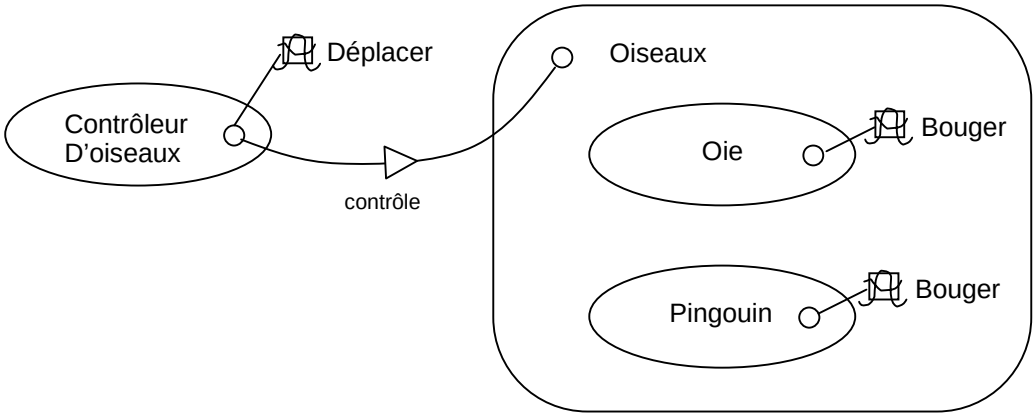
*u*n, (« un cercle *est une* forme »). Un test pour l'héritage peut donc être de déterminer si la relation « est-un » entre les deux classes considérées a un sens.
Mais parfois il est nécessaire d'ajouter de nouveaux éléments à l'interface d'une classe dérivée et donc d'étendre l'interface en créant une nouvelle classe. La nouvelle classe peut toujours être substitué à la classe de base, mais la substitution n'est plus parfaite parce que les nouvelles fonctions ne sont pas accessibles à partir de la classe parent.

Prenons le cas d'un système de climatisation. Supposons que notre maison dispose des tuyaux et des systèmes de contrôle pour le refroidissement, autrement dit elle dispose d'une interface qui nous permet de contrôler le refroidissement. Imaginons que le système de climatisation tombe en panne et qu'on le remplace par une pompe à chaleur qui peut à la fois chauffer et refroidir. La pompe à chaleur « *est-comme-un* » système de climatisation mais il peut faire davantage de choses. L'interface du nouvel objet a été étendue mais le système existant ne connaît rien qui ne soit dans l'interface originale.

2.3.- Le polymorphisme

Il arrive que l'on veuille traiter un objet non en tant qu'objet de la classe spécifique, mais en tant qu'objet de sa classe de base. Cela permet d'écrire du code indépendant des classes spécifiques. Dans l'exemple de la forme géométrique, les fonctions manipulent des formes génériques sans se soucier de savoir si ce sont des ellipses, des rectangles, des triangles ou même des formes non encore définies. Toutes les formes peuvent être dessinées, effacées etc … Ces fonctions envoient simplement un message à un objet forme, elles ne se soucient pas de la manière dont l'objet traite le message. Un tel code n'est pas affecté par l'addition de nouvelles classes et ajouter de nouvelles classes est la façon la plus commune d'étendre un programme orienté objet pour traiter de nouvelles situations.
Par exemple, on peut dériver une nouvelle classe de forme appelée « pentagone » sans modifier les fonctions qui traitent des formes génériques. Cette capacité à étendre facilement un programme en dérivant de nouvelles sous-classes est important car il améliore considérablement la conception tout en réduisant le coût de maintenance.

Un problème se pose cependant en voulant traiter les classes dérivées comme leur classe de base générique (les ellipses comme des formes géométriques, les vélos comme des véhicules, les cormorans comme des oiseaux etc...). Si une fonction demande à une forme générique de se dessiner, ou à un véhicule générique de tourner, ou à un oiseau générique de se déplacer, le compilateur ne peut savoir précisément lors de la phase de compilation quelle portion de code sera exécutée. C'est d'ailleurs le point crucial : quand le message est envoyé, le programmeur ne *veut* pas savoir quelle portion de code sera exécutée ; la fonction dessiner peut être appliquée aussi bien à une ellipse qu'à un rectangle ou à un triangle et l'objet exécute le bon code suivant son type spécifique. Si on n'a pas besoin de savoir quelle portion de code est exécutée, alors le code exécuté (lorsque on ajoute un nouvelle sous-classe) peut être différent sans exiger de modification dans l'appel de la fonction. Le compilateur ne peut donc précisément savoir quelle partie de code sera exécutée, comment va-t-il réagir ? Par exemple, dans le diagramme suivant, l'objet **Contrôleur d'oiseaux** travaille seulement avec des objets **Oiseaux** génériques, et ne sait pas de quelle classe ils proviennent. C'est pratique du point de vue de **Contrôleur d'oiseaux** car il n'a pas besoin d'écrire du code spécifique pour déterminer le type exact d'**Oiseau** avec lequel il travaille, ou le comportement de cet **Oiseau**. Comment se fait-il alors que, lorsque **bouger()** est appelé tout en ignorant le type spécifique de l'**Oiseau**, on obtienne le bon comportement (une **Oie** court, vole ou nage, et un **Pingouin** court ou nage) ?



La réponse constitue l'astuce fondamentale de la programmation orientée objet : le compilateur ne peut pas faire un appel de fonction au sens traditionnel du terme. Un appel de fonction généré par un compilateur non orienté objet crée ce qu'on appelle une *association prédéfini*e. En d'autres termes, le compilateur génère un appel à un nom de fonction spécifique, et l'éditeur de liens résout cet appel à l'adresse absolue du code à exécuter. En POO, le programme ne peut pas déterminer l'adresse du code avant la phase d'exécution, un autre mécanisme est donc nécessaire quand un message est envoyé à un objet générique.
Pour résoudre ce problème, les langages orientés objet utilisent le concept d'*association tardiv*e. Quand un objet reçoit un message, le code appelé n'est pas déterminé avant l'exécution. Le compilateur s'assure que la fonction existe et vérifie le type des arguments et de la valeur de retour (un langage omettant ces vérifications est dit *faiblement typé*), mais il ne sait pas exactement quel est le code à exécuter.

Pour créer une association tardive, Java utilise une portion spéciale de code en lieu et place de l'appel absolu. Ce code calcule l'adresse du corps de la fonction, en utilisant des informations stockées dans l'objet. Ainsi, chaque objet peut se comporter différemment suivant le contenu de cette portion spéciale de code. Quand un objet reçoit un message, l'objet sait quoi faire de ce message.

Dans certains langages (en particulier le C++), il faut préciser explicitement qu'on souhaite bénéficier de la flexibilité de l'association tardive pour une fonction. Dans ces langages, les fonctions membres ne sont *pas* liées dynamiquement par défaut. Cela pose des problèmes, donc en Java l'association dynamique est le défaut et mot clef supplémentaire n'est pas requis pour bénéficier du **polymorphisme**.

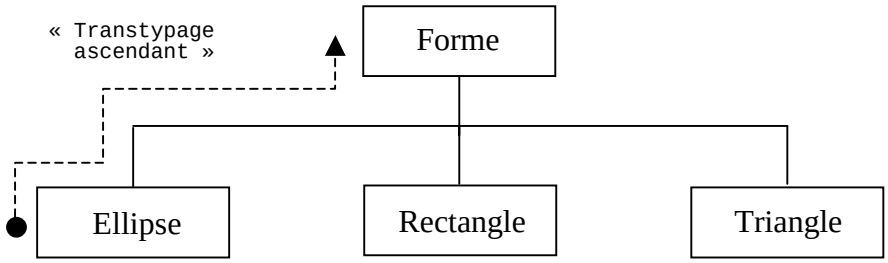
Reprenons l'exemple de la forme géométrique. Le diagramme de la hiérarchie des classes (toutes basées sur la même interface) se trouve plus haut dans ce chapitre. Pour illustrer le polymorphisme, écrivons un bout de code qui ignore les détails spécifiques du la classe et parle uniquement à la classe de base. Ce code est *déconnecté* des informations spécifiques à la classe, donc plus facile à écrire et à comprendre. Si une nouvelle classe - **Hexagone**, par exemple - est ajouté grâce à l'héritage, le code continuera de fonctionner pour cette nouvelle.
Si nous écrivons une méthode en Java :

```
void faireQuelqueChose(Forme f)
{
   f.effacer();
   // ...
   f.dessiner();
}
```

Cette fonction s'adresse à n'importe quelle **Forme**, elle est donc indépendante du type spécifique de l'objet qu'elle dessine et efface. Si nous utilisons ailleurs dans le programme cette fonction **faireQuelqueChose()** :

```
Cercle c = new Cercle();
Triangle t = new Triangle();
Ligne l = new Ligne();
faireQuelqueChose(c);
faireQuelqueChose(t);
faireQuelqueChose(l);
```

Les appels à **faireQuelqueChose()** fonctionnent correctement, sans se préoccuper du type exact de l'objet. Considérons la ligne : `faireQuelqueChose(c);`
Un **Cercle** est passé à une fonction qui attend une **Forme**. Comme un **Cercle** *est-une* **Forme**, il peut être traité comme tel par **faireQuelqueChose()**. C'est à dire qu'un **Cercle** peut accepter tous les messages que **faireQuelqueChose()** pourrait envoyer à une forme. C'est donc une façon parfaitement logique et sûre de faire. Traiter un objet de la classe dérivée comme s'il était un objet de classe de base est appelé *transtypage ascendan*t, *surtypage* ou *généralisation* (upcasting).
Un programme écrit en langage orienté objet contient obligatoirement des transtypages ascendants, car c'est de cette manière que la classe spécifique de l'objet peut être délibérément ignorée.



Dans le code de **faireQuelqueChose(),** remarquez qu'il ne dit pas « Si tu es un **Cercle**, fais ceci, si tu es un **Carré**, fais cela, etc... ». Ce genre de code qui vérifie tous les types possibles que peut prendre une **Forme** est confus et il faut le changer à chaque extension de la classe **Forme**. Ici, il suffit de dire : « Tu es une forme géométrique, je sais que tu peux te **dessiner()** et **t'effacer(**), alors fais-le et occupe-toi des détails spécifiques ». Ce qui est impressionnant dans le code de **faireQuelqueChose()**, c'est que tout fonctionne comme on le souhaite. Appeler **dessiner()** pour un **Cercle** exécute une portion de code différente de celle exécutée lorsqu'on appelle **dessiner()** pour un **Carré** ou une **Ligne**, mais lorsque le message **dessiner()** est envoyé à une **Forme** anonyme, on obtient le comportement idoine basé sur le type réel de la **Forme**. C'est impressionnant dans la mesure où le compilateur Java ne sait pas à quel type d'objet il a affaire lors de la compilation du code de **faireQuelqueChose()**. On serait en droit de s'attendre à un appel aux versions **dessiner()** et **effacer()** de la classe de base **Forme**, et non celles des classes spécifiques **Cercle**, **Carré** et **Ligne**. Mais quand on envoie un message à un objet, il fera ce qu'il a à faire, même quand la généralisation est impliquée. C'est ce qu'implique le polymorphisme. Le compilateur et le système d'exécution s'occupent des détails, et c'est tout ce que vous avez besoin de savoir, en plus de savoir comment modéliser avec.

3.- Le langage Java

3.1.- Les sources et les commentaires

À l'instar d'un programme C++, un programme écrit en Java est un ensemble de fichiers textes documentés (appelés **sources** et d'extension **.java**) respectant une grammaire précise bien qu'il existe toujours plusieurs façons d'écrire la même chose. Ce document présente la solution préconisée mais aussi les alternatives les plus fiables sous forme de **commentaires** (c'est-à-dire une suite de caractères, de préférence non accentués, qui seront ignorés par le compilateur et ne servent qu'à documenter le programme). On peut indiquer le début des commentaires et préciser qu'ils se terminent à la fin de la ligne courante ou bien (comme en C++) indiquer le début et la fin des commentaires éventuellement sur des lignes différentes :

// ce symbole indique le debut d'un commentaire qui se terminera a la fin de la ligne

ou bien

/ ce symbole indique le debut d'un commentaire qui peut être sur plusieurs lignes et ne se terminera qu'apres le symbole */*

En java il existe un troisième type de commentaire permettant au logiciel de documentation automatique (**javadoc**) de générer une documentation succincte sous forme de fichier HTML. Ces commentaires doivent commencer par **/**** et se terminer par ***/.** Ce document ne traite pas du logiciel **javadoc** mais il est possible de consulter la documentation complète de cet outil sur le site web de Sun (http://java.sun.com/products/jdk/1.2/docs/tooldocs/javadoc)

Un source est organisé de la manière suivante :

```
// Commentaires sur le contenu du fichier
// Indications des packages qui devront etre importes par le compilateur
/* Definition et corps des classes
  // La classe qui contient une méthode "main" est une application executable
*/
```

Pour l'affichage on dispose de fonctions appartenant au package système standard :

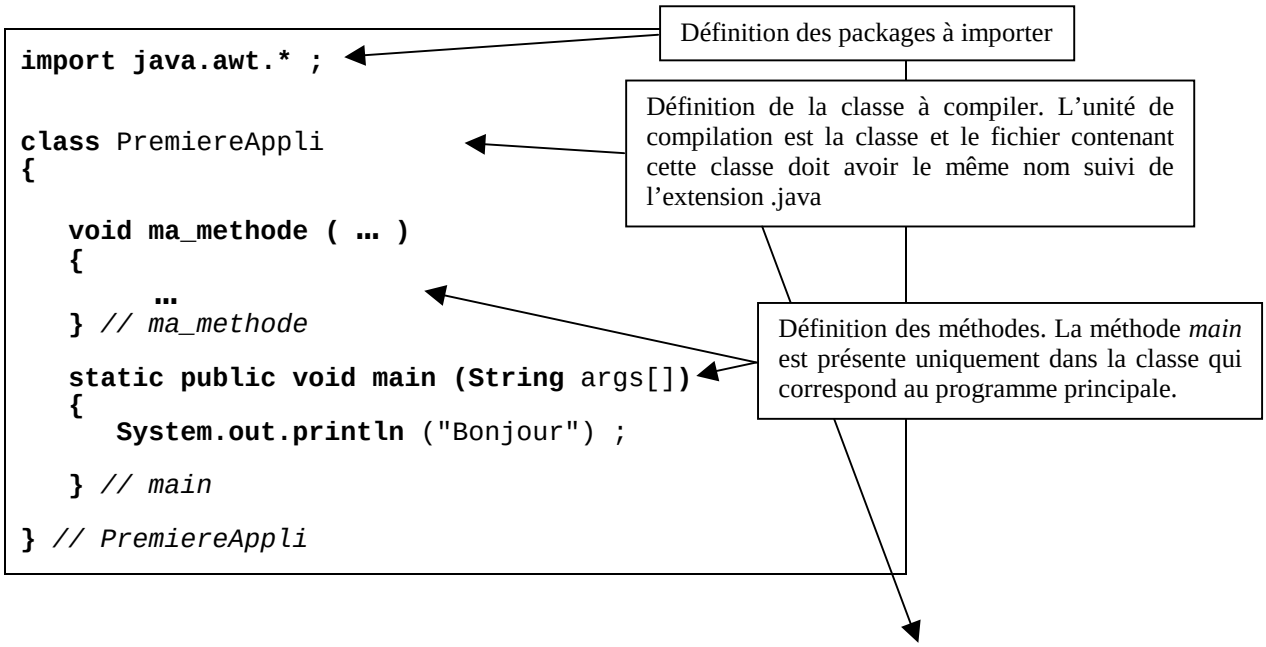
`System.out.print (``val`**`)`** : affiche à l'écran la variable *val* quelque soit le type de *val*.
`System.out.println (``val`**`)`** : affiche à l'écran la variable *val* quelque soit le type de *val* et effectue un retour chariot.

Exemples :

`System.out.println (``val`**`) ;`** affiche la valeur 4 à l'écran et effectue un retour chariot.
`System.out.print (``"valeur = "+ val`**`) ;`** affiche à l'écran : valeur = 4 (si val vaut 4).

Le « + » signifie que l'on concatène la chaîne de caractères `"valeur = "` et la chaîne représentant la variable `val`.

Ci-dessous le squelette très simple d'une application Java :



Le fichier contenant cette classe doit impérativement s'appeler **PremiereAppli.java.**

Nous reviendrons en détails sur les notions de **classe** et de **méthode** (Cf. chapitre 3.7) ainsi que sur l'utilisation des mots-clés **static, public** (Cf. chapitre 3.10). Un chapitre particulier est également consacré à la méthode **main** (Cf. chapitre 3.7.5).

3.2.- Types simples et opérations arithmétiques

Comme la plupart des langages de programmation, Java possède des types simples. Contrairement aux autres, Java impose que la représentation interne de ces types soit uniforme (même espace mémoire) qu'elle que soit la machine sur laquelle est exécuté le programme (gage de portabilité pour les applications écrites en Java).

3.2.1.- La déclaration des variables

déclaration : <type> <identificateur> ;

<identificateur> :

- lettres non accentuées **(attention minuscules ≠ majuscules)**
- chiffres (sauf le premier caractère)
- '_'

- attention à ne pas utiliser les mots-clefs du langage (**class**, **for**, **while**, **if**, **else**, ...)

<type> : entiers, décimaux, booléen, caractère.

3.2.2.- Les entiers

On distingue quatre type d'entiers : **byte, short, int et long.**

	byte	short	int	long
Taille (bits)	8	16	32	64
Etendue	-128 .. 127	-32768 .. 32767	-2^{31} .. $2^{31}-1$	-2^{63} .. $2^{63}-1$

3.2.3.- Les décimaux

On distingue deux type de décimaux : **float et double.**

	float	double
Taille (bits)	32	64
Etendue	1.4E-45 .. 3.4E+38	4.9E-324 .. 1.8E+308

3.2.4.- Le type booléen

On introduit une variable de ce type pas le mot clé **boolean**. Ce type accepte seulement deux états : l'un est nommé **true** et symbolise un état d'acceptation, l'autre, nommé **false**, symbolise un état de réfutation.

Attention, ce n'est plus comme en C : le type booléen n'est pas en Java, un sous-type numérique.

3.2.5.- Le type caractère

Ce type, introduit par le mot clé **char**, permet la gestion des caractères. Jusqu'à présent, la plupart des langages utilisaient à cet effet le codage ASCII. Ce codage permet de représenter seulement 128 caractères (on ne peut pas représenter les caractères accentués). Pour coder davantage de caractères, une nouvelle norme a été créée : le codage unicode. Ce nouveau standard utilise **16 bits** et permet donc de coder 65536 caractères. Les concepteurs de Java, dans un souci de portabilité du code produit, ont donc trouvé judicieux d'utiliser ce standard. Les exemples qui suivent vous donnent la syntaxe à utiliser pour décrire un caractère en Java.

'a'	'\t' pour un tab	'\u0521' un caractère quelconque

3.2.6.- Les opérations arithmétiques

On distingue cinq opérations arithmétiques fondamentales : **+, -, /, *, %**

Exemples :

```
int i ; // declaration
...
i = i + 1 ;   //  incrementation equivalente a :   i++ ;
i = i - 1 ;   //  decrementation equivalente a :   i-- ;
```

les pièges des raccourcis : i++ et ++i

```
int i, n ;    //  declarations
...
n = i++ ;    //  n prend la valeur i puis i est incremente
n = ++i ;    //  i est incremente puis n prend la valeur de i
```

les raccourcis peu explicites : +=, -=, /=, *****=, %=

```
int i, n ;    //   declarations
...
n += i ;    //  incrementation egale a   :   n = n + i ;
```

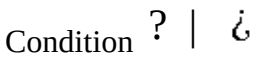
Les **opérations logiques** sur les bits (>>, <<, **&**, | , …) sont très peu utilisés en Java. Ces opérateurs sont issus du langage C. Ils permettent de réaliser des opérations directement sur les bits des nombres entiers. Ce document ne traite pas de ces opérations. Il s'agit d'un sujet « pointu » très bien expliqué dans la plupart des manuels de langage C.

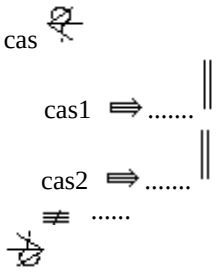
3.3.- Les instructions : de l'algorithme au programme

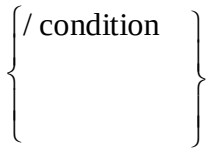
L'ADL ("Algorithm Descriptive Language") est un langage de description d'algorithmes très concis (pas de déclarations) proche de la sténographie. Pourquoi faire le parallèle avec L'ADL ?

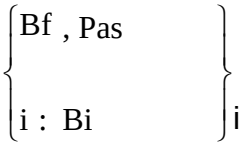
- ce langage est utilisé à l'IGN et surtout à l'ENSG ;
- ce langage est concis et facile à comprendre ;
- cette méthode de structuration des algorithmes est facile à transposer en programmation ;
- les algorithmes des exercices seront présentés en ADL (universalité des cours).

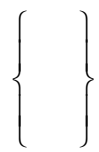
Signification	**ADL**	**Java**
Affectation	X←3	`x = 3 ;`

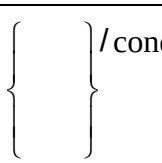
Test		`if (condition)`
		`{`
		`  instructions ;`
		`}`
		`else`
		`{`
		`  instructions ;`
		`} // if`

Test à choix multiple		`switch (cas)`
		`{`
		`  case cas1 : instructions ;`
		`  break ;`
		`  case cas2 : instructions ;`
		`  break ;`
		`  default : instructions ;`
		`} // switch`

Boucle "tant que"		`while (condition)`
		`{`
		`  instructions ;`
		`} // while`

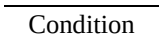
Boucle avec compteur		`for (i=Bi;i<=Bf;i=i+Pas)`
		`{`
		`  instructions ;`
		`} // for i`

Boucle infinie		`for (; ;)`
		`{`
		`  instructions ;`
		`} // for`

Boucle "jusqu'à ce que"		`do`
		`{`
		`  instructions ;`
		`} while ! ( condition) ;`

Sortie de boucle		`break ;`
Au suivant		`continue ;`
Sortie de procédure		`return ;`

Autres débranchements		`goto debranchement;`
		`  instructions`
		`debranchement:`
		`  instructions`

vrai		`true`
faux		`false`
non		`! (condition)`
et	∩	`&&`
ou	∪	`||`
égal	=	`==`
différent	≠	`!=`

Les principales remarques :

- l'affectation "=" ne doit pas être confondue avec le test d'égalité "==" ;
- dans les tests simples, le bloc "**else**" est facultatif et les limites d'un bloc sont facultatives s'il ne contient qu'une instruction (piège pour la maintenance) ;
- dans les tests à choix multiples, le test doit porter sur une valeur d'un type discret (entier ou booléen), le dernier "**break**" est facultatif (pas recommandé) et l'absence de séparation ("**break**") entre 2 choix ("**case**") entraîne l'exécution des instructions du 2ème choix (pas de débranchement implicite) ;
- la boucle "**while**" est la boucle de base du langage (la boucle "**for**" en dérive) ;
- la sortie de programme est "**return**" (sortie de la fonction principale), "**exit**" est un arrêt *violent* ;
- pas de débranchement de niveau supérieur à 1 (utiliser "**goto**" avec parcimonie) ;
- dans les tests, en cas d'égalité de priorité, le membre le plus à gauche est évalué en premier (les parenthèses sont fortement conseillées), les autres membres peuvent ne pas être évalués (compilation ANSI ou non).

3.4.- Les conversions de type

3.4.1.- Conversion des types simples

Deux cas de figures se présentent : convertir vers un type « plus grand » et convertir vers un type « plus petit ». Dans le premier cas, il n'est pas nécessaire d'expliciter la conversion, l'affectation suffit. Exemple : la conversion d'un « int » en un « long ».

```
long b ;
int a ;
b = a ;  // cette instruction suffit a convertir l'entier a en un entier long
```

Toutefois, la conversion d'entiers en valeurs à virgule flottante fait exception à cette règle générale. La conversion d'un « int » en un « float » ou un « double » peut entraîner une perte de précision.

Dans le deuxième cas, il faut expliciter la conversion en indiquant entre parenthèses le type dans lequel on désire convertir. Exemple : la conversion d'un « long » en un « int ».

```
long b ;
int a ;
a = (int)b ;  // on « force » b, a devenir un entier simple
```

3.4.2.- Conversion des objets

Un objet d'une classe (Cf. chapitre 3.6) peut également être converti en un objet d'une autre classe, à une condition : les classes en question doivent être liées par le mécanisme de l'héritage que nous verrons plus loin (Cf. chapitre 3.9). Mis à part cette restriction, le mécanisme de conversion est identique à celui utilisé pour les variables de type simple.
Voici un exemple de conversion d'une instance de la classe « sapin » vers la classe « arbre » (où « sapin » est une sous-classe de la classe « arbre »).

```
Sapin s ;
Arbre a ;

s = new Sapin() ; // on cree un objet s de la classe Sapin
a = (Arbre)s ;  // on « force » s, a devenir un objet de type Arbre
```

3.4.3.- Des types simples vers les objets et vice versa

En Java, il est impossible de convertir des variables de type simple en objet et vice versa. Cependant, Java fournit un package (**java.lang**) contenant des classes spéciales correspondant à chaque type simple (ex : **Integer** pour les entiers, **Float** pour les flottants …). Grâce aux méthodes de classes définies dans ces classes, ont peut créer, pour chaque variable de type simple, un objet correspondant de la classe. Exemple : création d'un objet de type **Integer** à partir de l'entier *35* .

```
Integer aObj  = new Integer (35) ;
```

On dispose maintenant d'un véritable objet *aObj*. Il s'avère souvent nécessaire de revenir aux valeurs de type simple. Il existe des méthodes pour réaliser ces opérations.

```
int a  = aObj.intValue() ; // renvoie 35
```

3.5.- Tableaux et chaînes de caractères

Un tableau est la succession d'un certain nombre d'éléments d'un même type (n'importe quel type). On peut par exemple définir un tableau (une succession) de dix éléments de type **float**.

Notons qu'un tableau d'éléments d'un type donné possède lui aussi un type : le type tableau d'élément du type initial. On peut donc créer des tableaux de tableaux. C'est ce mécanisme qui permet de simuler les tableaux multidimensionnels.

Vous devez déclarer une variable comme étant de type tableau de quelque chose, sinon vous ne pourrez plus, ultérieurement, accéder à celui-ci. Pour ce faire, vous devez placer l'opérateur **[]** dans une déclaration de variable. Soit vous placez l'opérateur immédiatement après le type, soit vous le placez derrière chaque variable. Dans le premier cas l'opérateur agira sur toutes les variables de la déclaration, dans le second cas, il agira uniquement sur la variable qui le précède.

Vous devez allouer de la mémoire pour contenir les données de ce tableau. Pour ce faire, on utilise l'opérateur **new** que nous étudierons plus tard (Cf. chapitre 3.6.2). Cet opérateur doit forcément connaître la taille du tableau pour le créer : on la spécifie directement après la variable (entre crochets). Il est possible de réunir ces deux étapes en une seule opération, comme le montrent les exemples ci-dessous.

int monTableau[]; nomTableau = **new int**[10];	un tableau de 10 entiers
int tableau[][] = **new** tableau[10][];	un tableau de 10 tableaux d'entiers
int a, b[10];	un entier (a) et un tableau de 10 entiers (b)
int[] monTableau2 = **new int**[10];	un tableau de dix entiers
int[] a, b[];	un tableau d'entiers non alloué (a) et un tableau de tableaux d'entiers non alloué (b). Si l'opérateur **[]** se trouve avant les variables, il agit sur toutes les variables de la déclaration.
int a[], b[][];	cette déclaration est équivalente à la précédente.

Le type chaîne de caractères (String)

Première remarque par rapport au langage C/C++, le type chaîne de caractères est totalement différent du type tableau de caractères. Le type chaîne de caractères est une classe proprement dite (**String)**. Pour ce qui est de la syntaxe d'une chaîne de caractères, rien n'a changé par rapport à C/C++ : une chaîne de caractères commence et se termine par un caractère « double guillemets ». Ainsi, *"Bonjour"* est une chaîne de caractères.

Le programme ci-dessous illustre la déclaration et l'utilisation des chaînes de caractères.

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

    String ch = "Cours de Java";


    System.out.println("chaine = "+ch);
  }
}
```

Définition et affectation d'un chaîne de caractères. La variable *ch* devient de fait une variable de 13 caractères.

à l'écran → `chaine = Cours de java`

3.6.- Classes et objets

Comme nous l'avons décrit au chapitre 2 de ce document, le langage Java est "orienté objet", il est entièrement basé sur la notion d'**encapsulation**, c'est-à-dire qu'il permet de définir des **classes** qui contiennent des **membres**. Les membres sont soit des **champs** soit des **méthodes**. **L'instance** de la classe (appelé **objet**) est alors implicitement passée en paramètre à la méthode.

Les champs (parfois appelé attribut) sont des variables qui définissent des caractéristiques propres à chaque objet (variables d'instance) ou à un groupe d'objets de la classe (variables de classe). Ces variables ont une **portée** différente suivant l'endroit où elles sont déclarées.

Les méthodes sont des fonctions qui définissent les comportements des objets de la classe.

Contrairement au C/C++, Java est « purement » orienté objet, c'est-à-dire qu'il est impossible de définir des fonctions autres que des méthodes. Toutes les classes Java héritent d'une classe de base unique : la classe `Object` du package `java.lang`.

3.6.1.- Déclaration d'une classe

En Java, la déclaration d'une classe est très simple :

Fichier "Point2D.java"

```
class Point2D {

}
```

Il est possible d'imbriquer des classes dans des classes (Cf. chapitre 5.6.6 - Les classes intérieures).

3.6.2-. Création d'un objet d'une classe : l'opérateur new

Pour qu'un objet puisse réellement exister au sein de la machine, il faut qu'il puisse stocker son état dans une zone de la mémoire. Or, deux objets définis à partir de deux classes différentes n'ont pas forcément besoin du même espace mémoire. La seule manière de créer un objet d'une classe en Java est d'utiliser l'opérateur **new**. Cet opérateur doit être suivi du nom de la classe dans lequel l'objet est crée. L'opérateur **new** détermine la place mémoire nécessaire à la définition d'un objet. La création d'un objet est parfois appelée **instanciation**.

Voici quelques exemples d'instanciation d'objets :

```
String chaine = new String () ;
Point2D point = new Point2D () ;
String chaine2 = new String ("Bonjour") ;
```

Les opérations réalisées par new

L'appel à l'opérateur new déclenche une série d'actions :

- création de l'objet d'une classe ;
- allocation de la mémoire nécessaire a cet objet ;
- appel d'une méthode particulière définie dans la classe : le **constructeur** (Cf. chapitre 3.8). Les constructeurs (il peut y en avoir plusieurs pour une classe donnée) servent à créer et à initialiser les nouvelles instances de la classe. Les constructeurs initialisent également tous les autres objets dont l'objet créé a besoin pour être initialisé.

3.6.3.- Les variables d'objet (ou d'instance de classe)

Ces variables sont déclarées dans une classe mais en dehors d'une méthode et sont globales pour tous les objets de la classe.

Fichier "Point2D.java"

```
class Point2D {
  double x,y;
}
```

Fichier "Cercle.java"

```
class Cercle {
  Point2D centre = new Point2D();
  double rayon;
  Color couleur;
}
```

Fichier "Program.java"

```
class Program {

  static public void main (args[]) {
    Cercle C1 = new Cercle() ;
    Cercle C2 = new Cercle() ;

    C1.centre.x = 3.45;
    ...
    C2.centre.y = 3.56;
    ...
    C1.couleur = Color.red ;
    C2.couleur = Color.red ;
    ...
  }
}
```

On crée un objet *Point2D*. Sans cette création, le programme génère une erreur (mémoire non allouée) lors de l'affectation d'un des champs d'un objet de la classe.

3.6.4.- Les variables de classe

Ces variables sont globales pour une classe et pour toutes les instances de cette classe. Les variables de classe sont utiles pour faire communiquer entre eux différents objets d'une même classe ou pour définir des propriétés communes à un groupe d'objets de la même classe. Pour différencier les variables d'objet des variables de classe on utilise le mot-clé **static**

Fichier "Point2D.java"

```
class Point2D {
  double x,y;
}
```

Fichier "Cercle.java"

```
class Cercle {
  Point2D centre = new Point2D();
  double rayon;
  static Color couleur;
}
```

Le champ *couleur* est une variable de classe

Fichier "Program.java"

```
class Program {

  public void main (args[]) {
     Cercle C1 = new Cercle() ;
     Cercle C2 = new Cercle() ;

     C1.centre.x = 3.45;
     C1.centre.y = 3.56;

     Cercle.couleur = Color.red ;
     ...
  }
}
```

On applique une couleur à **tous** les objets de la classe.

3.6.5.- Référence aux objets

Même si la manipulation des références aux objets reste transparente pour le programmeur Java, il est intéressant d'en comprendre le mécanisme. Examinons un programme très simple :

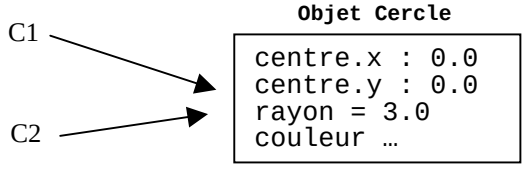
Fichier "TestReference.java"

```
class TestReference {

  public static void main (args[]) {

     Cercle C1, C2 ;
     C1 = new Cercle() ;
     C2 = C1 ;

     C1.rayon = 3.0;

     System.out.println ("r1 = "+C1.rayon) ;
     System.out.println ("r2 = "+C2.rayon) ;
  }
}
```

à l'écran

```
r1 = 3.0
r2 = 3.0
```

Dans la première partie du programme, on déclare deux variables de type *Cercle* (*C1* et *C2*), on crée un objet *Cercle* affecté à *C1* et on affecte à *C2* la valeur de *C1*. Comme on peut le constater par le résultat à l'écran, la variable d'instance *x* a été modifiée pour les deux objets. L'affectation de la valeur de *C1* à *C2* n'a pas créé un nouvel objet mais simplement une **référence à l'objet pointé** par *C1* (Cf. figure ci-dessous). Pour créer deux objets distincts, il **faut** utiliser l'opérateur **new** pour chacun des objets.



L'utilisation des références prend toute son importance lorsque les arguments sont transmis aux méthodes (Cf. chapitre 3.7.4)

3.6.6.- Les classes intérieures

Depuis la version 1.1, Java a introduit la notion de classes intérieures (***inner classes***). Java autorise en effet la définition de classes à l'intérieur d'une classe. L'exemple suivant montre comment créer ce type de classe.

```
class TopLevel {

  class Internal {
    int attribut=0;
  }

  static public void main(String argv[]){

    TopLevel tl=new TopLevel();

    Internal i=tl.new Internal();

    ...
  }
}
```

on ne peut créer un objet instancié sur une classe intérieure qu'à partir d'un objet instancié sur la classe englobante

Si l'on ne mentionne pas d'objet pour appliquer l'opérateur **new**, c'est la classe courante qui est prise en considération. On peut alors écrire :

```
class TopLevel {

  class Internal {
    int attribut=0;
  }

  Internal internal() {
      return new Internal();
  }

  static public void main(String argv[]){

    TopLevel tl=new TopLevel();

    Internal i=tl Internal();

    ...
  }
}
```

3.7.- Les méthodes

3.7.1.- Utilisation des méthodes

La définition du comportement d'un objet passe par la création de méthodes (ensemble d'instructions Java exécutant certaines tâches). Les méthodes sont définies dans une classe et accessibles seulement dans cette classe. Contrairement au C++, Java ne possède pas de fonctions définies en dehors des classes. L'appel des méthodes est similaire à la référence aux variables d'instances dont nous avons parlé au chapitre précédent : les appels de méthodes utilisent aussi la notation pointée.
Ajoutons une méthode **`Move`** (qui permet de déplacer un objet de la classe Cercle) à notre classe Cercle.

Fichier "Point2D.java"

```
class Point2D {
  double x,y;
}
```

Fichier "Cercle.java"

```
class Cercle {
  Point2D centre = new Point2D ();
  double rayon;
  Color couleur;

  void Move(double x ,double y) {
    centre.x = centre.x + x;
    centre.y = centre.y + y;
  }
}
```

Définition de la méthode **`Move`**

Fichier "Program.java"

```
class Program {

  public void main (args[]) {
     Cercle C = new Cercle() ;

     C.centre.x = 3.45;
     C.centre.y = 3.56;

     C.Move(4.0,3.0) ;
     ...
  }
}
```

Appel de la méthode **`Move`** appliquée à l'objet C de la classe Cercle

Vous aurez sans doute remarqué l'utilisation de mot clé **void**. Celui-ci sert à définir un type de retour nul (sans valeur) pour la méthode. Si une méthode doit retourner une valeur (non nulle), il faut spécifier son type à cet endroit précis de la définition de la méthode.

3.7.2.- Les méthodes de classe

Comme les variables de classe, les méthodes de classe (à déclarer en `static`) s'appliquent à la toute classe et non à ses instances. Ces méthodes servent surtout à grouper au même endroit, dans une classe, des méthodes à caractère général. Par exemple, la classe **`Math`** définie dans **`java.lang`** contient de nombreuses méthodes de classe. Il n'existe pas d'instance de la classe **`Math`**, mais on peut utiliser directement ses méthodes. Par exemple la méthode **`max(…)`** :

```
int a = Math.max (x, y) ;
```

3.7.3.- surcharge des méthodes

Le polymorphisme autorise la définition de plusieurs méthodes ayant le même nom dans la mesure où les paramètres sont différents (nombres de paramètres différents ou types différents).
Ainsi, pour définir un déplacement du cercle uniquement en x, vous pouvez définir une autre méthode **`Move`** n'ayant qu'un seul paramètre. Il est également possible de définir une méthode **`Move`** qui effectue un déplacement par défaut en x et en y.

Fichier "Cercle.java"

```
class Cercle {
  Point2D centre = new Point2D() ;
  double rayon ;
  Color couleur ;

  void Move(double x ,double y) {
    centre.x = centre.x + x ;
    centre.y = centre.y + y ;
  }

  void Move(double x) {
    centre.x = centre.x + x ;
  }

  void Move() {
    centre.x = centre.x + 1.0 ;
    centre.y = centre.y + 1.0 ;
  }
}
```

Fichier "Program.java"

```
class Program {

  public void main (args[]) {
     Cercle C = new Cercle() ;

     C.centre.x = 3.45 ;
     C.centre.y = 3.56 ;

     C.Move(4.0,3.0) ;
     ...
     C.Move(2.0) ;
     ...
     for (int i=1;i<=10;i++)
       C.Move() ;
  }
}
```

Appel des trois méthodes **`Move`** appliquée à l'objet C de la classe Cercle

3.7.4.- Le passage des paramètres

Lorsque vous appelez une méthode dont les arguments comportent un ou plusieurs objets, les variables qui sont passées au corps de la méthode sont passées par **référence**. Ceci signifie qu'il n'y a pas de copie locale (à la méthode) de l'objet passé en paramètre. Par conséquent, toute manipulation de l'objet à l'intérieur de la méthode affecte l'objet original. Cette règle de passage des paramètres s'applique à tous les objets ainsi qu'aux tableaux. Seul les variables de type simple (entiers, réels, booléens et caractères) sont passées par valeur (on recopie localement la variable et on travaille sur cette copie).

3.7.5.- Une méthode particulière : la méthode « main »

Comme nous l'avons déjà dit en présentant le langage, on lance l'exécution d'un programme Java en démarrant une machine virtuelle Java avec, en paramètre, le nom de la classe de démarrage. Cette classe doit impérativement contenir une méthode **main**. Une fois que la machine virtuelle s'est mise en place, elle lance le programme proprement dit par la première instruction de la méthode **main**, et ce, sans instancier d'objet à partir de la classe de démarrage. Ceci est possible car la méthode **main** est déclarée **static** : c'est à dire qu'elle existe sans qu'il y ait eu instanciation. La tâche principale de cette méthode est alors d'instancier des objets sur différentes classes afin que le programme puisse s'exécuter. Comme la méthode **main** existe indépendamment de toute instance de classe, si elle doit utiliser des attributs ou des méthodes de la classe, il faut que ces champs soient eux aussi déclarés **static**, sans quoi, ils n'existent pas. Plus formellement, les méthodes déclarées statiques, sur une classe, ne peuvent manipuler que des champs statiques.

Notons au passage que la méthode **main** admet en paramètre un tableau de chaînes de caractères ("`String argv[]`"). Celui-ci contient les éventuelles options spécifiées sur la ligne de commande lors du lancement de la machine virtuelle. Pour connaître la taille du tableau, il suffit de récupérer la valeur renvoyée par **argv.length**. A titre d'information, le nom du paramètre peut être n'importe quel nom, mais on utilise souvent **argv**, repris de la terminologie du langage C/C++.

Fichier "Start.java"

```
class Start {
  static int a = 3;

  static public void main(String argv[]) {
    a = a + 5;
    System.out.println("a^2 = " + Square(a));
  }

  static int Square(int value) {
    return value*value;
  }
}
```

3.8.- Les constructeurs et le destructeur

3.8.1.- Les constructeurs

De manière basique, on peut dire qu'un constructeur est une méthode, d'une classe donnée, servant à créer des objets. Contrairement aux méthodes, un constructeur n'a pas de type de retour et il a nécessairement le même nom que la classe dans laquelle il est défini. De même que les autres méthodes, les constructeurs acceptent la surcharge. L'exemple suivant propose donc quelques constructeurs pour nos classes déjà étudiées.

Fichier "Point2D.java"

```
class Point2D {
  double x,y;

  Point2D () { x=0.0, y=0.0 ; }
}
```

Le constructeur de la classe *Point2D* initialise les valeurs de *x* et de *y*.

Fichier "Cercle.java"

```
class Cercle {
  Point2D centre ;
  double rayon;
  Color couleur;

  void Move(double x ,double y) {
    centre.x = centre.x + x;
    centre.y = centre.y + y;
  }
  Cercle () {
    centre  = new Point2D();
    rayon = 2.0 ;
    couleur = Color.red ;
  }
  Cercle (double r, Color c) {
    centre  = new Point2D();
    rayon = r ;
    couleur = c ;
  }
}
```

On définit deux constructeurs d'objets de la classe *Cercle.* Notez que l'on construit un objet de la classe *Point2D* dans chacun des constructeurs.

Fichier "Program.java"

```
class Program {

  public void main (args[]) {

    Cercle C1 = new Cercle() ;
    Cercle C2 = new Cercle(3.0,Color.blue) ;

    C1.centre.x = 3.45;
    C1.centre.y = 3.56;

    C1.Move(4.0,3.0) ;
    ...
    C2.Move(3.4,3.6) ;
    ...
  }
}
```

C1 est un cercle de couleur rouge et de rayon 2.

L'utilisateur de la classe Cercle choisit le rayon et la couleur de *C2* (ici 3 et bleue)

Sur cet exemple, on peut remarquer qu'un constructeur peut créer des objets qu'il utilise. On s'aperçoit également que les constructeurs servent principalement à définir l'état initial des objets instanciés. Il est bien sûr possible d'initialiser les variables lors de leur déclaration, mais le constructeur permet de créer des objets « sur mesure ».

Deux règles fondamentales sur les constructeurs :

- Si aucun constructeur n'est spécifié, dans la définition de la classe, un constructeur par défaut vous est obligatoirement fourni, celui-ci n'admettant aucun paramètre.
- Si vous en définissez au moins un, le constructeur par défaut (qui n'admet pas de paramètres) n'est plus fourni. Si vous en avez utilité il vous faudra alors le définir explicitement.

3.8.2.- Le destructeur

Nous venons donc de voir que des constructeurs pouvaient être fournis pour permettre la création d'objets. Parallèlement, un destructeur (et un seul) peut être défini, pour être utilisé lors de la destruction de l'objet. Celui-ci doit forcément se nommer **finalize**, il ne prend aucun paramètre et ne renvoie aucun type (**void**). Voici un petit exemple :

```
class Point2D {
  ...
  void finalize() {
      System.out.println("Objet point (2D) détruit");
  }
  ...
}
```

3.8.3.- Le ramasse miettes (Garbage Collector)

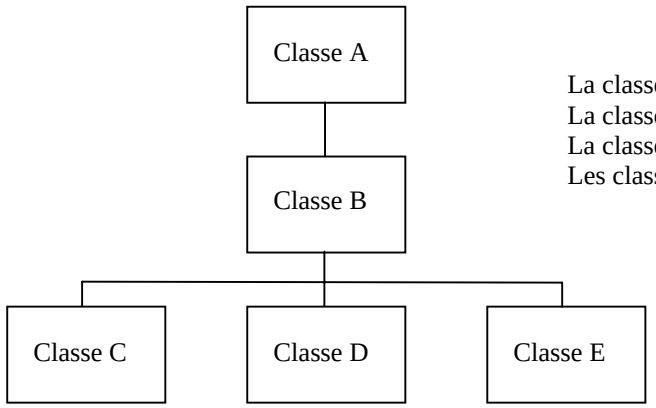
Un programme Java a besoin de mémoire pour pouvoir s'exécuter (en règle général, plus il en a, mieux c'est). Comme on l'a déjà vu, l'opérateur **new** se charge d'allouer de la mémoire à la demande. Une conséquence évidente est que si l'on ne libère pas la mémoire des objets devenus inutiles, on peut rapidement en manquer. Le ramasse-miettes (Garbage Collector) se charge de repérer ces objets inutiles, et de libérer la mémoire qu'ils utilisent inutilement. Il opère de façon totalement automatisé, et dans la quasi totalité des cas, vous n'avez pas à vous en soucier. N'oubliez pas que vous avez la possibilité de définir, par l'intermédiaire d'un destructeur, des actions à effectuer en cas de destructions d'objet.

3.9.- L'héritage

3.9.1.- Qu'est-ce que l'héritage ?

L'héritage est un mécanisme qui facilite l'écriture des classes et permet de rendre plus compact le code source d'un programme. Le concept est le suivant : toutes les classes font partie d'une hiérarchie stricte (figure ci-dessous). Chaque classe possède des super-classes (les classes situées au-dessus dans la hiérarchie) et un nombre quelconque de sous-classes (les classes situées au-dessous dans la hiérarchie). Les sous-classes héritent des attributs et des comportements de leurs super-classes, ce qui permet de ne pas réécrire le code concerné.

En Java, la classe Object se trouve au sommet de la hiérarchie des classes. Toutes les classes héritent de cette super-classe.



La classe A est la super-classe de B
La classe B est une sous-classe de A
La classe B est la super-classe de C, D et E
Les classes C, D et E sont des sous-classes de B

3.9.2.- Le fonctionnement de l'héritage

Comment des instances de classe accèdent-elles automatiquement aux méthodes et aux variables des classes situées au-dessus dans la hiérarchie ?

Pour les variables d'instances, à la création d'un nouvel objet, un emplacement est attribué à chaque variable définie dans la classe courante et dans toutes ses super-classes. Ainsi, la combinaison de toutes les classes constitue un modèle de l'objet courant. De plus, chaque objet apporte l'information appropriée à sa situation. Les méthodes opèrent de manière similaire. Les objets ont accès à tous les noms de méthodes de leur classe et de leurs super-classes. Par contre, les définitions de méthodes sont choisies dynamiquement (à l'exécution) au moment de l'appel. Si l'appel porte sur un objet particulier, Java commence par chercher la définition de la méthode dans la classe de l'objet. S'il ne la trouve pas, il poursuit sa recherche dans la super-classe et ainsi de suite (Cf. figure 1 page suivante).

Si deux méthodes ont la même signature (nom, nombre et type d'arguments) dans une sous-classe et dans une super-classe, Java exécute la première trouvée en remontant la hiérarchie. On dit alors que la méthode a été **redéfinie** (Cf. figure 2, page suivante).

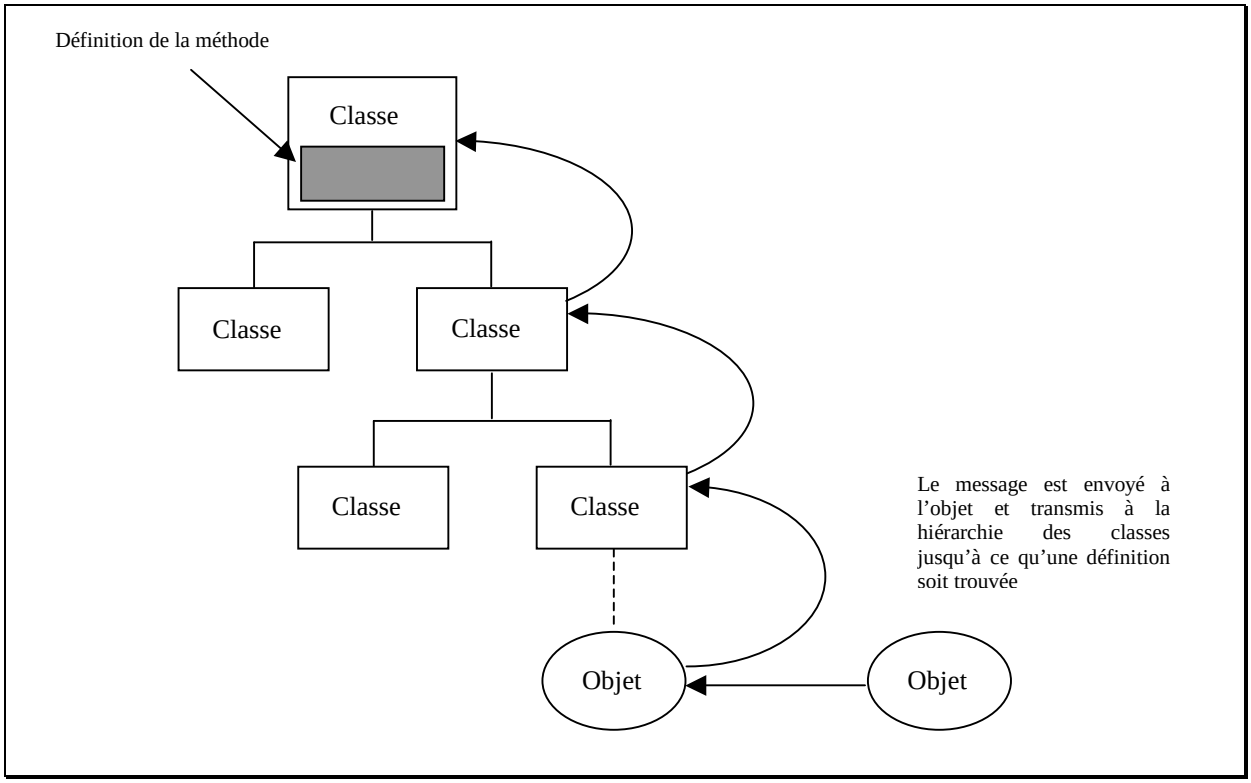


figure 1

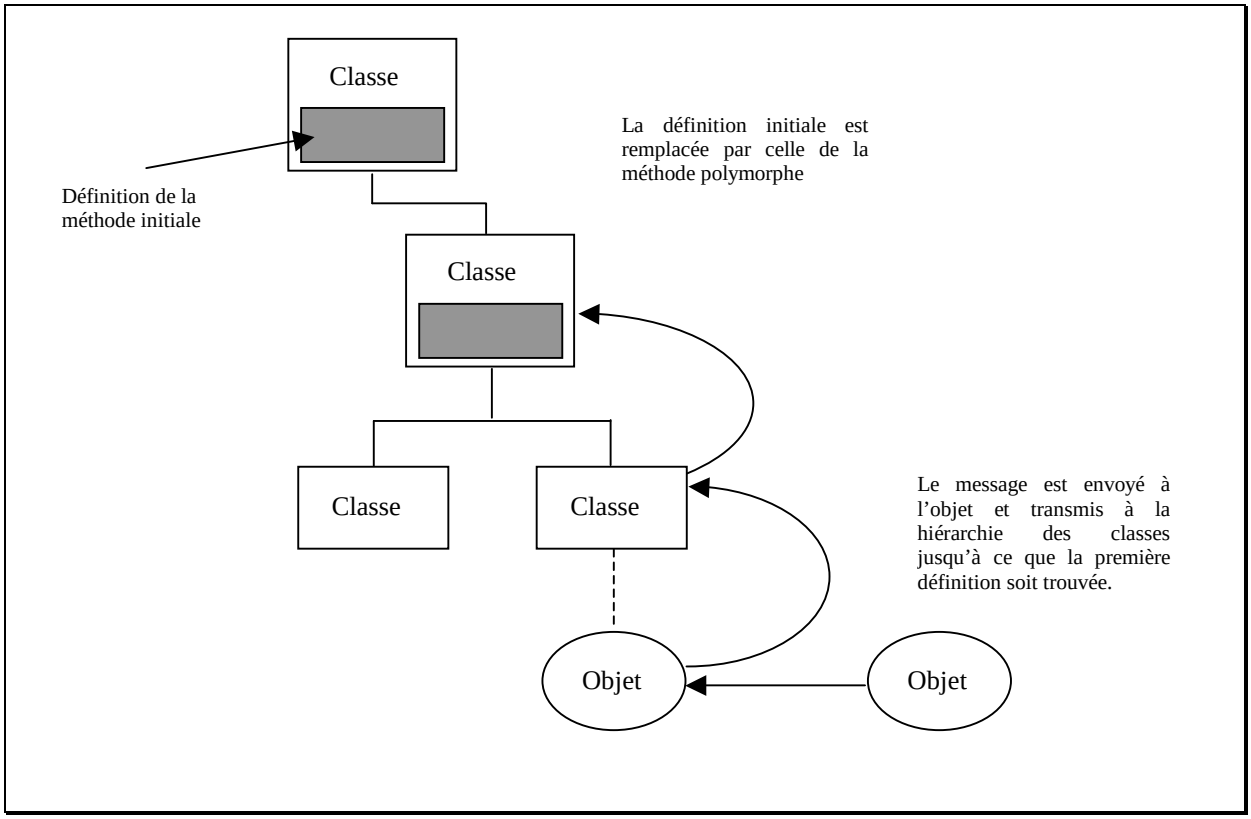


figure 2

3.9.3.- L'utilisation de l'héritage

Afin d'assimiler ce nouveau concept, considérons une extension du programme manipulant la classe *Cercle*. Il s'agit d'introduire de nouvelles classes permettant de manipuler non seulement des cercles mais aussi des carrés, des triangles et toute autre figure géométrique. L'idée la plus « basique » consiste à définir une classe *Cercle*, une classe *Carre*, une classe *Rectangle* … indépendantes les unes des autres. Ci-dessous un exemple de la définition d'une classe *Carre* et de notre classe *Cercle*.

```
class Carre {
  Point2D centre ;
  double cote ;
  Color couleur ;

  Carre() {
    centre = new Point2D() ;
    cote = 2.0;
    couleur = Color.red ;
  }
  Carre(double cote, Color c) {
     centre = new Point2D() ;
     this.cote = cote ;
     couleur = c;
  }
  void Move(double x ,double y) {
    centre.x = centre.x + x ;
    centre.y = centre.y + y ;
  }
  void Move(double x) {
    centre.x = centre.x + x ;
  }
  void Move() {
     centre.x = centre.x + 1.0 ;
     centre.y = centre.y + 1.0 ;
  }
}
```

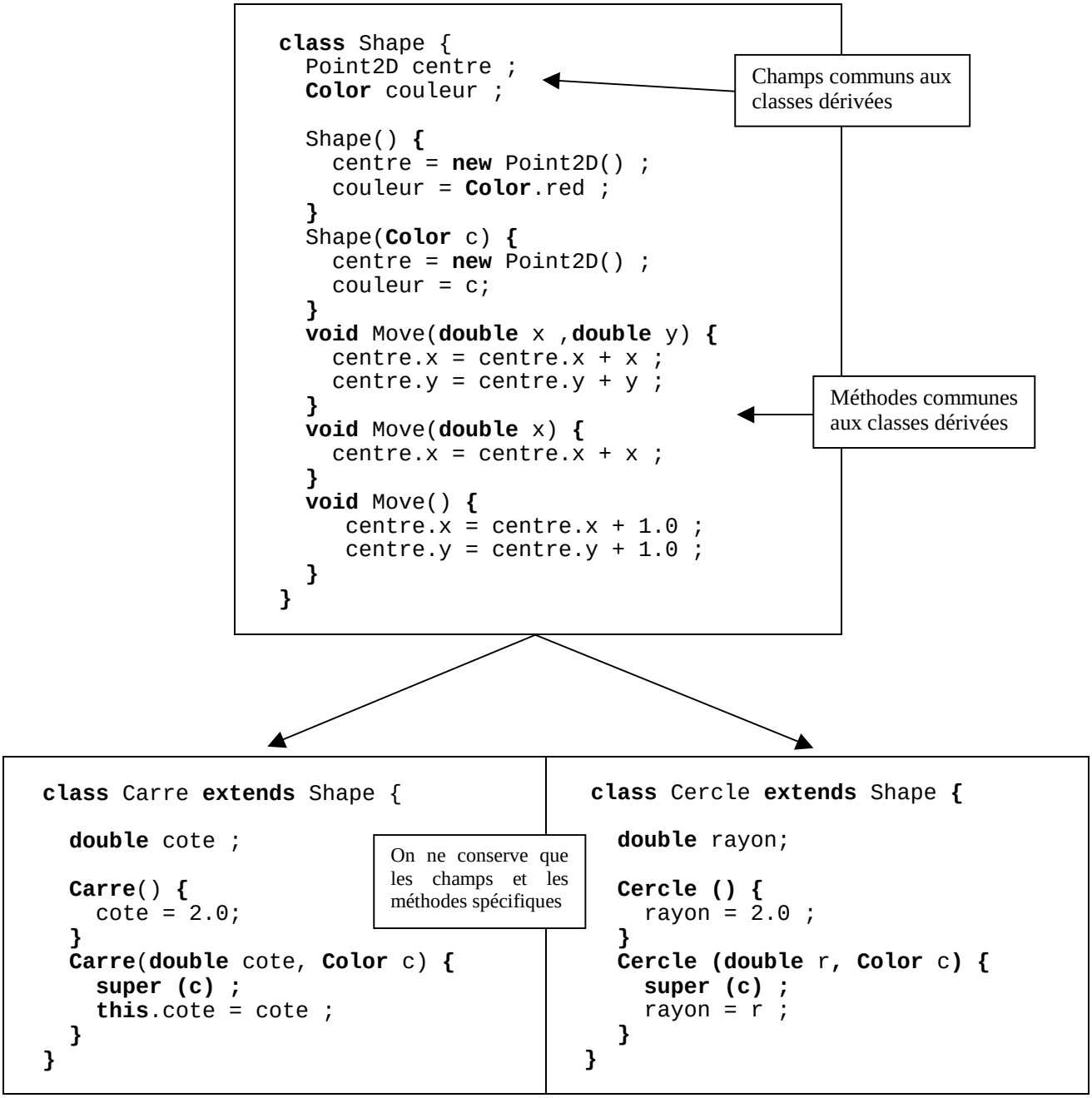
```
class Cercle {
  Point2D centre ;
  double rayon;
  Color couleur;

  Cercle () {
    centre  = new Point2D();
    rayon = 2.0 ;
    couleur = Color.red ;
  }
  Cercle (double r, Color c) {
    centre  = new Point2D();
    rayon = r ;
    couleur = c ;
  }
  void Move(double x ,double y) {
    centre.x = centre.x + x;
    centre.y = centre.y + y;
  }
  void Move(double x) {
    centre.x = centre.x + x;
  }
  void Move() {
    centre.x = centre.x + 1.0 ;
    centre.y = centre.y + 1.0 ;
  }
}
```

Si l'on regarde le code de la classe *Cercle* et celui de la classe *Carre*, on se rend compte immédiatement de la similitude de la plupart des champs et des comportements. L'héritage nous permet de regrouper les champs et des comportements communs dans une super-classe (appelons-là *Shape*) dont les classes *Cercle* et *Carre* vont **dériver**. On dit encore que les classes *Cercle* et *Carre* **héritent** des champs et des méthodes de la classe *Shape*.
En Java, pour signifier qu'une classe hérite d'une autre on utilise le mot-clé **`extends`.** Par exemple, si la classe *Cercle* hérite de la classe *Shape* on écrira :

```
class Cercle extends Shape {
   …
}
```

Un point important et souvent source d'erreur : on n'hérite en aucun cas des constructeurs. Si vous ne spécifiez pas explicitement un constructeur particulier, vous ne pourrez pas l'utiliser.

Après utilisation de l'héritage, notre exemple devient :



Les mots clés *super et this*.

Le mot clé ***super*** sert à accéder aux définitions de classe parente de la classe considérée (ces définitions pouvant être des méthodes ou des constructeurs).

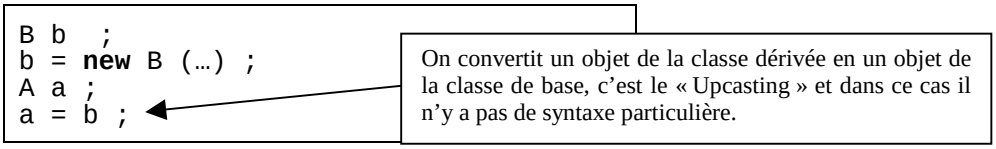
Le mot clé ***this*** sert à accéder aux membres de la classe courante.

Règles importantes concernant l'appel des constructeurs

- si la première instruction d'un constructeur ne commence pas par le mot clé *super***,** le constructeur par défaut de la classe mère est appelé (ne pas oublier de le définir).
- un appel au constructeur de la classe mère peut uniquement se faire en première instruction d'une définition de constructeur. Une conséquence évidente est qu'on ne peut utiliser qu'un seul appel au constructeur de la classe mère.

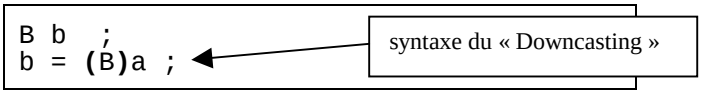
3.9.4.- Un mot sur les conversions de type

On distingue le **« Upcasting »** du **« Downcasting »**.
Supposons une classe abstraite A et deux classes dérivées de cette classe abstraite B et C. Les classes B et C peuvent être traitées comme des classes A, on peut donc écrire :



- Une sous-classe peut toujours être convertie dans le type de sa super-classe ;
- Une fois convertie dans ce sens, on ne peut accéder qu'aux champs définis dans la super-classe ;
- Les méthodes surchargées de la sous-classe peuvent toujours être appelées.

Dans le sens contraire, un objet de la classe A peut être converti en un objet de la classe B uniquement s'il a été défini au départ comme un objet de B et non de C. Dans ce cas (« Downcasting »), on utilise une syntaxe particulière (la même qu'en C/C++) :



Pour savoir à quelle classe appartient un objet, on utilise l'opérateur « **`instanceof` »** qui renvoie la valeur **true** ou **false**. Exemple :

```
if (a instanceof B) {
   … ;
}
```

3.9.5.- Avantages de l'héritage : résumé

Le premier point important est que l'héritage supprime, en grande partie, les redondances dans le code. Une fois la hiérarchie de classes bien établie, on localise en un point unique les sections de code (celles-ci restant à tous moments accessibles grâce au mot clé **super**).

La seconde chose importante (en considérant que la hiérarchie de classes a été bien pensée), est qu'il est possible de rajouter facilement une classe, et ce à moindre coût, puisque l'on peut réutiliser le code des classes parentes.

Dernier point, si un comportement n'a pas été modélisé dans une classe donnée, et qu'il est nécessaire de le rajouter, une fois l'opération terminée, ce comportement sera directement utilisable dans l'ensemble des sous-classes de la classe en question.

3.9.6.- L'héritage multiple

La forme d'héritage étudiée précédemment s'appelle « l'héritage simple ». Dans un tel héritage, chaque classe Java a une seule super-classe directe. Dans d'autres langages (notamment en C++), les classes peuvent avoir plusieurs super-classes. Elles héritent alors de la combinaison des variables et des méthodes de toutes leurs super-classes. Cela s'appelle « l'héritage multiple ». Il s'agit d'un mécanisme puissant mais très complexe qui alourdit souvent le code. Pour cette raison, Java se limite à l'héritage simple. Nous verrons dans le chapitre 5.12 qu'il est possible de simuler l'‘héritage multiple sans en avoir les inconvénients grâce au concept **d'interface**.

3.10.- Les modificateurs

Les modificateurs sont des mots-clé spéciaux du langage qui modifient la définition et le comportement d'une classe, d'une méthode ou d'une variable. On peut les classer en quatre grandes catégories :

- les modificateurs permettant de contrôler l'accès à une classe, une méthode ou une variable : **`public`**, **`private`** et **`protected`** ;
- le modificateur **`static`** (dont nous avons déjà parlé) utilisé pour les variables et les méthodes de classe ;
- le modificateur **`final`** permettant de « figer » l'implémentation d'une classe, d'une méthode ou d'une variable ;
- le modificateur **`abstract`** pour la création de classes ou de méthodes abstraites (que nous verrons plus tard).

Il existe d'autres modificateurs que nous ne détaillerons pas : **`synchronized`** et **`volatile`** (utilisés pour les *threads*) et **`native`** (utilisé pour la création de *méthodes natives*).

3.10.1.- Le contrôle d'accès

La notion de package (Cf. chapitre 3.11)

Aux trois modificateurs de contrôle d'accès (**`public`**, **`private`** et **`protected`**), il est utile d'ajouter le mot **`package.`** Bien que **`package`** ne soit pas un modificateur en soi, il comporte un niveau implicite de protection d'accès.

Un package permet d'organiser des groupes de classes et n'est véritablement utile que si le programme comporte un grand nombre de classes. Au lieu de la protection au niveau des fichiers (comme en C/C++), Java possède le concept de package appliqué à un groupe de classes ayant un même objectif et des fonctions similaires. Les méthodes et les variables bénéficiant de la « protection » des packages sont visibles par toutes les autres classes du package mais pas par les classes d'autres packages. Notez que Java comporte ses propres packages qu'il est possible d'utiliser dans vos programmes en important le ou les packages utilisés. Exemple : importer le package contenant les classes permettant de gérer les entrées-sorties.

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

Le modificateur `public`

On utilise le mot clé **public** pour signifier qu'une classe, une méthode ou une variable sont accessibles n'importe où dans « l'univers des classes Java » (même dans d'autres packages). Le mot-clé **public** est le mode par défaut.

```
class Carre {
   double cote ;
   void Move { ... }
   ...
}
```

Est équivalent à →

```
public class Carre {
   public double cote ;
   public void Move { ... }
   ...
}
```

Le modificateur `private`

Diamétralement opposé au modificateur **public**, il s'agit de la forme de protection la plus restrictive qui limite la visibilité des méthodes et des variables d'instance à la classe dans laquelle elles sont définies. Notons que les sous-classes ne peuvent hériter ni de variables privées, ni de méthodes privées.

```
class Carre {
   private double cote ;
   private void Move { ... }
    ...
}
```

Une règle pratique d'utilisation de la protection **private** veut que toute donnée ou comportement interne à une classe, que les autres classes ou sous-classes n'ont pas besoin de modifier, doit être **private**. Il faut bien garder à l'esprit que le rôle de l'encapsulation est justement de cacher à la vue du monde extérieur certaines données propres aux objets afin de sécuriser leurs manipulations.

Une autre règle veut que toutes les variables d'instance d'une classe soient privées et que pour obtenir ou pour changer ces variables on crée des méthodes spéciales non privées.

Le modificateur `protected`

Il s'agit d'une forme de protection entre classe et sous-classes. Le mot-clé **protected** signifie que des méthodes et des variables d'une classe donnée demeurent accessibles à toute classe du même package (différence avec C++), mais ne sont accessibles, en dehors du package, qu 'aux sous-classes de la classe à laquelle elles appartiennent.

Résumé des modes de protection

Visibilité	`public`	`protected`	`private`	`package`
Depuis la même classe	oui	oui	oui	oui
Depuis tout autre classe du même package	oui	oui	non	oui
Depuis tout autre classe extérieure au package	oui	non	non	non
Depuis une sous-classe du même package	oui	oui	non	oui
Depuis une sous-classe extérieure au package	oui	oui	non	non

3.10.2.- Le modificateur « static »

Ce modificateur est utilisé pour définir des variables ou des méthodes de classe (Cf. chapitre 3.7.2). Le mot-clé **static** indique simplement que l'élément est stocké dans la classe. Il est possible de « mixer » des modificateurs de contrôle d'accès avec **static**.

```
class Cercle {
   public static float pi = 3.14 ;
   void Move { ... }
    ...
}
```

3.10.3.- Le modificateur « final »

Ce modificateur s'applique aux classes, aux méthodes et aux variables :

- appliqué à une classe, **final** empêche le « sous-classement » de la classe ;
- appliqué à une méthode, **final** signifie que la méthode ne peut pas être redéfinie par les sous-classes ;
- appliqué à une variable, **final** indique que la « variable » est constante.

On utilise très rarement **final** pour les classes. Le seul intérêt est l'efficacité. En effet, dans le cas de classes finales, on a la certitude que les instances n'appartiennent qu'à cette classe et on peut ainsi optimiser le code relatif à ces instances. Il en est de même pour l'utilisation de **final** pour les méthodes. Par contre, **final** est très utilisé pour les déclarations de constantes.

```
class Cercle {
   public static final float pi = 3.14 ;
   public final String ConstStr = "Bonjour" ;

   ...
}
```

A noter qu'une méthode déclarée **private** est par définition finale, puisqu'on ne peut pas la dériver. Le fait de déclarer une méthode **private final** est donc redondant mais autorisé par le compilateur.

3.11.- Les packages

Les packages permettent d'organiser des groupes de classes. Un package contient un nombre quelconque de classes reliées entre elles par leur objectif, leur portée ou leur héritage.

Les packages deviennent intéressants à utiliser dès que l'application contient beaucoup de classes.

Plusieurs raisons d'utiliser les packages :

- ils permettent d'organiser les classes en unité (de la même façon qu'on a des dossiers ou des répertoires) ;
- ils réduisent les conflits de noms ;
- ils permettent de protéger des classes, des méthodes ou des variables ;
- ils peuvent servir à identifier les classes. Il est possible de nommer un package avec un identificateur unique propre à une organisation donnée.

la commande `import`

Pour utiliser un package il faut faire appel à la commande **`import`**.
Il est possible d'importer une classe individuelle, par exemple : **`import java.util.Vector`** ; ou tout un package de classe, par exemple : **`import java.awt.* ;`**
Noter que le caractère « * » est différent de celui utilisé à la suite d'un prompt pour spécifier le contenu d'un répertoire. Par exemple, si l'on demande d'afficher le contenu du répertoire classes/java/awt/*, cette liste inclut les fichiers **`.class`** et tous les sous répertoires tels que **`peer`** et **`image`**. La commande **`import java.awt.*`** ; importe toutes les classes publiques de ce package mais pas les sous packages **`peer`** et **`image.`** L'import ne marche que « sur un niveau ».

Conflits de nom

Lorsqu'il y a un conflit de noms entre deux classes de packages différents, il suffit de préfixer le nom de la classe par le nom du package pour lever l'ambiguïté. Exemple : deux packages ayant chacun une classe « Name ».

```
import MonProjet.* ;
import SonProjet.* ;


Name myName = new Name ("dupont") ;
MonProjet.Name myName = new MonProjet.Name ("dupond") ;
```

Créer ses propres packages

Pour créer ses propres packages, il suffit de définir au début de chaque classe une ligne du type :

```
package MonProjet ;
```

Cette ligne signifie que toutes les classes appartenant au package `MonProjet` se trouvent dans un répertoire `MonProjet`. Il est donc important de préciser dans la variable d'environnement **CLASSPATH** l'ensemble des chemins des différents packages à importer.

3.12.- Classes abstraites et interfaces

3.12.1.- Les classes abstraites

Les classes abstraites sont des classes dont le seul but est de fournir des informations communes aux sous-classes. Les classes abstraites n'ont pas d'instance, mais peuvent contenir tout ce qu'une classe normale peut contenir (variables et méthodes de classe ou d'instance …). En plus elles peuvent contenir des méthodes abstraites. Ces méthodes abstraites sont-elles aussi un moyen de mettre à part, dans des super-classes, des comportements communs dont l'utilisation concrète est définie dans les sous-classes. Les méthodes abstraites n'ont pas d'implémentation, elles sont définies par leur **signature** (nom de la méthode, noms et types des paramètres éventuels).

On déclare les classes et les méthodes abstraites au moyen du modificateur **abstract**.

```
public abstract class MyAbstractClass {
   int a ;

   public abstract int subClass (double x) ;

   public void ma_methode () {
      ...
   }
}
```

Déclaration de la **signature** de la méthode abstraite

```
public class MyConcretClass extends MyAbstractClass {

   int a ;

   public int subClass (double x) {

   }
}
```

Implémentation de la méthode abstraite dans une classe dérivée non abstraite

3.12.2.- Les interfaces

Le mécanisme de l'interface est une généralisation du concept de classe abstraite. Plus précisément, une interface est une classe dont **toutes** les méthodes sont abstraites. On n'a donc plus besoin de spécifier que les méthodes sont abstraites car elle doivent forcément l'être.

Au niveau de la syntaxe, on introduit une interface non plus par le mot clé **class** mais par le mot **interface**. Petite subtilité au passage : bien que les interfaces bénéficient aussi du mécanisme de l'héritage, on n'hérite pas d'une interface, mais on **l'implémente**. En d'autres termes, on doit forcément fournir le code de toutes les méthodes de l'interface utilisée (sauf dans le cas d'une classe abstraite qui implémente une interface, ou bien d'une interface dérivée d'une autre). On utilise le mot-clé **implements** (et non **extends**) pour signifier que l'on implémente une interface.

La différence essentielle entre une classe abstraite dont toutes les méthodes seraient abstraites et une interface réside dans le fait que l'on ne peut hériter que d'une seule classe (héritage simple), alors que l'on peut implémenter plusieurs interfaces. **C'est une solution pour simuler l'héritage multiple**.

Voici quelques exemples.

```
public interface I1 {
  void m();
}

public abstract class C1 {
  void g();
}

class C2 extends C1 implements I1 {
  void m(){
    // Le code de m
  }

  void g() {
    // Le code de g
  }
}

public interface I2 extends I1 {
  void n();
}

abstract class C3 implements I2 {
  void n() {
    // Le code de n();
  }
}
```

```
class MyClass extends C2 implements I1, I2 {
   ...
}
```

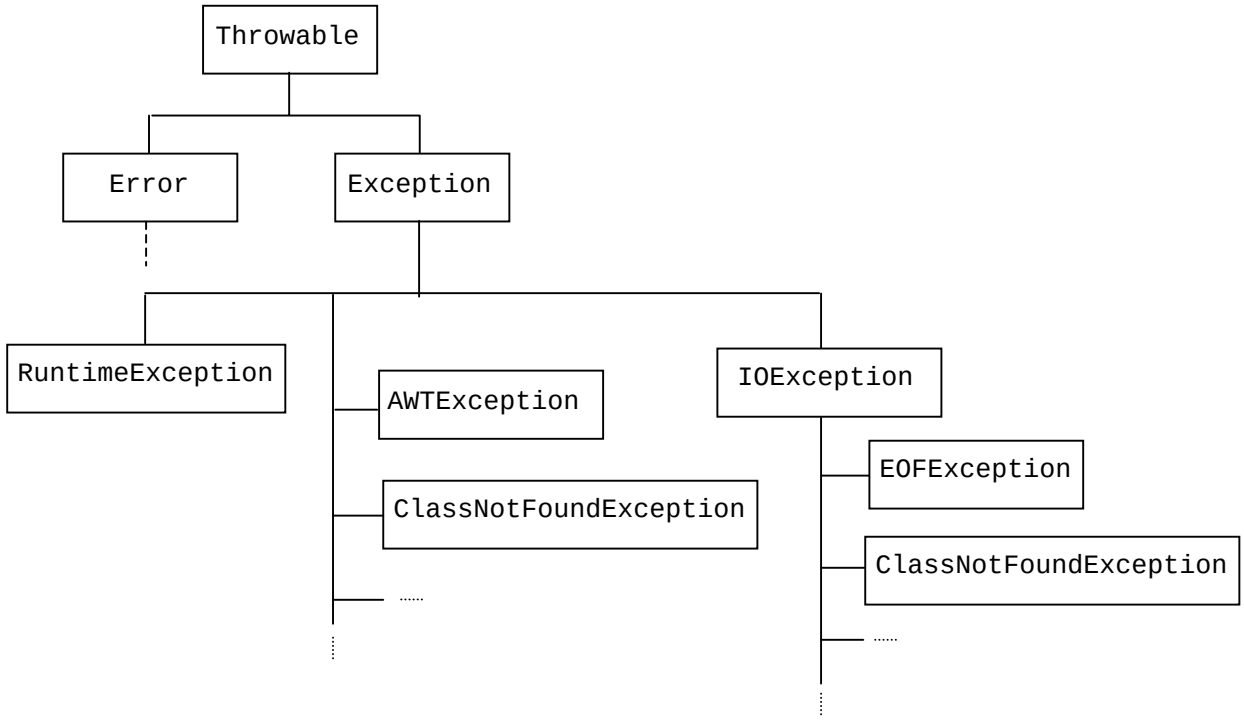
On peut implémenter plusieurs interfaces mais on hérite d'une seule classe.

5.13.- Les exceptions

La notion **d'exception** est offerte aux programmeurs Java pour résoudre de manière efficace et simple le problème de la gestion des erreurs émises lors de l'exécution d'un programme. Contrairement au C/C++, les exceptions et leur traitement font partie intégrante du langage.
L'idée fondamentale est qu'une méthode qui rencontre un problème impossible à traiter immédiatement **lève** (instructions **throw**) une exception en espérant que le programme appelant pourra la traiter (instructions **try, catch, finally**). Une méthode qui désire gérer ce genre de problèmes peut indiquer qu'elle est disposée à **intercepter** l'exception (instruction **throws**). Une fois acquis le principe de cette forme de traitement d'erreur, vous pourrez utiliser les classes d'exceptions Java prédéfinies ou créer vos propres classes pour traiter les erreurs qui peuvent survenir dans vos méthodes.

3.13.1.- Les exceptions prédéfinies

En Java, les exceptions sont de véritables objets. Ce sont des instances de classes qui héritent de la classe **Throwable**. Lorsqu'une exception est levée, une instance de la classe **Throwable** est créée. Ci-dessous un aperçu de la hiérarchie des classes pour les exceptions.

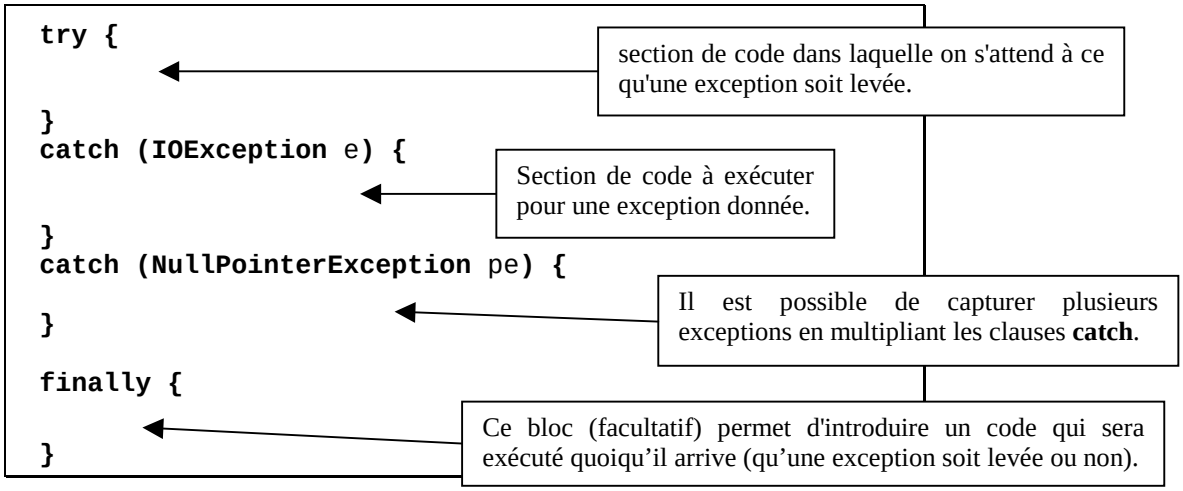


Les instances de la classe **Error** sont des erreurs internes à la machine virtuelle Java. Elles sont rares et fatales.

Les sous-classes de la classe **Exception** sont réparties en deux catégories :

- les exceptions d'exécution (runtime) sont souvent l'effet du manque de robustesse du code. Par exemple l'exception **NullPointerException** est levée lorsque l'on manipule un objet non instancié (oubli de l'instruction **new**) ;
- les autres exceptions correspondent à des événements anormaux échappant au contrôle du programme. Par exemple, l'exception **EOFException** est levée si on essaie de lire au-delà d'un fichier.

Traiter les exceptions levées : les mots clés **`try, catch et finally`**



Intercepter une exception : le mot clé **`throws`**

Si une méthode est susceptible de lever une exception et si elle ne peut pas la traiter, elle se doit de prévenir le système qu'elle relaye cette tâche. Pour ce faire, on utilise le mot clé **throws** dans la définition de la méthode**.** Ce mot clé permet d'avertir le système qu'une certaine catégorie d'exception ne sera pas traitée en local (dans l'exemple suivant, l'ensemble des exceptions liées aux entrées/sorties).

```
public void ma_methode (int x) throws IOEXception {
  ...
}
```

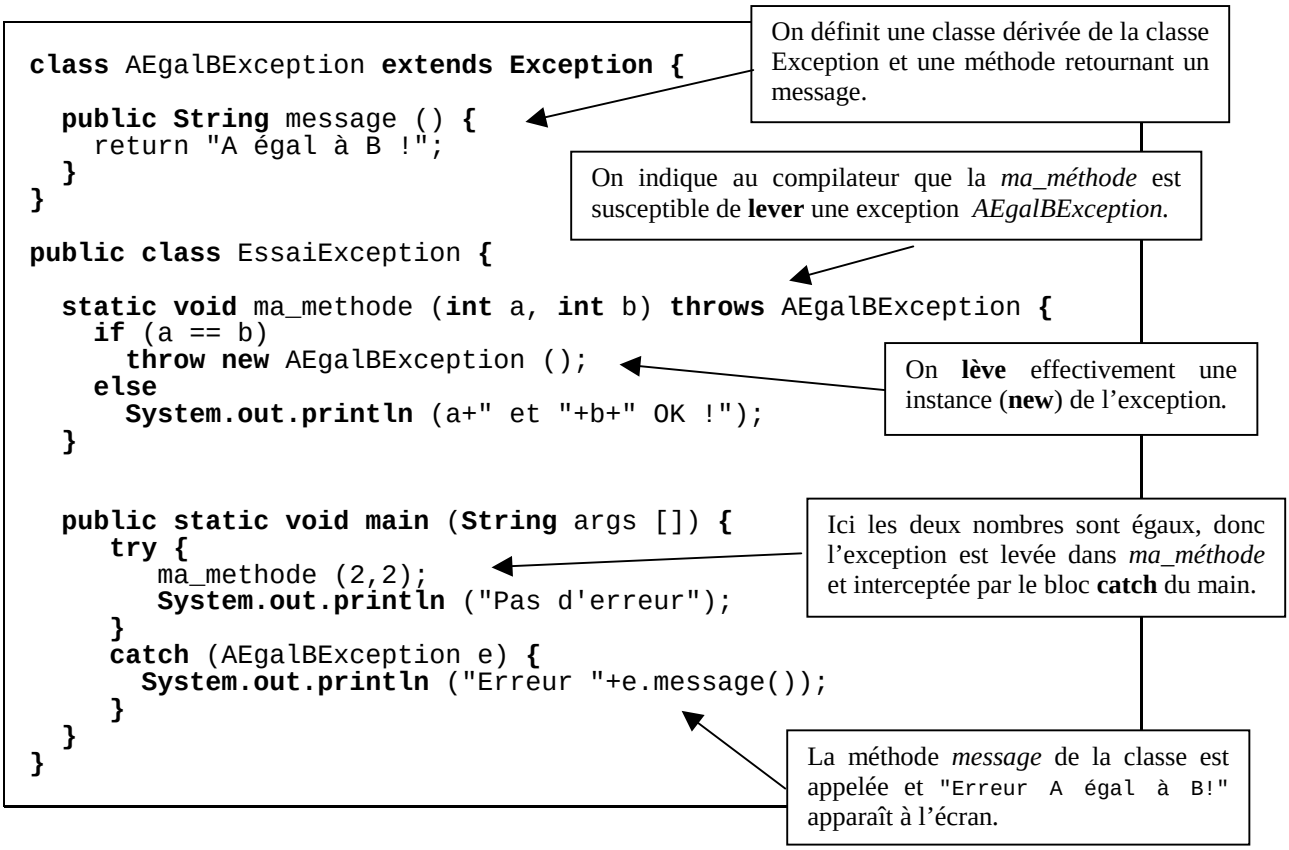
Il est également possible de signifier l'interception de plusieurs exceptions :

```
public void ma_methode (int x) throws IOException, EOFException {
  ...
}
```

3.13.2.- Les exceptions définies par le programmeur

Jusqu'à présent nous avons parlé des exceptions prédéfinies et nous ne nous sommes pas posé la question « comment lever une exception ? » puisque les exceptions prédéfinies se déclenchent toutes seules. Java offre au programmeur la possibilité de définir ses propres exceptions. Ces exceptions doivent hériter d'une autre exception de la hiérarchie des classes Java. Le programmeur doit lui-même lever ses exceptions. Pour se faire Java met à sa disposition le mot-clé **`throw`** (à ne pas confondre avec **`throws`**). Pour le reste (**`try, catch, finally`**) le mécanisme est identique.

Ci dessous, un exemple complet de création et d'utilisation d'une classe d'exceptions définie par le programmeur.



Nous aurons l'occasion de découvrir d'autres exemples d'utilisation des exceptions dans le chapitre suivant consacré aux fichiers.

3.14.- Les fichiers

Comme en C++, la gestion des fichiers dépend de la notion de flux (chemin de communication entre la source d'une information et sa destination).A la base de toutes les opérations d'entrée de flux se trouvent deux classes abstraites **`InputStream`** et **`Reader`** et à la base de toutes les opérations de sortie de flux les classes abstraites **`OutputStream`** et **`Writer`**. Il existe plusieurs classes dérivées permettant de lire et d'écrire dans des fichiers (texte ou binaire). Nous avons choisi l'une d'entre-elles (**`RandomAccessFile`**), permettant de manipuler les fichiers « texte » et les fichiers « binaire » de manière identique. La méthode la plus structurée consiste à créer ses propres classes de manipulation de fichiers en utilisant le mécanisme de l'héritage. On peut ainsi définir des classes dérivant de la classe **`RandomAccessFile`** et écrire ses propres méthodes de lecture et d'écriture pour chaque type de fichier à traiter.

Ci-dessous une exemple complet illustrant ces propos. Nous avons défini quatre classes :

- une classe *CoordText* contenant la description d'un enregistrement de type coordonnées (x, y, z sont codés en chaînes de caractères…) et la définition de méthodes permettant de lire et d'écrire un tel enregistrement dans un fichier texte ;
- une classe *ficTexte* montrant l'utilisation des méthodes de lecture et d'écriture. Il s'agit d'un programme qui lit des enregistrements de type *CoordText* dans un fichier et qui écrit ces enregistrements lus dans un autre fichier (avec un format différent) ;
- une classe *CoordBin* contenant la description d'un enregistrement de type coordonnées (x, y, z sont codés en binaire (double)…) et la définition de méthodes permettant de lire et d'écrire un tel enregistrement dans un fichier binaire ;
- une classe *ficBinaire* montrant l'utilisation des méthodes de lecture et d'écriture. Il s'agit d'un programme qui écrit des enregistrements de type *CoordBin* dans un fichier, qui les relit dans le même fichier avant d'afficher les valeurs à l'écran.

Fichier CoordText.java

```
import java.io.*;

public class CoordText  {

   String xt,yt,zt,pres_z;


   final int TailleEnr=19 ;

   public CoordText () {
     xt=new String();
     yt=new String();
     zt=new String();
     pres_z=new String("1");
   }

   public void setCoordText (String xt,String yt,String zt,String pres_z) {
     this.xt=xt;
     this.yt=yt;
     this.zt=zt;
     this.pres_z=pres_z;
   }
```

xt, yt et zt sont des coordonnées codés en chaînes de caractères. *pres_z* est un caractère ("0" ou "1") indiquant la présence ou non de la troisième dimension.

Taille d'un enregistrement (6+6+6+1 définie par des spécifications).

Constructeur. Il initialise les chaînes à "vide" et le booléen à vrai "1".

Affecte des valeurs aux champs d'un objet *CoordText.*

```
    public void readCoordText (RandomAccessFile ficText) {

        try {

          String line=ficText.readLine();

          xt=line.substring(0,5);
          yt=line.substring(6,11);
          zt=line.substring(12,17);
          pres_z=line.substring(18,19);

        }

        catch (IOException e) {
           System.out.println("Erreur a la lecture d'un objet CoordText");
        }
     }

     public void writeCoordText (RandomAccessFile ficText) {


        try {

           ficText.writeBytes(xt+";"+yt+";"+zt.trim()+";"+pres_z+"\n");
        }

        catch (IOException e) {
           System.out.println("Erreur a l'ecriture d'un objet CoordText");
        }
     }
}
```

Lit dans un fichier texte suivant un format établi, un objet de la classe *CoordText*.
Format => chaque ligne du fichier comporte 19 caractères : 6 pour l'abscisse, 6 pour l'ordonnée, 6 pour l'altitude et 1 pour le booléen.
Il n'y a aucun séparateur.

La fonction *readLine* lève un exception de type *IOException* en cas de problème à la lecture. Il est obligatoire de la traiter ici.

Ecrit dans un fichier texte, suivant un format établi, un objet de la classe *CoordText*.
Format => le même que ci-dessus mais avec un ";" entre chaque champ et la suppression des blancs (devant et derrière) sur la coordonnée z.

La fonction *writeBytes* lève un exception de type *IOException* en cas de problème à la lecture. Il est obligatoire de la traiter ici.

Fichier ficTexte.java

```
import java.awt.*;
import java.io.*;

// Ce programme est un exemple d'écriture/lecture d'objets de type texte
// dans un fichier.
public class ficTexte {

  public static void main (String args[]) {

    String nomfic = new String("fictext.txt");
    String nomfic2 = new String("fictext2.txt");

    CoordText coord = new CoordText ();

    try {
       // Ouverture d'un fichier texte de nom "nomfic" en mode lecture
       RandomAccessFile ficText = new RandomAccessFile(nomfic,"r");
       // Ouverture d'un fichier texte de nom "nomfic2" en mode ecriture
       RandomAccessFile ficText2 = new RandomAccessFile(nomfic2,"rw");

       //  calcul du nombre de lignes du fichier ficText
       int NbLigne=(int)ficText.length()/coord.TailleEnr;
```

On importe deux packages.

On crée un objet *CoordText* et deux objets *RandomAccessFile*.

```
      for (int i=1;i<=NbLigne;i++) {

         coord.readCoordText(ficText);

         //Affichage a l'ecran pour verification
         System.out.print(coord.xt+" ");
         System.out.print(coord.yt+" ");
         System.out.print(coord.zt+" ");
         System.out.println(coord.pres_z);

         coord.writeCoordText(ficText2);
      }

      // fermeture des fichiers
      ficText.close();
      ficText2.close();
   }

   catch (IOException ee) {
      System.out.print("Erreur a l'ouverture du fichier ");
      System.out.print(nomfic);
   }
 }
}
```

On lit *NbLigne* objets de type *CoordText* dans le fichier *fictext.txt*, puis on écrit les objets lus dans un autre fichier texte

Ecriture des mêmes objets dans un format différent défini d ans la classe *CoordText.*

Le constructeur *RandomAccessFile* lève une exception de type *IOException* s'il y a un problème à l'ouverture du fichier.
Il est obligatoire de traiter l'exception à ce niveau.

Fichier CoordBin.java

```
import java.io.*;

public class CoordBin  {

  double x,y,z;
  boolean pres_z;

  final int TailleEnr=25; //8+8+8+1


  public CoordBin() {
    x=0.0;
    y=0.0;
    z=0.0;
    pres_z=false;
  }

  public void setCoord (double x,double y,double z,boolean pres_z) {
    this.x=x;
    this.y=y;
    this.z=z;
    this.pres_z=pres_z;
  }

  public void writeCoordBin (RandomAccessFile ficBin) {
     try {
       ficBin.writeDouble(x);
       ficBin.writeDouble(y);
       ficBin.writeDouble(z);
       ficBin.writeBoolean(pres_z);
     }
     catch (IOException e) {
       System.out.println("Erreur a l'ecriture d'un objet de type Coord");
     }
   }
```

x, y et z sont des coordonnées codés en binaire (flottant double précision) *pres_z* est un booléen (*true* ou *false*) indiquant la présence ou non de la troisième dimension

Le constructeur initialise les champs de la classe

Cette méthode affecte les valeurs des champs.

Ecrit, dans un fichier binaire, un objet de type

```
  public void readCoordBin (RandomAccessFile ficBin) {
     try {
       x=ficBin.readDouble();
       y=ficBin.readDouble();
       z=ficBin.readDouble();
       pres_z=ficBin.readBoolean();
     }
     catch (IOException e) {
       System.out.println("Erreur a la lecture d'un objet de type Coord");
     }
  }
}
```

Lit dans un fichier binaire un objet de type *CoordBin* ayant été écrit par la méthode *writeCoordBin*

Fichier ficBinaire.java

```
import java.awt.*;
import java.io.*;

public class ficBinaire {

 // Exemple d'ecriture/lecture d'objets binaires dans un fichier.

 public static void main (String args[]) {
   String nomfic = new String("ficessai.out");
   CoordBin coord = new CoordBin ();

   try {

     RandomAccessFile ficBin = new RandomAccessFile(nomfic,"rw");

     coord.setCoord(2.2,3.4,4.5,true);
     coord.writeCoordBin(ficBin);
     coord.setCoord(12.34,3.56,0.0,false);
     coord.writeCoordBin(ficBin);
     coord.setCoord(123.4,23.7,1.1,true);
     coord.writeCoordBin(ficBin);

     ficBin.close();
   }
   catch (IOException ee) {
     System.out.print("Erreur a l'ouverture/creation du fichier ");
     System.out.print(nomfic);
   }

   try {

     RandomAccessFile ficBin = new RandomAccessFile(nomfic,"r");

     int NbEnr = (int)ficBin.length()/coord.TailleEnr;
     for (int i=1;i<=NbEnr;i++) {

        coord.readCoordBin(ficBin);

        System.out.print(coord.x+" "+coord.y+" "+coord.z+" ");
        System.out.println(coord.pres_z);
     }

     // fermeture du fichier
     ficBin.close();
   }
   catch (IOException e) {
     System.out.print("Erreur a l'ouverture du fichier ");
     System.out.print(nomfic);
   }
 }
}
```

Ouverture du fichier *ficessai.out* en mode lecture/écriture ("rw"). Si le fichier n'existe pas, il est crée.

Affectation et écriture dans le fichier de trois objets de la classe *CoordBin.*

Il est nécessaire de fermer le fichier ici, car on doit le rouvrir ensuite en lecture.

Ouverture du même fichier en lecture.

On lit les objets et on affiche le résultat de la lecture à l'écran.

3.15.- La sérialisation

3.15.1.- Introduction

Nous avons vu dans le chapitre précédent (les bases des flux en Java) comment écrire ou lire des caractères (du texte) à partir d'un fichier stocké sur un disque. Nous allons voir dans ce chapitre un principe de flux plus avancé : comment écrire et lire des *objets* dans un fichier, c'est la **sérialisation**. Les fichiers que vous *désérialisez* seront lus et chargés dans votre programme en conservant l'état dans lequel ils étaient quand ils ont été *sérialisé*s. Par exemple, imaginez que vous créez un objet de type **Integer** (pas int), que vous l'initialisez avec la valeur 0, que vous le modifiez ensuite pour avoir la valeur 1, et que, pour finir, vous le sérialisez. Quand vous désérialiserez cet objet, il aura la valeur 1. Nous allons voir comment créer des objets sérialisables, comment les sérialiser et les désérialiser, et comment empêcher que certaines variables de ces objets ne soient sérialisées.

3.15.2.- L'interface *Sérializable*

Pour qu'une instance d'une classe (un objet) soit *sérialisabl*e, la classe doit implémenter l'interface **`Serializable`**. Cette interface est un peu particulière puisqu'une classe qui l'implémente n'a pas besoin de redéfinir ses méthodes vides. Il suffit d'écrire :

```
class MaClasse implements Serializable
{
   // …
}
```

3.15.3. – La classe *ObjectOutputStream*

La classe **`ObjectOutputStream`** permet de sérialiser des objets et accessoirement des données issues des types de base (int, byte, char, double, float…). Son constructeur admet comme argument un **`FileOutputStream`.**

```
class MaClasse implements Serializable
{
   //…
}
```

```
class AutreClasse
{
    MaClasse mc = new MaClasse() ; // instance de la classe MaClasse
    public AutreClasse() { // constructeur de la classe autreClasse
       try {
          // créer un nouveau fichier
          FileOutputStream fos = new FileOutputStream("monFichier.dat");

          // ce fichier stockera des objets
          ObjectOutputStream oos = new ObjectOutputStream(fos);
          oos.writeObject(mc); // on écrit l'objet mc dans le flux oos
          oos.close() ; //fermer le flux

          // ObjectOutputStream susceptible de générer des exceptions
       } catch(IOException err) { }
    }
}
```

Vous pouvez trouver d'autres méthodes que **`writeObject()`** dans ObjectOutputStream :

```
write(int)
write(byte[])
write(byte[], int, int)
writeBoolean(boolean)
writeByte(int)
writeBytes(String)
writeChar(int)
writeChars(String)
writeDouble(double)
writeFloat(float)
writeInt(int)
writeLong(long)
writeShort(short)
```

3.15.4.- La classe *ObjectInputStream*

`ObjectInputStream` sert à récupérer des objets sérialisés. Elle fonctionne de façon similaire à **`ObjectOutputStream`**. Voici un exemple :

```
class EncoreAutreClasse {
   public EncoreAutreClasse() { // constructeur de la classe EncoreAutreClasse
     try {
       // lit le fichier
       FileInputStream fis = new FileInputStream("monFichier.dat");
       // ce fichier stocke des objets
       ObjectInputStream ois = new ObjectInputStream(fis);

       // lire un objet et le convertir en MaClasse
       MaClasse mc2 = (MaClasse)ois.readObject() ;

       ois.close() ; //fermer le flux

       // ObjectInputStream susceptible de générer des exceptions
     } catch(IOException err) { }
}
```

Tout comme **`ObjectOutputStrea`**m, **`ObjectInputStream`** possède plein de méthodes qui servent à lire des types de base :

```
o read() (lire des octets, retourne des int)
o read(byte[], int, int)
o readBoolean()
o readByte()
o readChar()
o readDouble()
o readFloat()
o readInt()
o readLine()
o readLong()
o readShort()
o readUnsignedByte() (retourne int, un vestige du C/C++…)
o readUnsignedShort() (de même)
```

Vous pouvez sérialiser plusieurs objets dans un même fichier en ayant recours plusieurs fois à la méthode **`writeObject()`** ou en utilisant les autres méthodes de **`ObjectOutputStream`**. Dans ce cas, vous devez appeler plusieurs fois les méthodes de **`ObjectInputStream`**. La règle est la suivante : le premier objet sérialisé est le premier fichier désérialisé, le second écrit est le second lu, etc….

3.15.5. - Mot réservé *transcient*

La sérialisation ne vous servira le plus souvent pas à écrire et lire des données appartenant aux types de base mais plutôt des objets, donc des instances de classes. Pour économiser de la place et du temps d'écriture, on peut rendre certaines variables de classe non-sérialisables à l'aide du mot réservé **`transcient`**. Par exemple, les variables **`manger`** et **`dormir`** ne seront pas sérialisées si on sérialise la classe **`Activites`** :

```
class Activites implements Serializable {

   //…

   public transcient int manger = 0 ;
   transcient int dormir = 1;

   private int travailler = 2 ;
   protected int programmer = 3;

   //…
}
```

4.- Java et les Interfaces Homme - Machine (IHM)

Dans la création d'IHM, on distingue deux grandes parties : **la partie statique** qui constitue l'aspect graphique de l'interface (menu, boutons, boîtes de dialogues …) et **la partie dynamique** qui définit et gère les actions à effectuer en fonction d'événements générés par l'utilisateur (ex : un clic souris sur un bouton provoque l'ouverture d'une boîte de dialogue).

Java met à la disposition du programmeur **le package `Swing`** (extension de l'ancien package **AWT** - Abstract Windowing Toolkit) qui offre tous les composants nécessaires à la création des IHM. Les principaux composants (menu, bouton …) sont regroupés dans des classes de ce package (classe *JMenu*, classe *JButton* …). La classe **`Event`** est utilisée pour la gestion des événements.

Les deux chapitres suivants n'ont pas pour but de dresser une liste exhaustive des composants de Swing et de leur utilisation, mais d'expliquer les mécanismes généraux permettant la création d'IHM. Pour connaître l'ensemble des composants (classes, méthodes, attributs) il faut consulter l'API Java (fichier HTML) livrée avec tout compilateur Java. Notons que les composants **Swing** ne sont accessibles que depuis la version 1.2 (Java 2). La plupart des composants de Swing existe dans le package AWT mais ce dernier présente une contrainte : certaines méthodes d'affichage des composants font directement appel au système d'exploitation, ce qui ne permet pas d'assurer 100% de portabilité au niveau de l'aspect graphique (un bouton n'aura pas tout à fait le même aspect sous Unix et sous Windows).

C'est pour pallier ce petit défaut que **le package `Swing`** a été créé. Pour assurer la portabilité 100%, un bouton (ou tout autre composant graphique) est dessiné non plus par les fonctions du système d'exploitation, mais par Java lui-même (ce qui a un prix en terme de temps d'exécution).

Les mécanismes de gestion d'évènements et de positionnement des composants de Swing sont le mêmes que ceux de l'AWT.

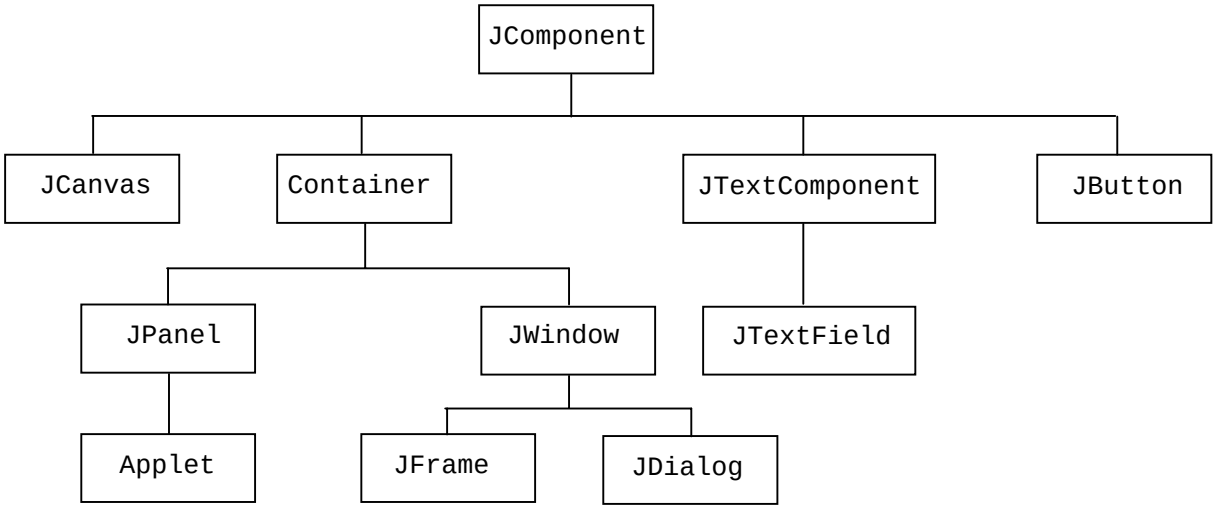
Le package Swing est plus riche que l'AWT (plus de composants, plus de méthodes …) ce qui incite désormais le programmeur à utiliser ce nouveau package. L'utilisation de l'un ou de l'autre des packages dépend de ce que l'on veut privilégier (la vitesse d'exécution ou la richesse et la portabilité de l'interface). Le nom des classes est quasiment identique. Le nom de la plupart des classes Swing est précédé d'un « J ». Exemple : la classe `Button` d'AWT est nommée `JButton` dans Swing.

4.1.- Les composants du package Swing

Il existe quatre grandes catégories de composants :

- les conteneurs (*Containers*). Ce sont des composants Swing généraux pouvant contenir d'autres composants (qui peuvent d'ailleurs être d'autres conteneurs). La forme la plus courante est le panneau (*JPanel*) qui peut être affiché à l'écran ;
- les canevas (*JCanvas*). Il s'agit d'une simple surface de dessin où l'on effectue toutes les opérations graphiques (image, dessin vecteur) ;
- les composants d'interface. Ils peuvent inclure des boutons, des listes, des menus popup, des cases à cocher, des champs de test …
- les composants pour la construction de fenêtres. Il s'agit des fenêtres elles-mêmes, des cadres, des barres de menu et des boîtes de dialogue.

Ci-dessous une vue partielle de la hiérarchie Swing.



La disposition des objets dans une interface dépend de :

- **l'ordre dans lequel ils ont été ajoutés dans le conteneur ;**
- **la politique de placement de ce conteneur.**

4.1.1.- Ajout d'un composant dans un conteneur : la méthode add()

Le principe de création d'une interface utilisateur est simple, il consiste à définir la hiérarchie des composants de l'interface et à ajouter (selon une politique de placement) ces composants dans les différents conteneurs définis.

Il y a plusieurs sortes de conteneurs : *JWindow* (rectangle sans bordure, sans titre … rarement utilisé) , *JFrame* (fenêtre cadre avec barre de titre, ascenseurs, poignées …) et *JDialog* (ressemble au *JFrame* – utilisé pour les dialogues de saisie). Dans les exemples qui suivent nous avons choisi le plus fréquemment utilisé : *JFrame*

L'exemple ci-dessous explique la création d'un bouton. Cet exemple est très simple mais il illustre parfaitement le mécanisme.

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

public class ButtonTest extends JFrame {
 JButton MonBouton;
 public ButtonTest (String titre) {
   setTitle (titre);
   Container contentPane = getContentPane();
   contentPane.setLayout(new FlowLayout());
   MonBouton = new JButton ("Mon bouton") ;
   ContentPane.add(MonBouton);
   setSize (200,100);
   setVisible(true);
 }

 public static void main (String args []) {
   new ButtonTest ("Mon Bouton") ;
 }
}
```

Import du package contenant tous les composants Swing

Notre classe hérite de la classe *JFrame* (conteneur)

Création d'un objet bouton et ajout de ce bouton dans le conteneur (méthode **`add()`**).

La différence majeure avec AWT est l'utilisation de **`getContentPane()`** pour configurer le LayoutManager et ajouter les composants.

La méthode **`setVisible(…)`** est indispensable si l'on veut un affichage à l'écran

Le programme principal se contente de créer un objet *ButtonTest*

Résultat de ce programme à l'écran :

4.1.2.- La politique de placement dans un conteneur

Il existe quatre principales politiques : **`FlowLayout`**, **`GridLayout`**, **`BorderLayout`** et « par coordonnées » :

- Le placement **FlowLayout** : les composants sont ajoutés les uns à la suite des autres, ligne par ligne. Si un composant ne peut être mis sur une ligne, il est mis sur la suivante. Par défaut les composants sont centrés.

```
...
 setLayout (new FlowLayout()) ;
 contentPane.add (new Button ("un")) ;
 contentPane.add (new Button ("deux")) ;
 contentPane.add (new Button ("trois")) ;
 contentPane.add (new Button ("quatre")) ;
 contentPane.add (new Button ("cinq")) ;
 setSize (400,100);
 setVisible(true);
 ...
```

Résultat de ce programme à l'écran :

- Le placement **GridLayout** : les composants sont rangés en ligne et en colonnes et ils ont la même largeur et la même hauteur. Ils sont placés de gauche à droite puis de haut en bas.

```
...
 setLayout (new GridLayout(2,3)) ;
 contentPane.add (new Button ("un")) ;
 contentPane.add (new Button ("deux"))
 contentPane.add (new Button ("trois")) ;
 contentPane.add (new Button ("quatre")) ;
 contentPane.add (new Button ("cinq")) ;
 setSize (400,100);
 setVisible(true);
 ...
```

Résultat de ce programme à l'écran :

- Le placement **BorderLayout** : permet de placer les composants sur les bords et au centre. On indique la politique de positionnement, puis chaque fois qu'on ajoute un composant on indique où le placer.

```
...
 setLayout (new BorderLayout()) ;
contentPane.add ("North",new Button ("un")) ;
contentPane.add ("East",new Button ("deux"))
contentPane.add ("Center",new Button ("trois")) ;
contentPane.add ("West",new Button ("quatre")) ;
contentPane.add ("South",new Button ("cinq")) ;
setSize (400,100);
setVisible(true);
...
```

Résultat de ce programme à l'écran :

- Le placement **« par coordonnées »** : permet de placer les composants indépendamment de la surface d'affichage, ce qui présente un avantage important pour les applets (Cf. chapitre 6) qui doivent s'afficher à l'intérieur d'une fenêtre de navigation. Le problème principal de cette « politique » de placement tient au fait que les composants sont définis avec des coordonnées absolues. Lorsqu'on « retaille » la fenêtre les dimensions des composants ne sont pas recalculées.

4.1.3.- Conclusion

Créer la partie statique d'une interface utilisateur est relativement simple en Java. On dispose d'un nombre important de composants que l'on peut disposer à sa guise grâce aux différentes politiques de placement prises en compte par le langage. Chaque composant possède des attributs et des méthodes permettant de modifier leur aspect et leurs comportements.

Bien que les mécanismes mis en œuvre soient simples, le programmeur doit garder à l'esprit que la réalisation d'une interface utilisateur « professionnelle » (plusieurs niveaux de conteneurs, des menus, des boutons, des boîtes dialogues …) est une tâche difficile et longue.

La partie statique de votre interface étant terminée, il reste à l'animer …

4.2. La gestion des événements

Un événement est une façon de prévenir le programmeur et les autres composants AWT que quelque chose s'est produit (des mouvements ou des clics souris, appuis sur des touches du clavier), des changements dans l'environnement … La détection de ces événements dans un programme Java lui permet de réagir aux actions de l'utilisateur en adaptant son comportement à ces actions. Les événements Java font partie du package AWT.

Il existe deux modèles de gestion d'événements. Dans ce document, nous ne parlerons que du modèle implémenté depuis la version 1.1 (le modèle implémenté dans la version 1.0 n'étant pratiquement plus utilisé).

L'écriture du code relatif aux événements comporte trois étapes fondamentales :

- Détermination des événements de l'application et attribution de ces événements à des écouteurs ;
- Ecriture du code de chaque écouteur et du code de traitement de chaque événement ;
- Enregistrement des écouteurs pour l'application.

4.2.1.- La notion d'écouteur d'événements

Un écouteur est un objet qui possède au moins une méthode qui pourra être invoquée si l'événement attendu apparaît. Une classe « écouteur » (*Listener*) implémente une interface. Il existe plusieurs types d'écouteurs, donc plusieurs types d'interfaces. Toutes ces interfaces ont un nom se terminant par *Listener* (par exemple **ActionListener**, **WindowListener**, ...). Selon l'interface, il y a une ou plusieurs méthodes à implémenter.

4.2.2.- Ecriture du code des écouteurs

A titre d'exemple, considérons l'évènement **ActionEvent** : cet évènement est déclenché lorsqu'on actionne un bouton (clic souris ou clavier). Pour être capable d'écouter et de traiter un tel évènement, un objet écouteur se doit d'implémenter une interface nommée **ActionListener** qui ne contient qu'une méthode à implémenter : **actionPerformed**. Cette méthode doit donc contenir le code à exécuter à chaque clic souris sur le bouton.

Ci-dessous un exemple de classe d'écouteur d événements : `MyListener.`

```
import java.awt.event.*;
public class MyListener extends EventFrame implements ActionListener {
    EventFrame f;

    MyListener (EventFrame frame) {
       f = frame;
    }
   public void actionPerformed(ActionEvent event) {
      if (event.getSource() == f.BoutonOk) {
          f.statusLabel.setText("OK") ;
      }
    if (event.getSource() == f.BoutonCancel) {
          f.statusLabel.setText("Cancel") ;
      }
   }
}
```

Il s'agit bien des boutons de l'objet **f,** copie locale de l'objet **frame** passé en paramètre du constructeur.

4.2.3.- Enregistrement des écouteurs

Le code ci-dessus correspond à la définition d'un écouteur d'événements (objet de classe *MyListener*) susceptible de répondre à un click sur le bouton mais on a pas « dit au bouton » qu'il fallait avertir l'écouteur en cas de clic souris. Chaque écouteur doit être enregistré auprès des objets qu'il est censé écouter (un écouteur peut écouter plusieurs sources d'évènements - une source d'évènement peut alerter plusieurs écouteurs). Prenons l'exemple complet en y ajoutant une ligne qui permet d'enregistrer l'écouteur d'événement aux boutons en question et l'ensemble du code de la classe représentant la frame.

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

public class EventFrame extends JFrame {

   Jbutton BoutonOk, BoutonCancel ;
   Jlabel statusLabel ;

   public EventFrame(String titre) {
      setTitle(titre);
      Container contentPane = getContentPane();
      JPanel ButtonPanel = new JPanel();
      ButtonPanel.setLayout(new FlowLayout());
      BoutonOk = new JButton ("OK");
      ButtonPanel.add(BoutonOk);
      BoutonCancel = new JButton ("Cancel");
      MyListener listener = new MyListener(this);
      BoutonOk.addActionListener(listener);
      BoutonCancel.addActionListener(listener);
      ButtonPanel.add(BoutonCancel);
      contentPane.add(ButtonPanel,BorderLayout.CENTER);
      statusLabel = new JLabel("Zone de texte");
      contentPane.add(statusLabel,BorderLayout.SOUTH);
      setSize(200,100);
      setVisible(true);
   }
}
```

On passe à l'écouteur un objet de type EventFrame

On ajoute un écouteur (un objet de la classe *MyListener*) au boutonOk et au boutonCancel .A chaque fois que l'un de ces boutons est actionné, la méthode *actionPerformed* est appelée sur chacun de ses écouteurs.

Reste à écrire la troisième classe `MonTest` contenant la fonction principale : `main`.

```
public class MonTest {
    public static void main(String[] args) {
       EventFrame ef = new EventFrame("Test evenements");
    }
}
```

On obtient :

4.2.4.- Les événements « classiques »

Composant	Evénement créé	Méthode invoquée
JButton	ActionEvent	actionPerformed
JCheckBox	ItemEvent	itemStateChanged
JList	ListSelectionEvent	valueChanged
JTextField	ActionEvent	actionPerformed
JWindow	WindowEvent	WindowOpen/Closed/Closing/ Minimized/Maximized …
JComponent	ComponentEvent	componentMoved/Resized/ Hidden/Show …
JComponent	FocusEvent	focusLost, focusGained
JComponent	KeyEvent	keyPressed/Released/Typed
JComponent	MouseEvent	mouseClicked/Pressed/Releazed/ Moved/Dragged/Entered/Exited

4.3.- Les menus et les boites de dialogue

4.3.1.- Créer un menu

La création d'un menu peut se décomposer en six étapes :

- créer une barre de menu (`JMenuBar`) ;
- créer les menus (`JMenu`) à ajouter à la barre de menu ;
- créer les sous-menus (`JMenuItem`) à ajouter au menu ;
- ajouter les sous-menus à chaque menu ;
- ajouter les menus à la barre de menu ;
- associer la barre de menu à une fenêtre cadre (frame).

Pour les menus dynamiques (pop-up), le processus de création est le même, il faut simplement ajouter les éléments de menu à un menu dynamique (`JPopupMenu`) et non à un barre de menu. Il faut également inscrire le listener (qui hérite de `MouseAdapter`) pour les événements de « relâché de souris » (`mouseReleased`) et afficher le menu dynamique à la position de la souris.

4.3.2.- Créer une boite de dialogue

L'utilisation d'une boite de dialogue (`JDialog`) est similaire à celle d'une `JFrame`. On l'utilise comme si on surchargeait une `JFrame`. Les boites de dialogue peuvent être « modale » (l'utilisateur doit fermer la boite de dialogue avant de continuer) ou non « modale » (permet de travailler dans d'autres fenêtres).

Exemple de surcharge de JDialog :

```
public class TestModalDialog extends JDialog {

    public TestModalDialog (String title, JFrame owner) {
       super (owner, title, true) ;
       …
    }
    public actionPerformed (ActionEvent e) {
       dispose() ;
    }
}
```

Appel du constructeur de `JDialog` en mode modal (**`true`**)

Rend la main à la frame appelante

Exemple de création de boite de dialogue :

La classe interne doit référencer la classe englobante

```
public class Appli extends JFrame {

    class EventHandler implements ActionListener {
       public void actionPerformed (ActionEvent e) {
         new TestModalDialog ("Test",ApplicationFrame.this);
    }
}
```

4.3.3.- Exemple de création de menus et d'une boite de dialogue

Ce chapitre propose le code source commenté d'une application comportant un menu barre, un menu pop-up et une boite de dialogue. L'application comporte sept classes :

- l'application (`appliMenu` qui contient la fonction `main`) ;
- une fenêtre cadre (`Frame1`) ;
- un menu barre (`MenuBarre`) ;
- un menu pop-up (`PopUp`) ;
- une boite de dialogue (`Dialogue`) ;
- un écouteur d'événements pour les actions sur le menu barre (`actionListener`) ;
- un écouteur d'événements pour l'affichage du menu pop-up (`popupAdapter`) ;

appliMenu.java

```
public class appliMenu {

    public static void main(String[] args) {
       Frame1 frame = new Frame1();
       frame.validate();
   }
}
```

Création d'un objet « fenêtre cadre »

Rafraîchissement de l'écran

Frame1.java

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

public class Frame1 extends JFrame {

   public Frame1() {
     setSize(new Dimension(400, 300));
     setTitle("Essai Menu");
     setVisible(true);

     MenuBarre mb= new MenuBarre(this);
     PopUp pop = new PopUp(this);
  }
}
```

On importe les composants des packages Swing et AWT

Constructeur de la frame

Caractéristiques de la fenêtre cadre

Création d'un objet Menu barre et d'un objet Pop-up menu associés à la fenêtre cadre (this)

MenuBarre.java

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

public class MenuBarre {

  private JMenuBar jMenuBar1 ;
  private JMenu jMenu1 ;
  private JMenu jMenu2 ;
  private JMenuItem jMenuItem1 ;
  private JMenuItem jMenuItem2 ;
  private JMenuItem jMenuItem3 ;
  private JMenuItem jMenuItem4 ;
  private JMenuItem jMenuItem5 ;

  public MenuBarre(JFrame f) {

       jMenuBar1 = new JMenuBar();
       jMenu1 = new JMenu("Fichier");
       jMenu2 = new JMenu("Edition");
       jMenuItem1 = new JMenuItem("Nouveau");
       jMenuItem2 = new JMenuItem("Ouvrir");
       jMenuItem5 = new JMenuItem("Quitter");
       jMenuItem3 = new JMenuItem("Couper");
       jMenuItem4 = new JMenuItem("Coller");

       jMenuBar1.add(jMenu1);
       jMenuBar1.add(jMenu2);
       jMenu1.add(jMenuItem1);
       jMenu1.add(jMenuItem2);
       jMenu1.add(jMenuItem5);
       jMenu2.add(jMenuItem3);
       jMenu2.add(jMenuItem4);
       f.setJMenuBar(jMenuBar1);

       actionListener aList = new actionListener(this);
       jMenuItem1.addActionListener(aList);
  }
}
```

Déclaration des composants du menu barre

Constructeur du menu associé à la frame

Construction des menus et sous-menus

Construction d'un objet écouteur d'événements

Ajout de l'écouteur au sous-menu « Nouveau »

PopUp.java

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

public class PopUp
{
    private JPopupMenu p ;
    private JMenuItem jMenuItem1 ;
    private JMenuItem jMenuItem2 ;
    private JMenuItem jMenuItem3;
    private JMenuItem jMenuItem4 ;

    public PopUp(JFrame f)
    {
      p = new JPopupMenu();

      jMenuItem1 = new JMenuItem("annuler");
      jMenuItem2 = new JMenuItem("copier");
      jMenuItem3 = new JMenuItem("couper");
      jMenuItem4 = new JMenuItem("coller");

      p.add(jMenuItem1);
      p.add(jMenuItem2);
      p.add(jMenuItem3);
      p.add(jMenuItem4);

      popupAdapter popup = new popupAdapter(p);
      f.addMouseListener(popup);
    }
}
```

Constructeur du menu associé à la frame

Construction des menus et sous-menus

Construction d'un objet écouteur d'événements

Ajout de l'écouteur à la frame associée

Dialogue.java

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

public class Dialogue extends JDialog
{
  private JButton jButton1 ;
  private JPanel jPanel1 ;
  private FlowLayout flowLayout1 ;

  public Dialogue() {
    jPanel1 = new JPanel();
    flowLayout1 = new FlowLayout();
    jPanel1.setLayout(flowLayout1);

    Container contentPane = getContentPane();
    contentPane.add(jPanel1);

    jButton1 = new JButton("jButton1");
    jPanel1.add(jButton1);
  }
}
```

actionListener.java

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

public class actionListener implements ActionListener {
  private MenuBarre mB ;

  public actionListener(MenuBarre menu) {
     mB=menu;
  }

  public void actionPerformed(ActionEvent e) {

      Dialogue dial = new Dialogue();
      dial.setLocation(150,150);
      dial.setSize(200,100);
      dial.setVisible(true);
  }
}
```

ActionListener est une interface

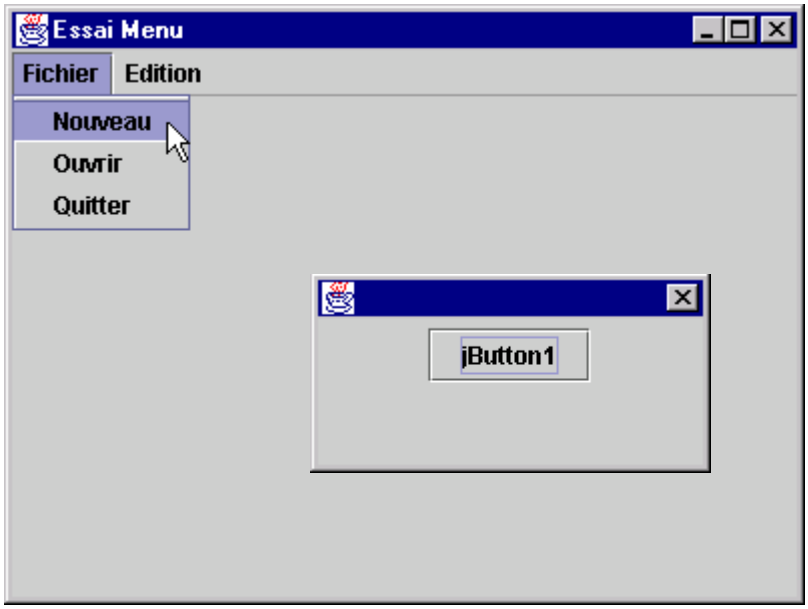
Construction et affichage d'une boite de dialogue

popupAdapter.java

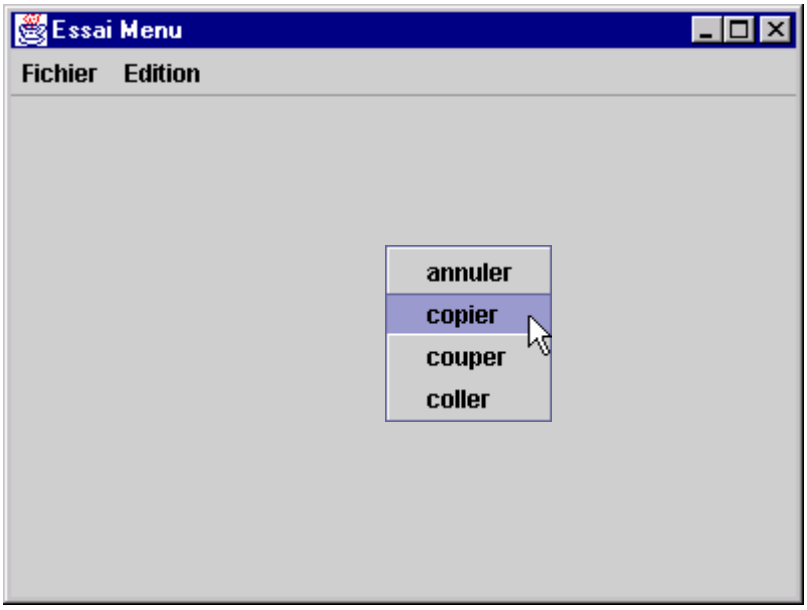
```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class popupAdapter extends MouseAdapter {
  private JPopupMenu p;

  popupAdapter(JPopupMenu popup) {
      p=popup;
  }

  public void mouseReleased (MouseEvent e) {
     if (e.isPopupTrigger()) {
         p.show((Component)e.getSource(),e.getX(),e.getY());
     }
  }
}
```

Affichage du menu pop-up quand on relâche le bouton droit de la souris.

Ouverture de la boite de dialogue après sélection de l'item « Nouveau » dans le menu « Fichier » :



Affichage du pop-up menu après un clic droit de souris :



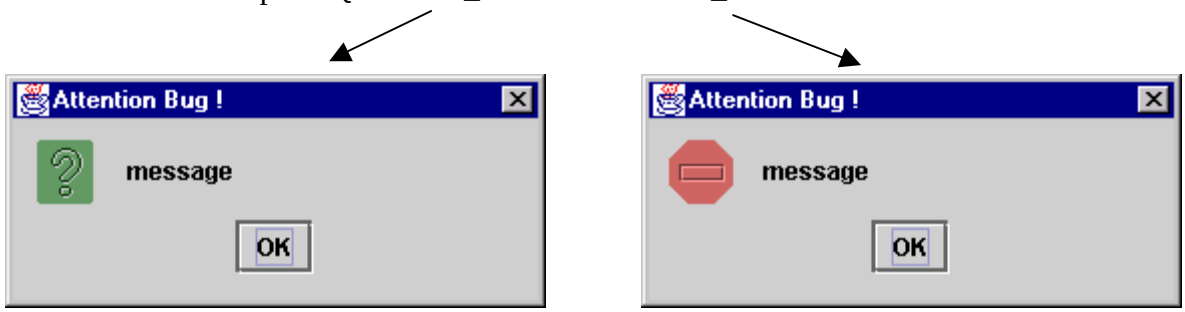
4.3.4.- Une boite de dialogue particulière : JOptionPane

Cette classe est utilisée pour les messages à l'écran, les avertissements … Son utilisation est très simple :

```
JOptionPane.showMessageDialog (null, "message ", "Attention bug !",
                               JOptionPane.ERROR_MESSAGE) ;
```

Le premier paramètre est la frame propriétaire (à activer quand la boite se referme – null signifie qu'aucune frame n'est active). Les deuxième et troisième sont le titre de la fenêtre et le message, le dernier paramètre est le type du message (parmi `ERROR_MESSAGE`, `INFORMATION_MESSAGE`, `WARNING_MESSAGE`, `QUESTION_MESSAGE`, `PLAIN_MESSAGE`).

Ci-dessous deux exemples : `QUESTION_MESSAGE` et `ERROR_MESSAGE`



4.4.- Les environnements de développement intégrés

Ce sont des logiciels d'aide à la programmation d'applications importantes et plus particulièrement d'aide à la création d'IHM. Ces logiciels utilisent tous les concepts que nous venons de voir dans les chapitres précédents. Il s'agit de soulager la tâche du programmeur en générant le squelette du code correspondant à une IHM. Grâce à ce genre d'outils, il est possible de réaliser la maquette d'une IHM en quelques clics de souris. On ajoute ensuite de manière très simple les comportements en fonction des actions que l'on veut effectuer et l'outil génère le code correspondant. Il reste au programmeur à remplir le corps des méthodes réalisant les actions. Le programmeur n'a pas besoin de connaître le fonctionnement exact du système de gestion des événements pour réaliser une interface utilisateur puisque toute cette partie est générée automatiquement.

Le produit de Borland, **Jbuilder**, est également gratuit (sauf la version professionnelle) et gourmand en mémoire (moins que Forte 1.0). Il s'inspire des générateurs permettant de programmer des IHM en C++ (C++Builder, VisualC++) et peut paraître plus familier à certains. La dernière version est la version 7.
Il est possible de le télécharger à l'adresse **http://www.borland.com/jbuilder/foundation/**
Autres sites très intéressants concernant JBuilder (version 4 et 5).
http://www.developpez.com/java et http://www.inprise.com/jbuilder/

Le produit de Sun, **Forte 1.0** est gratuit (logiciel et documentations) et très simple à utiliser. Il est malheureusement très gourmand en mémoire (compter minimum 128 MO, 256MO pour être à l'aise). Il est possible de le télécharger à l'adresse **http://www.sun.com/forte/.**

Il existe également un produit Microsoft (payant !) équivalent au VisualC++ : **VisualJ++.**
Vous trouverez des renseignements sur ce produit à l'adresse **: http://www.microsoft.com/catalog/**

Deux autres produits intéressants :

VisualAge for Java d'IBM : **http://www-3.ibm.com/software/ad/vajava/**

Visual Café de Symantec : **http://www.webgain.com/products/visual_cafe/**

Principales qualités des environnements intégrés :

- environnement graphique ergonomique et « intuitif » ;
- gestion automatisée des projets ;
- conception interactive d'IHM ;
- compilateur rapide.

Principaux défauts des environnements intégrés :

- pas toujours 100% de compatibilité Java ;
- risques adjonctions de fonctionnalités non-standard ;
- très gourmands en ressources mémoire (256 Mo de RAM conseillé).

5.- Java : programmation avancée

5.1.- Les threads

5.1.1.- définition

L'environnement de la Machine Virtuelle Java est multi-threads. L'équivalent de « thread » pourrait être « tâche » en français, mais pour éviter la confusion avec la notion de système multitâches, il est préférable d'employer le mot « thread » plutôt que « tâche ». Le fait que Java permettent de faire tourner en parallèle plusieurs threads lui donne beaucoup d'intérêt. Ceci permet par exemple de lancer le chargement d'une image sur le Web (ce qui peut prendre du temps), sans bloquer le programme qui peut ainsi effectuer d'autres opérations.
Plusieurs aspects des threads sont à étudier pour bien comprendre leur fonctionnement et leur utilisation :

- la gestion par la Machine Virtuelle Java pour répartir le temps d'exécution entre les différents threads ;
- la manière de créer un thread ;
- les différents états possibles d'un thread ;
- la synchronisation entre threads pour le partage de données.

5.1.2.- La gestion des threads - mécanisme général

Comment faire tourner plusieurs threads en même temps alors que votre ordinateur ne possède qu'un seul microprocesseur ? La réponse vient de manière assez évidente : à tout moment il n'y a en fait qu'un seul thread en cours d'exécution et éventuellement d'autres threads en attente d'exécution.
Si le système permet de partager le temps d'exécution entre les différents threads, il leur donne chacun leur tour un petit temps d'exécution du processeur (quelques millisecondes). Sinon, chaque fois qu'un thread a terminé d'exécuter une série d'instructions et qu'il cède le contrôle au système, celui-ci exécute un des threads en attente et ainsi de suite ... Si la série d'instructions qu'exécute chaque thread prend un temps assez court, **l'utilisateur aura l'illusion que tous les threads fonctionnent ensemble.**
Le seul contrôle que peut avoir le programmeur sur la gestion de l'ordre dans lequel les threads en attente s'exécuteront, s'effectue en donnant une priorité à chaque thread qu'il crée. Quand le système doit déterminer quel thread en attente doit être exécuté, il choisit celui avec la priorité la plus grande, ou à priorité égale, celui en tête de file d'attente.

Sur certains systèmes, la Machine Virtuelle Java ne partage pas d'elle-même le temps du processeur entre les threads susceptibles d'être exécutés. Si vous voulez que vos programmes Java se comportent de façon similaire sur tous les systèmes, vous devez céder le contrôle régulièrement dans vos threads avec les méthodes telles que `sleep()` ou `yield()`, pour permettre aux autres threads en attente, de s'exécuter. Sinon, tant que l'exécution d'un thread n'est pas interrompue, les autres threads resteront en attente !

5.1.3.- La création d'un thread

Il existe deux moyens d'écrire des threads dans un programme. Les deux moyens passent par l'écriture d'une méthode `run()` décrivant les instructions que doit exécuter un thread. Cette méthode `run()` est soit déclarée dans une classe dérivée de la classe `Thread`, soit déclarée dans n'importe quelle classe qui doit alors implémenter l'interface `Runnable` (cette interface ne décrit que la méthode `run()`). La seconde méthode est très utile : elle permet d'ajouter les fonctionnalités des threads à une classe existante, dans une classe dérivée.

Un certain nombre de méthodes sont nécessaires pour contrôler l'exécution d'un thread, en voici les principales :

La classe `Thread` dispose principalement de deux sortes de constructeurs **: `Thread ()` et `Thread (Runnable objetRunnable)`**.Quand vous créer un objet instance d'une classe `ClasseThread` dérivée de `Thread`, le premier est appelé implicitement par le constructeur de `ClasseThread`. Le second est utilisé quand vous voulez créer un objet de classe `Thread` dont la méthode `run()` à exécuter se trouve implémentée dans un classe implémentant l'interface `Runnable`. Au passage, vous noterez que le paramètre du second constructeur doit être une référence d'interface `Runnable`. En fait, vous passerez en argument une référence sur un objet d'une classe `ClasseRunnable` implémentant `Runnable` ; ceci est un exemple de conversion d'une référence d'une classe `ClasseX` dans une référence d'interface `InterfaceY`, si `ClasseX` implémente l'interface `InterfaceY`, appliquée avec `ClasseRunnable` pour `ClasseX` et `Runnable` pour `InterfaceY`.

`start()`: Cette méthode permet de démarrer effectivement le thread sur lequel elle est invoquée, ce qui va provoquer l'appel de la méthode `run ()` du thread. Cet appel est obligatoire pour démarrer l'exécution d'un thread. En effet, la création d'un objet de classe `Thread` ou d'une classe dérivée de `Thread` (par exemple, grâce à l'appel `new Thread ()`) ne fait que créer un objet sans appeler la méthode `run ()`.

`stop()`: Cette méthode permet d'arrêter un thread en cours d'exécution. Elle est utile principalement pour stopper des threads dont la méthode `run()` ne se termine jamais. Néanmoins, cette méthode est « deprecated » (non recommandée) dans la dernière version du JDK. Motif invoqué par les concepteurs : « Deadlock-prone » (sujet au blocage du système). Les méthodes « deprecated » sont détaillées en annexe (9.2).

`sleep()`: Ces méthodes `static` permettent d'arrêter le thread courant pendant un certain laps de temps, pour permettre ainsi à d'autres threads en attente, de s'exécuter. Par exemple, une fois mis à jour une horloge par un thread, celui-ci peut arrêter son exécution pendant une minute, en attendant la prochaine mise à l'heure.

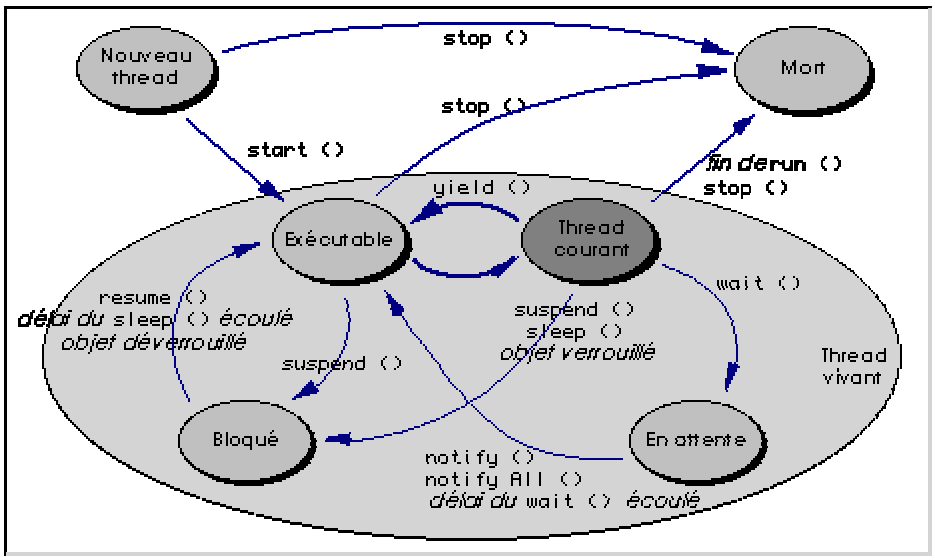
`yield()` : Cette méthode `static` permet au thread courant de céder le contrôle pour permettre à d'autres threads en attente, de s'exécuter. Le thread courant devient ainsi lui-même en attente, et regagne la file d'attente. De manière générale, vos threads devraient s'arranger à effectuer des séries d'instructions pas trop longues ou à entrecouper leur exécution grâce à des appels aux méthodes `sleep()` ou `yield()`.Il faut éviter de programmer des séries d'instructions interminables sans appel à `sleep()` ou `yield()`, en pensant qu'il n'y aura pas d'autres threads dans votre programme. La Machine Virtuelle Java peut avoir elle aussi des threads système en attente et votre programme s'enrichira peut-être un jour de threads supplémentaires...

`setPriority()` : Permet de donner une priorité à un thread. Le thread de plus grande priorité sera toujours exécuté avant tous ceux en attente.

Un exemple d'utilisation de ces méthodes est en annexe (cf. chapitre A3.1).

5.1.4.- Les états d'un thread

Pendant son existence, un thread passe par plusieurs états qu'il est utile de connaître pour bien comprendre les threads et leurs utilisations. La figure suivante représente l'ensemble des états possibles d'un thread et les transitions existantes pour passer d'un état à un autre. Ce modèle peut sembler complexe mais il faut garder à l'esprit que pour la plupart des utilisations, seules les méthodes `start()`, `stop()`(« `deprecated` »), `yield()` et `sleep ()` sont nécessaires. A noter que les méthodes `resume()` et `suspend()` sont également « deprecated ».



Quand un nouveau thread est créé, il est dans l'état ***nouveau thread*** et ne devient ***exécutable*** qu'après avoir appelé la méthode `start()` sur ce nouveau thread.

Parmi tous les threads dans l'état *exécutable*, le système donne le contrôle au thread de plus grande priorité, ou à priorité égale, celui en tête de file d'attente, parmi les threads dans l'état *exécutable*. Le thread qui a le contrôle à un moment donné est le ***thread courant***. Le *thread courant* en cours d'exécution cède le contrôle à un autre thread *exécutable* dans l'une des circonstances suivantes :

- A la fin de la méthode `run()`, le thread passe dans l'état *mort* ;
- A l'appel de `yield()`, le thread passe dans l'état *exécutable* et rejoint la fin de la file d'attente ;
- Sur les systèmes permettant de partager le temps d'exécution entre différents threads, le thread passe dans l'état *exécutable* après qu'un certain laps de temps se soit écoulé ;
- En attentant que des opérations d'entrée/sortie se terminent, le thread passe dans l'état *bloqué* ;
- A l'appel de `sleep()`, le thread passe dans l'état *bloqué* pendant le temps spécifié en argument, puis repasse à l'état *exécutable*, une fois ce délai écoulé ;
- A l'appel d'une méthode `synchronized` sur un objet `Objet1`, si `Objet1` est déjà verrouillé par un autre thread, alors le thread passe dans l'état *bloqué* tant que `Objet1` n'est pas déverrouillé (voir le chapitre suivant sur la synchronisation des threads) ;
- A l'invocation de `wait()` sur un objet, le thread passe dans l'état ***en attente*** pendant le délai spécifié en argument ou tant qu'un appel à `notify()` ou `notifyAll()` n'est pas survenu (cf. la synchronisation des trheads). Ces méthodes sont déclarées dans la classe Object ;
- A l'appel de `stop()`, le thread passe dans l'état *mort* ;
- A l'appel de `suspend()`, le thread passe dans l'état *bloqué* et ne redevient *exécutable* qu'une fois que la méthode `resume()` a été appelée.

Les deux derniers méthodes ne sont pas `static` et peuvent être invoquées aussi sur les threads exécutables qui ne sont en cours d'exécution.

5.1.5.- La synchronisation des threads

Tous les threads d'une même Machine Virtuelle partage le même espace mémoire et peuvent donc avoir accès à n'importe quels méthode ou champ d'objets existants. Ceci est très pratique, mais dans certains cas, vous pouvez avoir besoin d'éviter que deux threads n'aient accès n'importe quand à certaines données. Si par exemple, un thread `threadCalcul` a pour charge de modifier une champ `var1` qu'un autre thread `threadAffichage` a besoin de lire pour l'afficher, il semble logique que tant que `threadCalcul` n'a pas terminé la mise à jour ou le calcul de `var1`, `threadAffichage` soit interdit d'y avoir accès. **Vous aurez donc besoin de synchroniser vos threads**.

5.1.5.1.- L'utilisation de « synchronized »

La synchronisation des threads se fait grâce au mot clé **`synchronized`**, employé principalement comme « modifieur » d'une méthode. Soient une ou plusieurs méthodes `methodeI()` déclarées `synchronized`, dans une classe `Classe1` et un objet `objet1` de classe `Classe1`. Comme tout objet Java comporte un verrou (*lock* en anglais) permettant d'empêcher que deux threads différents n'aient un accès simultané à un même objet, quand l'une des méthodes `methodeI() synchronized` est invoquée sur `objet1`, deux cas se présentent :

- **L'objet** `objet1` **n'est pas verrouillé.** Dans ce cas, le système pose un verrou sur cet objet puis la méthode `methodeI()` est exécutée normalement. Quand `methodeI()` est terminée, le système retire le verrou sur `Objet1`.
  La méthode `methodeI ()` peut être récursive ou appeler d'autres méthodes `synchronized` de `Classe1`. A chaque appel d'une méthode `synchronized` de `Classe1`, le système rajoute un verrou sur `Objet1`, retiré en quittant la méthode. Quand un thread a obtenu accès à un objet verrouillé, le système l'autorise à avoir accès à cet objet tant que l'objet a encore des verrous ;

- **L'objet** `objet1` **est déjà verrouillé** . Si le *thread courant* n'est pas celui qui a verrouillé `objet1`, le système met le *thread courant* dans l'état *bloqué*, tant que `objet1` est verrouillé. Une fois que `objet1` est déverrouillé, le système remet ce thread dans l'état *exécutable*, pour qu'il puisse essayer de verrouiller `objet1` et exécuter `methodeI ()`.

Si une méthode `synchronized` d'une classe `Classe1` est également `static`, à l'appel de cette méthode, le même mécanisme s'exécute mais cette fois-ci en utilisant le verrou associé à la classe `Classe1`.

Si `Classe1` a d'autres méthodes qui ne sont pas `synchronized`, celles-ci peuvent toujours être appelées n'importe quand, que `objet1` soit verrouillé ou non.

5.1.5.2.- La synchronisation avec les méthodes `wait()` et `notify()`

Comme il est expliqué dans le chapitre précédent, `synchronized` permet d'éviter que plusieurs threads aient accès en même temps à même objet, mais ne garantit pas l'ordre dans lequel les threads ces méthodes seront exécutées. Pour cela, il existe plusieurs méthodes de la classe Object qui permettent de mettre *en attente* volontairement un thread sur un objet (méthode `wait()`) et de prévenir des threads en attente sur un objet que celui-ci est à jour (méthodes `notify()` ou `notifyAll()`). Ces méthodes ne peuvent être invoquées que sur un objet verrouillé par le *thread courant*, c'est-à-dire que le *thread courant* est en train d'exécuter une méthode ou un bloc `synchronized`, qui a verrouillé cet objet. Si ce n'est pas le cas, une exception `IllegalMonitorStateException` est déclenchée.

Quand `wait()` est invoquée sur un objet `objet1` (`objet1` peut être `this`), le *thread courant* perd le contrôle, est mis *en attente* et l'ensemble des verrous d'`objet1` est retiré. Comme chaque objet Java mémorise l'ensemble des threads mis *en attente* sur lui-même, le *thread courant* est ajouté à la liste des threads *en attente* de `objet1`. `objet1` étant déverrouillé, un des threads *bloqués* parmi ceux qui désiraient verrouiller `objet1` peut passer dans l'état *exécutable* et exécuter une méthode ou un bloc `synchronized` sur `objet1`.
Un thread `thread1` mis *en attente* est retiré de la liste d'attente de `objet1`, quand une des trois raisons suivantes survient :

- `thread1` a été mis *en attente* en donnant en argument à `wait()` un délai qui a fini de s'écouler ;
- Le *thread courant* a invoqué `notify()` sur `objet1`, et `thread1` a été choisi parmi tous les threads *en attente* ;
- Le *thread courant* a invoqué `notifyAll ()` sur `objet1`.

`thread1` est mis alors dans l'état *exécutable*, et essaye de verrouiller `objet1` pour continuer son exécution. Quand il devient le *thread courant*, l'ensemble des verrous qui avait été enlevé d'`objet1` à l'appel de `wait()`, est remis sur objet1, pour que `thread1` et `objet1` se retrouvent dans le même état qu'avant l'invocation de `wait()`.

5.1.6.- Conclusion

La programmation de la synchronisation des threads est une tâche ardue, sur laquelle vous passerez sûrement du temps ... Si vous désirez l'utiliser, il faut bien s'imaginer par quels états vont passer les threads, quelle implication aura l'utilisation des méthodes `wait()` et `notify()` sur l'ordre d'exécution des instructions du programme, tout en gardant bien à l'esprit que ces méthodes ne peuvent être invoquées que sur des objets verrouillés.

Le piège le plus classique est de se retrouver avec un « deadlock » parce que les threads sont tous *en attente* après avoir appelé chacun d'eux la méthode `wait()`.

5.2.- Les collections

Le package `java.util` rassemble des classes d'utilitaires dont les plus intéressantes permettent de gérer les collections de données (classes `Vector, Stack, Dictionary, Hashtable` et interface `Enumeration`).

5.2.1.- L'interface java.util.Enumeration

Cette interface est implémentée par les classes désirant pouvoir faire une énumération des objets mémorisés par une autre classe, comme par exemple la classe `Vector`. Les méthodes de cette interface sont généralement utilisées dans une boucle `while`.

5.2.2.- La classe java.util.Vector

Cette classe qui implémente l'interface `Cloneable`, permet de créer un vecteur. Ce type d'ensemble permet de mémoriser un ensemble d'objets de classe quelconque dans un tableau de taille variable (ces éléments peuvent être éventuellement égal à `null`). Comme pour les tableaux, l'accès aux éléments se fait par un indice. La classe `Vector` comporte de nombreuses méthodes qui permettent d'ajouter, d'insérer, de supprimer ou de rechercher des éléments. Toutes les méthodes de `Vector` sont `final` sauf `clone()`.

5.2.3.- La classe java.util.Stack

Cette classe qui dérive de `Vector` permet de créer des piles, où vous pouvez empiler un objet avec la méthode `push()`, retirer l'élément en haut de la pile avec `pop()` ou consulter sans le retirer l'élément en haut de la pile avec `peek()`.

5.2.4.- La classe java.util. Dictionnary

Cette classe `abstract` permet de créer un dictionnaire représenté par un ensemble d'*entrées* associant un élément et une clé. Chaque clé du dictionnaire est unique et est associé au plus à un élément, mais un même élément peut avoir plusieurs clés d'accès. Les éléments et les clés devant être de la classe `Object`, le cast de référence permet donc d'utiliser n'importe quel type de classe pour les clés et les éléments (chaînes de caractères, classes d'emballage des nombres ou d'autres classes). Un dictionnaire peut être comparé à un tableau. Dans un tableau `tab`, vous mémorisez un ensemble d'éléments accessible grâce à un indice entier `i`, par l'expression `tab [i]`. Il est possible de mémoriser plusieurs fois le même objet dans `tab` à des indices différents, mais par contre chaque indice `i` est unique et vous permet d'accéder aux différents éléments du tableau grâce à `tab [i]`. Dans un dictionnaire `dict`, vous mémorisez de la même manière des éléments auquel vous accédez grâce à une clé plutôt que par un indice entier. Cette clé peut être de n'importe quelle classe, ce qui permet de mémoriser les éléments d'une manière plus riche qu'avec un simple indice entier. Pour faire un parallèle entre l'utilisation d'un tableau `tab` et d'un dictionnaire `dict`, l'expression `tab[i] = element` a pour équivalent `dict.put (cle, element)` et l'expression `element = tab[i]` a pour équivalent `element = dict.get (cle)`. Voici toutes les méthodes que doit implémenter une classe dérivant de `Dictionary`, pour pouvoir être instanciée (comme la classe Hashable) :

5.2.5.- La classe java.util.Hashable

Cette classe implémente l'interface `Cloneable` et dérive de la classe `Dictionnary` dont elle implémente toutes les méthodes, en y ajoutant certaines autres. Les tables de hash utilise la valeur que retourne la méthode `hashCode()` des clés, pour organiser le rangement des *entrées* de la table afin que `get()` fonctionne avec efficacité. Les objets utilisés comme clés dans une table de hash devraient avoir leur classe qui outrepassent les méthodes `equals()` et `hashCode()` pour un fonctionnement correct (voir la classe `Object`).

5.3.- Les images

5.3.1.- La génération d'images

Les images, instances de la classe `Image`, peuvent provenir de deux sources différentes :

- chargées à partir d'un fichier grâce aux méthodes `getImage()` des classes `Applet` ou `Toolkit`. Ce fichier pouvant être éventuellement téléchargé à partir d'une URL sur un réseau, son chargement peut prendre un certain temps. C'est pourquoi la création de ce type d'image se fait en deux temps : `getImage()` permet de créer une instance de la classe `Image` tandis que d'autres méthodes se chargent d'initialiser et de surveiller son chargement, et d'attendre la fin de son chargement (Cf. chapitre suivant 5.3.2).
- créées de toute pièce grâce aux méthodes `createImage()` des classes `Component` et `Toolkit`. Ces images sont souvent utilisées comme buffer ou comme bitmap affichés à l'écran qu'une fois leur dessin terminé.

Les méthodes `drawImage()` de la classe `Graphics` permettent d'afficher une image à un point donné en la redimensionnant éventuellement (exemple d'applet – Cf. chapitre A.3.2).

5.3.2. - Le chargement d'images

Comme il est expliqué ci-dessus, la méthode `getImage()` permet créer une instance d'une image. Pour initialiser et surveiller le chargement d'une image et l'utiliser quand elle est partiellement ou entièrement chargée, il existe plusieurs moyens :

- Soit à l'appel de l'une des méthodes `drawImage()` de la classe `Graphics` avec en paramètre une image `img`. Si `img` n'est encore pas chargée, la méthode `drawImage()` débute le chargement de l'image de manière asynchrone et rend la main. Le dernier paramètre de `drawImage()` doit être une instance d'une classe implémentant l'interface `ImageObserver` comme par exemple la classe `Component`. Cette interface ne déclare qu'une seule méthode `imageUpdate()` et l'implémentation de cette méthode dans la classe `Component` redessine le composant pour mettre à jour le dessin de l'image au fur et à mesure de son chargement. Donc, si vous donnez en dernier paramètre le composant dans lequel l'image est affichée, l'image sera affichée automatiquement aussitôt qu'elle est disponible (exemple Cf. chapitre A.3.2) ;

- Soit à l'appel des méthodes `prepareImage(Image img, ImageObserver observer)` ou `prepareImage (Image img, int width, int height, ImageObserver observer)` des classes `Component` et `Toolkit`. Ces méthodes permettent de débuter le chargement de l'image `img` par un autre thread de manière asynchrone et rend la main. Les paramètres `width` et `height` permettent de redimensionner l'image dès son chargement. Le dernier paramètre `observer` doit être une instance d'une classe implémentant l'interface `ImageObserver` et permet de surveiller l'état du chargement de l'image. La classe `Component` implémentant cette interface, vous pouvez utiliser un composant comme paramètre ;

- Soit en utilisant la classe `MediaTracker`, qui permet de surveiller et d'attendre la fin du chargement d'une image. Cette classe qui utilise la méthode `prepareImage ()` et l'interface `ImageObserver`, simplifie la programmation du chargement d'une image.

De plus, les méthodes `checkImage ()` des classes `Component` et `Toolkit` permettent de vérifier l'état du chargement d'une image. Ces méthodes prennent en dernier paramètre une instance d'une classe implémentant l'interface `ImageObserver`, dont la méthode `imageUpdate ()` est appelée pour lui communiquer l'état de l'image.

5.3.3. – La création d'images

Une image peut être créée grâce à la méthode `createImage (int width, int height)` de la classe `Component`. Cette méthode crée une image vierge dans laquelle vous pouvez dessiner grâce aux méthodes de dessin de la classe `Graphics`. Il existe une deuxième version de la méthode `createImage()` disponible dans les classes `Component` et `Toolkit` : `createImage (ImageProducer producer)`. Le paramètre `producer` doit être d'une classe qui implémente l'interface `ImageProducer`. Le package `java.awt.image` fournit deux classes qui implémentent cette interface :

- La classe `MemoryImageSource` permet de créer une image initialisée avec un tableau décrivant la couleur de chacun des points d'une image ;
- La classe `FilterdImageSource` permet de créer une image qui est le résultat de l'application d'un `filtre` sur une image existante.

Un exemple se trouve en annexe A.3.2.

5.3.4. – Les images et les filtres

Java comporte le concept de filtres qui permettent de transformer une image en une autre. Ces filtres dérivent de la classe `ImageFilter`, et permettent toute sorte de transformation. Le package `java.awt.image` fournit deux classes de filtre dérivées de la classe `ImageFilter`, les classes `CropImageFilter` qui permet d'extraire une partie d'une image, et `RGBImageFilter` qui permet de transformer la couleur de chacun des points d'une image ; vous pouvez aussi imaginer toute sorte de filtre.
Une image filtrée est créée grâce à la méthode `createImage (ImageProducer producer)` des classes `Component` et `Toolkit`, avec le paramètre `producer` égal à une instance de la classe `FilteredIlageSource`.

Pour plus d'information sur le fonctionnement du filtrage d'images, voir le tutoriel Java livré avec le JDK.

5.3.5. – La gestion des animations

L'utilisation des threads et des images permet de réaliser rapidement des animations en Java. Comme le montrent l'exemple du chapitre A.3.2, le principe de programmation d'une animation est presque toujours le même : vous créez un thread dont la méthode `run()` utilise une boucle qui à chaque tour affiche une nouvelle image puis arrête le thread courant pendant un laps de temps avec la méthode `sleep()` de la classe threads.

Bien que comme au cinéma, une animation sera en apparence bien fluide à 25 images par seconde (équivalent à un laps de temps entre chaque image de 40 millisecondes), évitez un laps de temps de rafraîchissement aussi court, car les machines ont souvent du mal à suivre.

Si vous utilisez la méthode `repaint()` (comme dans la plupart des cas) pour mettre à jour l'image d'une animation, le « redessin » du fond du composant effectué par la méthode `update()` de la classe `Component` est inutile si vous devez transférer une image à l'écran occupant toute la surface du composant. N'oubliez pas alors de surcharger la méthode `update()` pour qu'elle appelle directement la méthode `paint()` sans redessiner le fond de l'image. Ceci évitera un clignotement désagréable.

Le double buffering

L'exemple qui suit montre comment programmer le « *double buffering* » pour gérer l'animation d'une image générée par un programme. Le double buffering a pour but d'éviter l'effet de clignotement d'une animation (exemple : un texte défilant horizontalement). Le principe est simple : au lieu de dessiner directement à l'écran un dessin qui évolue à chaque laps de temps, on utilise une image dans laquelle on dessine puis qu'on transfère à l'écran.

```
public class ScrollText extends Applet implements Runnable {
    private String texte;
    private Thread threadAnimation;
    private Dimension tailleApplet;
    private int positionTexte, largeurTexte;
    private FontMetrics metrics;

    public void start ()
    {
         // Mise en blanc du fond de l'applet
         setBackground (Color.white); tailleApplet = size ();
         // Récupération du texte à afficher
         texte = getParameter ("Texte");
         positionTexte = tailleApplet.width;
         // Création et démarrage du thread d'animation
         threadAnimation = new Thread (this);
         threadAnimation.start();
    }

    public void stop ()
    {
         threadAnimation.stop ();
    }

    public void run () {
       try {
          while (threadAnimation.isAlive ())
          {
               // Redessin de l'applet et calcul d'une nouvelle position
               repaint ();
               if (positionTexte > -largeurTexte)
                   positionTexte = positionTexte - tailleApplet.height / 2;
               else
                   positionTexte = tailleApplet.width;
               // Arrête le compteur pendant 2/10 de secondes (200 ms)
               Thread.sleep (200);
          }
       }
       catch (InterruptedException e) { }
    }

    public void paint (Graphics gc)
    {
       gc.setColor (Color.black);
       // Création d'une police de caractères et récupération de sa taille
       gc.setFont (new Font ("Times", Font.BOLD, tailleApplet.height - 4));
       if (metrics == null) {
           metrics = gc.getFontMetrics ();
           largeurTexte = metrics.stringWidth (texte);
       }
       // Utilisation d'un rectangle de clipping pour créer une bordure
       gc.clipRect (2, 0, tailleApplet.width - 4, tailleApplet.height);
       // Dessin du texte
       gc.drawString (texte, positionTexte + 2,
                      tailleApplet.height - metrics.getDescent () - 2);
    }
} // applet
```

5.4.- Dessiner avec Java

5.4.1. - La notion de contexte graphique - la classe Graphics

Les outils graphiques de base des bibliothèques Java permettent de tracer sur l'écran des lignes, des formes, des caractères .... La plupart des opérations graphiques sont le fait de méthodes définies dans la classe `Graphics`. Le constructeur de la classe `Graphics` étant déclaré `protected`, il est impossible de construire directement des objets de type `Graphics`. On obtient un tel objet (appelé « contexte graphique ») grâce à la méthode `getGraphics()` de la classe `Component` ou, dans le cas d'une applet (Cf. chapitre 6), comme paramètre de la méthode `paint()` (obligatoirement présente dans l'applet – le contexte graphique est initialisé dans le fichier HTML et passé en paramètre de la méthode `paint()`). Pour une application, il faut récupérer le contexte graphique d'un composant **après l'avoir affiché** en appelant la méthode `getGraphics` sur le composant. La méthode d'affichage du composant initialise le contexte graphique.

Pour une applet :

```
public class Mon_Applet extends java.applet.Applet
{
    public void paint (Graphics g) // le contexte graphique est recupere
    {                              // grace a la methode paint de l'applet
       g.drawRect (20,20,60,60) ;
    }
}
```

Pour une application :

```
public class Mon_Application
{
    Graphics g ;

    JPanel Appli = new JPanel() ;
    …

    Appli.setVisible (true) ;

    g = Appli.getGraphics() ; // on recupere le contexte graphique du panel

    g.drawRect (20,20,60,60) ;
    …
}
```

5.4.2. – Couleur et fontes

Lorsqu'on utilise le paramètre de type `Graphics` fourni par la méthode `paint()`, le composant lui attribue une couleur de dessin (définie par `setForeground()`) et une fonte par défaut (définie par `setFont()`). Ces valeurs peuvent être modifiées par des méthodes de la classe `Graphics`.

- **`setColor(Color)`** permet de fixer une nouvelle couleur de dessin ;
- **`setFont(Font)`** permet de choisir une nouvelle fonte.

D'autre part, on peut obtenir une variable de type `FontMetrics` correspondant à la fonte en cours en utilisant la méthode `getFontMetrics()`. L'affichage de texte se fait avec la méthode `drawString(String, int, int)`.

Pour dessiner un objet dans une couleur particulière, il faut créer une instance de la classe `Color` représentant cette couleur. La classe `Color` définit un objet `Color` standard, stockés dans des variables de classe, facilitant l'obtention d'un objet `Color` pour certaines des couleurs les plus répandues. Par exemple, `Color.red` retourne un objet `Color` représentant le rouge (valeur RVB de 255, 0, 0). Ci-dessous un tableau des couleurs « standard » :

couleur	Valeurs RVB
Color.white	255, 255, 255
Color.black	0, 0, 0
Color.lightGray	192, 192, 192
Color.gray	128, 128, 128
Color.darkGray	64, 64, 64
Color.red	255, 0, 0
Color.green	0, 255, 0
Color.blue	0, 0, 255
Color.yellow	255, 255, 0
Color.magenta	255, 0, 255
Color.cyan	0, 255, 255
Color.pink	255, 175, 175
Color.orange	255, 200, 0

Si la couleur de l'objet n'est pas standard, il faut créer un objet `Color` en utilisant directement les valeurs RVB de la couleur dans le constructeur de l'objet `Color`.

```
Color c = new Color (140,100,112) ;
```

5.4.3. – Dessin et remplissage de formes

La classe `Graphics` permet de dessiner et de remplir des figures ayant des formes élémentaires. La méthode de dessin est préfixée par "`draw`", la méthode de remplissage est préfixée par "`fill`".

Segments et suites de segments

drawLine(int, int, int, int) permet de tracer un segment. Les 4 paramètres sont les coordonnées des extrémités. Pour une suite de segments, on pourra utiliser *drawPolyline(int[], int[], int)* les tableaux d'entiers passés en paramètre contiennent les coordonnées des points à relier; le 3ème paramètre est le nombre de points; si le premier et le dernier point sont confondus, on retrouve `drawPolygon`.

Rectangles

drawRect(int, int, int, int) et fillRect(int, int, int, int) permettent de dessiner et de remplir des rectangles aux côtés verticaux et horizontaux. Les deux premiers paramètres représentent les coordonnées du coin supérieur gauche, les deux autres représentent la largeur et la hauteur.

On peut obtenir des rectangles à coins arrondis avec les méthodes *drawRoundRect(int, int, int, int, int, int) et fillRoundRect(int, int, int, int, int, int).* Dans ce cas les deux derniers paramètres définissent le paramétrage des arrondis.

On peut également dessiner des rectangles avec effet 3D (on n'utilise pas la même couleur pour les 4 côtés) en utilisant *draw3DRect(int, int, int, int, boolean) et fill3DRect(int, int, int, int, boolean)*; le 5ème paramètre de type boolean permet de choisir entre deux effets : en relief ou en creux.

Polygones

drawPolygon(int[], int[], int) et fillPolygon(int[], int[], int) permettent de dessiner et de remplir des polygones. Les deux tableaux d'entiers contiennent les coordonnées des sommets, le 3ème paramètre contient le nombre de sommets.

On peut aussi utiliser la classe Polygon et les méthodes drawPolygon(Polygon) et fillPolygon(Polygon)

Ellipses et cercles

drawOval(int, int, int, int) et fillOval(int, int, int, int) permettent de dessiner et de remplir des ellipses inscrites dans le rectangle dont les caractéristiques sont passées en paramètre. Pour obtenir un cercle, il suffit de prendre la largeur égale à la hauteur, donc les deux derniers paramètres égaux.

Arcs

drawArc(int, int, int, int, int, int) et fillArc(int, int, int, int, int, int) permettent de dessiner des arcs et de remplir des secteurs circulaires. Les 4 premiers paramètres définissent l'ellipse utilisée, les deux derniers les angles de début et de fin de l'arc.

5.4.4. – La classe Graphics2D

Depuis la version 2 de Java, il existe une autre classe, `Graphics2D`, qui hérite de la classe `Graphics`. Cette classe fournit des outils plus sophistiqués que la classe `Graphics` pour la gestion de la géométrie (transformations de coordonnées, gestion des couleurs, des fontes …). Il s'agit désormais de la classe principale pour tout ce qui touche à la gestion des objets géométriques en 2D.

Convertir un objet Graphics en Graphics2D

Certaines méthodes (par exemple la méthode `paint()` d'une applet) ont en paramètre un objet de la classe `Graphics`. Pour utiliser les méthodes spécifiques à la classe `Graphics2D`, il est nécessaire de convertir l'objet :

```
public class Mon_Applet extends java.applet.Applet
  {
      public void paint (Graphics g)
      {
         Graphics2D g2D = (Graphics2D)g ; // conversion
         …

      }
  }
```

Les transformations de coordonnées - la classe `AffineTransform`

Les transformations de coordonnées sont très utiles, elles permettent notamment de travailler avec des valeurs de coordonnées plus simples et plus significatives. Le contexte s'occupe des transformations en pixels. Il existe quatre transformations fondamentales :

- Le changement d'échelle ;
- La rotation ;
- La translation ;
- La déformation linéaire.

Les méthodes `scale`, `rotate`, `translate` et `shear` de la classe `Graphics2D` choisissent une transformation de coordonnées pour le contexte graphique en fonction de l'une ou l'autre des quatre transformations fondamentales. Il est également possible de composer des transformations. D'un

point de vue mathématique, l'ensemble de ces transformations peut être exprimé sous forme matricielle. Ce type de transformation est appelé : transformation affine. Dans la bibliothèque `Java 2D`, la classe `AffineTransform` décrit ce type de transformation. Il est possible de construire directement un objet de cette classe si les paramètres sont connus. Exemple :

```
AffineTransform t = new AffineTransform(a,b,c,d,e) ;
```

Il existe des méthodes de construction, `getRotateInstance`, `getScaleInstance`, `getTranslateInstance` et `getShearInstance`, qui construisent les matrices représentant ces types de transformations. Par exemple, l'appel suivant :

```
t = AffineTransform.getScaleInstance(2.0F,0.5F);
```

renvoie une transformation correspondant à la matrice :

$$\begin{vmatrix} 2 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

Il existe également des méthodes d'instance, `setToRotation`, `setToScale`, `setToTranslate` et `setToShear`, qui choisissent un nouveau type pour un objet de transformation.

Par exemple :

```
t.setToRotation(angle);
```

transforme l'objet « t » en rotation.

6.- Les applets

6.1.- Applet et application : différences

Comme nous l'avons déjà vu au premier chapitre, les applets sont des composants logiciels indépendants, exécutés au sein d'un navigateur (Netscape - Internet Explorer) ou d'un logiciel de visualisation (appletviewer livré avec le JDK). Contrairement aux applications, les programmes comportant des applets ne sont donc pas autonomes. Notons qu'un programme peut être à la fois applet et application. Bien que les créations respectives d'applets et d'applications Java soient soumises à deux ensembles distincts de règles et de procédures, rien ne les oppose. Les aspects propres aux applets sont ignorés lorsque le programme concerne une application et vice versa.

La différence fondamentale entre applet et application est l'ensemble des restrictions auxquelles est soumise l'exploitation des applets pour des raisons de sécurité. Etant donné que les applets peuvent être chargées depuis n'importe quel site Web pour être exécutées sur un système client du serveur client contenant l'applet, les navigateurs et les outils compatibles Java en restreignent les possibilités afin d'empêcher une « applet malfaisante » d'infliger des dommages au système utilisateur ou d'en violer la sécurité.

Les restrictions portent sur les points suivants :

- Les applets ne peuvent ni lire, ni écrire sur le système de fichier de l'utilisateur ;
- Les applets ne peuvent communiquer avec aucun autre serveur de réseau que celui sur lequel elles étaient enregistrées à l'origine ;
- Les applets ne peuvent faire marcher aucun programme sur le système de l'utilisateur ;
- Les applets ne peuvent pas charger des programmes qui résident sur la plate-forme locale.

6.2.- Création d'une applet

Pour créer une applet il faut créer une sous classe de la classe `Applet`. La classe Applet fait partie du package `java.applet` qui fournit l'essentiel des comportements dont une applet a besoin pour fonctionner au sein d'un navigateur. De plus, les applets font largement appel au package AWT de Java.

Une applet peut comporter un grand nombre de classes auxiliaires mais c'est toujours la classe principale qui déclenche son exécution effective. La première classe de l'applet est toujours conforme à la signature suivante :

```
public class myClass extends java.applet.Applet {
    …
}
```

Notons que Java exige qu'une applet soit déclarée publique.

Une application Java élémentaire comporte une classe dotée d'une fonction main, point d'entrée du programme. Les applets suivent un processus similaire mais plus complexe. Elles n'ont pas de fonction main. Elles ont différentes activités liées aux événements majeurs de leur cycle de vie, comme l'initialisation, la peinture et les événements souris. A chacune de ces activités correspond une méthode de sorte que lorsqu'un événement se produit le navigateur appelle la méthode appropriée. Les cinq méthodes les plus importantes pour l'exécution d'une applet concernent **l'initialisation** (méthode `init()`), **le démarrage** (méthode `start()`), **l'arrêt** (méthode `stop()`), **la destruction** (méthode `destroy()`) et **la peinture** (méthode `paint()` – seule méthode indispensable).

La méthode `init()`

L'initialisation se produit lorsque l'applet est chargée (ou rechargée) et s'apparente à la méthode `main()` des applications. L'initialisation d'une applet inclut généralement la lecture des paramètres de l'applet, la création d'objets auxiliaires dont elle peut avoir besoin, la mise à l'état initial, le chargement d'images, de polices …

La méthode `start()`

A la suite de l'initialisation, l'applet doit être démarrée. Le démarrage se distingue de l'initialisation car il peut se produire à des moments très différents dans la vie d'une applet alors que l'initialisation ne se produit qu'une seule fois (par exemple après l'arrêt de l'applet).

La méthode `stop()`

L'arrêt et le démarrage sont complémentaires. L'arrêt de produit par exemple lorsque l'utilisateur quitte la page Web contenant l'applet ou lorsque l'utilisateur arrête lui-même l'applet en invoquant la méthode `stop()`.

La méthode `destroy()`

La destruction permet à l'applet de faire une remise en ordre (libération de la mémoire, fermeture des connexions réseau …) juste avant la sortie du navigateur.

La méthode `paint(Graphics g)`

La peinture recouvre tout ce qu'une applet dessine à l'écran, qu'il s'agisse de texte, d'un arrière plan coloré ou d'une image. La peinture peut se produire des milliers de fois au cours de la vie d'une applet (déplacements et recouvrements des fenêtres …). La méthode **paint** prend un argument, une instance de la classe `Graphics`. Cet objet (qui représente le contexte graphique) est créé par le navigateur.

Un exemple d'applet simple :

```
import java.awt.Graphics;
import java.awt.Font;
import java.awt.Color;

public class HelloApplet extends java.applet.Applet {

   Font f = new Font ("TimeRoman",Font.BOLD,36) ;

   public void paint(Graphics g) {
      g.setFont(f) ;
      g.setColor(Color.red) ;
      g.drawString("Hello !",5,40) ;

  }

}
```

Tout le travail est effectué par la méthode `paint`

6.3.- Inclure une applet dans une page Web

Après avoir créé une classe (ou plusieurs) contenant une applet et l'avoir compilée, il faut créer, au moyen du langage HTML, une page Web contenant l'applet. Pour inclure l'applet dans la page, il existe une balise HTML spéciale. Les navigateurs utilisent l'information contenue dans cette balise pour localiser les fichiers « .class » et exécuter l'applet.

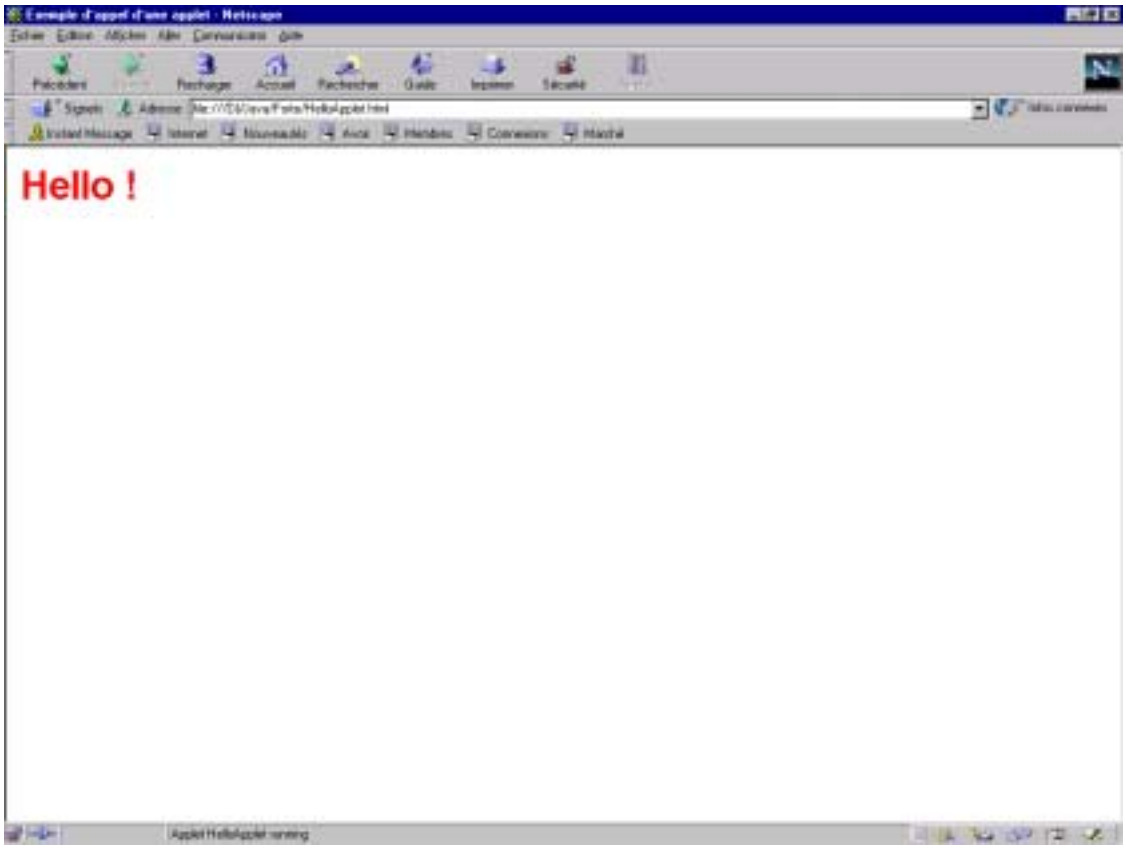
Un exemple de page HTML :

```
<HTML>
<HEAD>
<TITLE> Exemple d'appel d'une applet </TITLE>
</HEAD>
<BODY>
<APPLET CODE ="HelloApplet.class" WIDTH=200 HEIGHT=50>
</APPLET>
</BODY>
</HTML>
```

`CODE` indique le nom du fichier à exécuter

`WIDTH` et `HEIGHT` délimitent la taille de la fenêtre dans laquelle sera dessinée l'applet

Test du résultat avec le navigateur Netscape :



6.4.- Passer des paramètres à une applet

Dans le cas d'une application, les paramètres sont transmis à la méthode `main` sous la forme d'une ligne de commande. Les arguments sont ensuite analysés dans le corps de la classe. Les applets en revanche n'ont pas de ligne de commande mais elles peuvent recevoir différentes indications à partir du fichier HTML par le biais de la balise `<PARAM NAME= … VALUE=>`. Ajoutons une ligne au fichier HTML défini ci-dessus permettant de personnaliser le « Hello » :

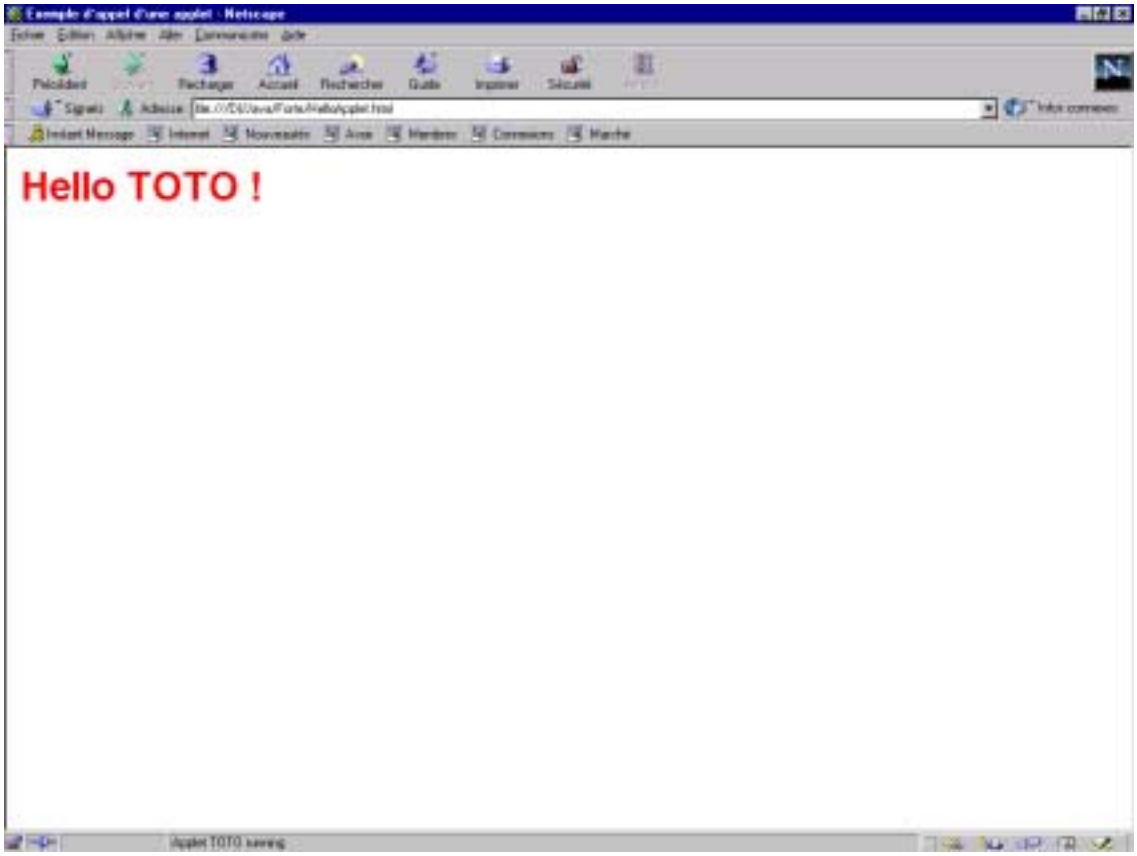
```
<HTML>
<HEAD>
<TITLE> Exemple d'appel d'une applet </TITLE>
</HEAD>
<BODY>
<APPLET CODE =“HelloApplet.class“ WIDTH=200 HEIGHT=50>
<PARAM NAME=name VALUE=“TOTO“>
</APPLET>
</BODY>
</HTML>
```

Pour récupérer la valeur `name` dans le code de l'applet, il existe une méthode spécifique `getParameter`. Le code de l'applet devient alors :

```
import java.awt.Graphics;
import java.awt.Font;
import java.awt.Color;

public class HelloApplet extends java.applet.Applet {

   Font f = new Font (“TimeRoman“,Font.BOLD,36) ;
   String name ;

   public void init() {
      name=getParameter(“name“);
      if (name==null)
         name=“Tout le monde“;

      name=“Hello “+name+“ !“;
   }

   public void paint(Graphics g) {
      g.setFont(f) ;
      g.setColor(Color.red) ;
      g.drawString(name,5,40) ;
  }
}
```

On récupère la valeur de `name`. Si la valeur n'est pas renseignée dans le fichier HTML on indique une valeur par défaut.

Test du résultat avec le navigateur Netscape :



7.- Java3D

Remarque : ce chapitre est fortement inspiré du cours d'Alain Berro, professeur à l'Université de Toulouse (« Découvrez Java 3D » – http://eva.univ-tlse1.fr/). Certaines notions très complexes (texture, éclairage, animation …) ne sont pas traitées en détails dans ce chapitre. Vous trouverez une documentation très complète traitant du sujet sur le site http://www.eteks.com.

7.1.- Les bases de la programmation en Java 3D

7.1.1.- Introduction

Java 3D est une API orientée objet, extension du JDK (Java Development Kit), destinée à la création de scènes en 3D. Elle offre une bibliothèque d'objets de très haut niveau permettant la construction et la manipulation de scènes 3D. Elle bénéficie des avantages intrinsèques à la programmation Java (portabilité, indépendance par rapport au matériel et système d'exploitation, exécution dans un butineur).

7.1.2.- Les premiers pas en Java 3D

7.1.2.1. - Téléchargement et installation

Pour utiliser Java 3D, il est nécessaire d'avoir téléchargé et installé préalablement le JDK de Sun. Ensuite téléchargez l'API Java 3D et installez-là dans le même répertoire que celui dans lequel a été installé le JDK. Lors de l'installation les fichiers `J3D.dll` et `j3daudio.dll` sont ajoutés dans le répertoire `jre/bin` et les fichiers `j3daudio.jar`, `j3dcore.jar`, `j3dutils.jar` et `vecmath.jar` sont ajoutés dans le répertoire `jre/lib/ext`. En fonction de votre environnement, il vous faudra également ajouter le chemin menant à ces fichiers archives (Menu « Projet » de l'environnement JBuilder : Propriété du projet ./ Bibliothèques nécessaires / Ajouter).

7.1.2.2. - Documentations, discussions en ligne et liens

Sun offre une documentation en ligne très riche et très bien réalisée :

http://java.sun.com/products/java-media/3D/

http://developer.java.sun.com/developer/onlineTraining/java3d/

http://java.sun.com/products/java-media/3D/faq.html

Groupes de discussions en ligne :

- comp.lang.java.3d (newsgroup en anglais) ;
- fr.comp.lang.java (newsgroup en français) ;
- java3d-interest (mailing-list en anglais).

Autres sites très intéressants :

- http://www.j3d.org/ ;
- http://www.eteks.com (en français).

7.1.3.- Les principes de l'API Java 3D

L'API Java 3D reprend les idées des principales APIs existantes. Elle synthétise les meilleures idées des APIs bas niveau (Direct3D, OpenGL, ...) et reprend le concept de systèmes basés sur la construction d'un graphe de scène. Elle introduit de nouvelles fonctionnalités telles que le son 3D et la compression des géométries.
Le **graphe de scène** fournit un moyen simple et flexible de représenter une scène en 3D. Il regroupe les objets à visualiser, les informations de visualisation et les éventuels outils d'interaction entre l'utilisateur et les objets.

7.1.3.1. - Le système de coordonnées

Par défaut, Java 3D utilise :

- le système de la main droite comme système de coordonnées ;
- l'axe Z positif orienté vers l'observateur ;
- le mètre comme unité de base du système ;
- le radian comme unité de mesure des angles.

Afin de pouvoir représenter une large gamme de scène, Java 3D permet de modifier l'unité de l'espace des coordonnées : Angstrom, Mètre, Kilomètre, Diamètre de la terre, Année lumière.

7.1.3.2. - Présentation du graphe de scène

Un **univers virtuel** (`VirtualUniverse`) est défini comme un espace tridimensionnel dans lequel est inséré une ou plusieurs scènes (`Locale`).

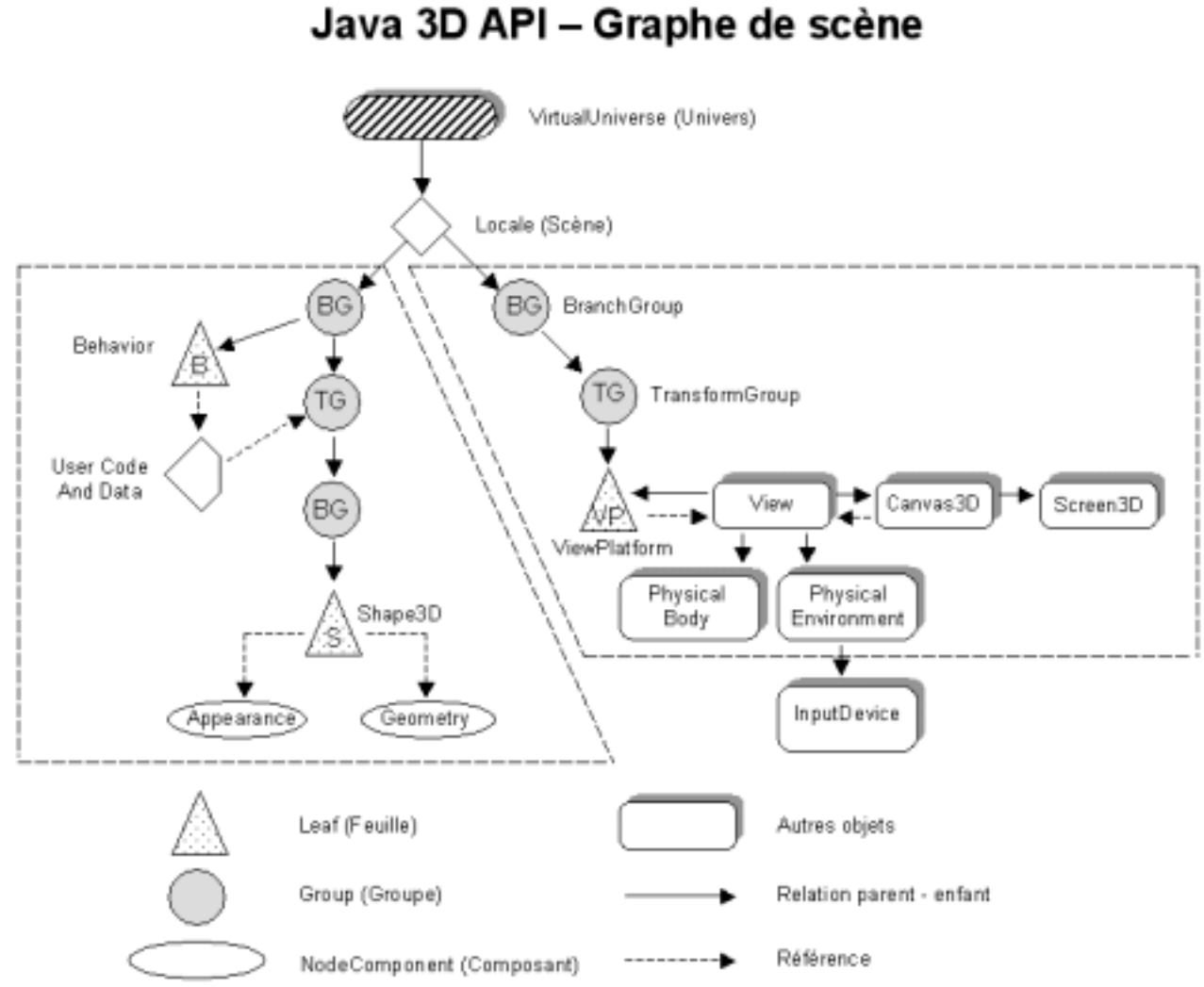
Construire une **scène** consiste à associer une branche de volume et une branche de visualisation.

La **branche de volume** forme la partie gauche du graphe de scène (ci-dessous) et elle va contenir les différents objets physiques (Géométrie, Son, Texte) à visualiser.

La **branche de visualisation** forme la partie droite de la scène (ci-dessous) et contient les outils nécessaires à visualiser l'espace de volume (position du point de vue, projection, style d'affichage ...). Afin d'avoir une visualisation plus réaliste, Java 3D permet également de tenir compte des caractéristiques physiques de la tête de l'utilisateur (`PhysicalBody`) (position des yeux, écartement, ...) et de l'environnement sensoriel (`PhysicalEnvironment`).

Chaque branche est un ensemble structuré de **noeuds** (`Node`) et de **composants de noeud** (`NodeComponent`) liés par des relations. Le point de départ d'une branche est obligatoirement un objet BranchGroup.

La plupart des applications 3D n'utilise qu'une branche visualisation (un seul point de vue) et qu'une seule branche de volume mais il est possible de concevoir des scènes complexes dans lesquelles seront intégrées plusieurs branches de volume et plusieurs branches de visualisation. Cela permet de créer des scènes multi représentation et multi point de vue. Ci-dessous un schéma du graphe de scène de Java3D.



7.1.3.3. -Description des principales classes d'objets formant le graphe de scène

La super structure

- **`VirtualUniverse`** contient l'ensemble des scènes de l'univers ;
- **`Locale`** permet de définir les caractéristiques d'une scène (origine et espace de coordonnées, branche(s) de volume et branche(s) de visualisation).

Les éléments de visualisation

- **`ViewPlatform`** définit la position et l'orientation du point de vue de l'utilisateur ;
- **`View`** regroupe toutes les caractéristiques du processus de rendu. Il contient les paramètres et les références sur les objets définissant la manière de visualiser la scène. Il possède aussi une référence sur les différents objets « `Canvas3D` »;
- **`Canvas3D`** est le cadre d'affichage dans lequel la scène doit être rendue ;
- **`Screen3D`** contient les informations sur les propriétés physiques du périphérique d'affichage ;
- **`PhysicalBody`** contient les informations sur les caractéristiques physiques de l'utilisateur ;
- **`PhysicalEnvironment`** définit l'environnement sensoriel, dans lequel on place les objets nécessaires à une interaction entre l'utilisateur et la scène (périphérique de saisie, périphérique d'interaction sonore, ...).

Les éléments d'interaction

`InputDevice` est une interface permettant de créer un pilote de périphérique.

Les éléments de construction de la structure de la scène (les nœuds)

La structure de la scène est constituée de **nœuds** (`Node`) qui sont soient des **groupes** (`Group`) ou des **feuilles** (`Leaf`). Les nœuds sont reliés entre eux par des relations parent-enfant formant ainsi une hiérarchie en arbre inversé.

- **`Group`** est une super classe qui permet de regrouper et gérer plusieurs nœuds. Un groupe a un et un seul parent mais peut avoir plusieurs enfants.

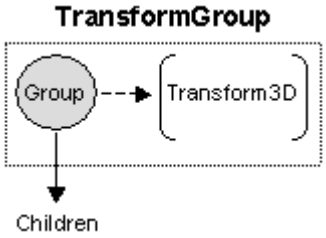
Méthode	Rôle	Capacité du nœud
`AddChild`	ajoute un nœud au groupe	`ALLOW_CHILDREN_EXTEND`
`GetChild`	retourne un nœud	`ALLOW_CHILDREN_READ`
`InsertChild`	insère un nœud	`ALLOW_CHILDREN_EXTEND`
`SetChild`	mise à jour d'un nœud	`ALLOW_CHILDREN_WRITE`
`RemoveChild`	supprime un nœud	`ALLOW_CHILDREN_WRITE`
`NumChildren`	retourne le nombre de nœud	`ALLOW_CHILDREN_READ`

Les classes filles de `Group` les plus usités pour la construction d'un graphe de scène sont les classes `BranchGroup` et `TransformGroup`.

- **`BranchGroup`** est un groupe racine d'un sous graphe de la scène. Il peut être attaché à un autre sous graphe ou à un objet `Locale`. Il peut être compilé en appelant la méthode `compile()`. Un `BranchGroup` attaché à un autre sous graphe peut être détaché pendant l'exécution à condition que l'aptitude associée (à l'action 'détacher') soit mise à jour.

Méthode	Rôle	Capacité du nœud
`Compile`	compile la branche dont le `BranchGroup` est la racine	
`Detach`	détache le `BranchGroup` de son parent	`ALLOW_DETACH`

- **`TransformGroup`** est un groupe de transformation spatiale. C'est un objet `group` enrichi d'un objet `Transform3D` qui va permettre de positionner, orienter ou mettre à l'échelle tous les enfants du groupe.



Méthode	Rôle	Capacité du nœud
`SetTransform`	mise à jour du `Transform3D`	`ALLOW_TRANSFORM_WRITE`
`GetTransform`	récupère le `Transform3D`	`ALLOW_TRANSFORM_READ`

- **`Leaf`** est la super classe parente de tous les objets servant à terminer l'arbre de scène. Par définition, les feuilles n'ont pas de fils : `Shape3D`, `Light`, `Fog`, `Background`, `Behavior`, `ViewPlatform`.

Les composants de nœud

Les composants de nœud sont liés aux nœuds par une relation de référence.
La classe `NodeComponent` est la super classe commune à tous les objets permettant de définir les caractéristiques d'une scène (forme, couleur, texture, matière, ...).

L'optimisation en Java 3D

L'API Java 3D inclus un certain nombre de techniques permettant d'optimiser le temps de rendu d'une scène.

Les bits d'aptitude (Capability bits)

Tous les objets (`SceneGraphObjet`) du graphe de scène (`Node` ou `NodeComponent`) sont susceptibles de subir un certain nombre d'actions (modification ou interrogation). Afin d'optimiser le graphe de scène, Java 3D associe un **bit d'aptitude** (capability) aux différentes actions possibles sur un objet. Si le bit d'apitude n'est pas positionné alors l'action n'est pas possible. L'utilisation des bits d'aptitude est nécessaire si l'objet appartient à une **branche compilée** ou si l'objet est **vivant**. Par exemple, si vous souhaitez ajouter un nœud à un groupe, ce dernier devra avoir la capacité `ALLOW_CHILDREN_EXTEND`.

Méthode	Rôle
`SetCapability`	attribue une aptitude à un objet
`GetCapability`	retourne l'état de l'aptitude
`ClearCapability`	désactive l'aptitude d'un objet

La compilation des branches de l'arbre de scène (Compile)

La compilation d'un sous graphe de scène permet d'optimiser la structure de la scène de manière à avoir un rendu plus rapide. Elle ne peut être effectuée que sur les sous graphes dont la racine est un `BranchGroup` ou un `SharedGroup`. Après la compilation, seuls les objets ayant définis leurs aptitudes pourront être modifiés ou consultés.

Les limites d'influence (Bounds)

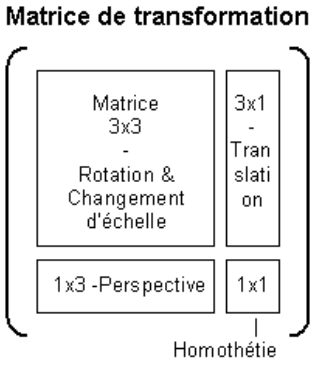
Beaucoup d'objets de l'API Java 3D (`Light`, `Behavior`, `Fog`, `Background`,`Sound`, ...) peuvent avoir une limite spatiale (`Bounds`). Le but de cette limite est de contenir l'influence de l'objet sur la scène pour reduire le temps de calcul du rendu de la scène. Par exemple, on pourra limiter l'influence d'une source lumineuse dans la scène à une certaine distance, ainsi le calcul de l'illumination sera plus rapide avec une perte de précision très faible.

Appearance Bundle

L'ensemble des attributs constituant l'apparence d'un objet est défini dans divers composants. Tous ces composants sont regroupés dans un objet `Appearance`. Cette conception permet de modifier simplement et efficacement chacun des attributs.

Les transformations géométriques

En image de synthèse, afin d'unifier le traitement des transformations géométriques d'une scène, on utilise des **coordonnées homogènes**. Les transformations géométriques sont codées sous la forme d'une matrice 4x4 : translation, rotation, homothétie, changement d'échelle et projection (Cf. schéma ci-dessous).



En 3D, un point *P(X,Y,Z)* a comme coordonnées homogènes *(X,Y,Z,1)*. Après le multiplication par la matrice de transformation on obtient le résultat *(XP, YP, ZP, H)*. Les coordonnées transformées de *P* sont alors *(XP/H, YP/H, ZP/P)*.

Dans Java 3D, la matrice des transformations géométriques est enregistrée dans un objet `Transform3D`. Il n'est pas utile de connaître les formules de transformation, il suffit d'utiliser les méthodes de mise à jour de la classe `Transform3D`.

Méthode	Rôle
`setTranslation`	modifie la transformation en translation
`rotX, rotY, rotZ`	modifie la transformation en rotation d'axe X, Y ou Z
`setScale`	modifie la transformation en changement d'échelle

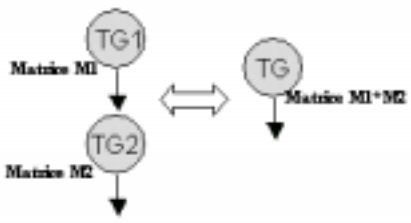
Remarques :

- Les méthodes `setTranslation`, `rotX`, `rotY` et `rotZ` ne sont pas cumulatives. Chaque appel à l'une de ces méthodes suppriment la transformation précédemment contenue dans le `Transform3D`. Alors que la méthode `setScale` combine la mise à l'échelle à la transformation déjà présente dans le `Transform3D`.
- Les transformations s'effectuent toujours par rapport au centre du repère.
  La classe `Transform3D` possède également tous les outils mathématiques permettant de manipuler les matrices de transformation.

Méthode	Rôle
`mul`	permet de multiplier deux `Transform3D`
`add, sub, transpose`	addition, soustraction, transposition
`setIdentity`	la matrice de transformation devient la matrice identité

Remarques :

- Multiplier deux matrices de transformation est identique à l'enchaînement de deux `TransformGroup`.



- La multiplication de matrices n'étant pas commutative, attention à l'ordre de la multiplication.

7.1.4.- Comment écrire un programme en Java 3D ?

7.1.4.1.- Création d'un univers virtuel complet

L'enchainement suivant permet de réaliser la structure d'une scène :

- Créez une classe qui hérite de `JFrame`. Cette classe va servir de fenêtre de visualisation ;
- Créez un objet `Canvas3D` que vous insérez dans votre fenêtre ;
- Créez un `VirtualUniverse` ;
- Créez un objet `Locale` lié au `VirtualUniverse` ;
- Créez un branchement entre l'objet `Locale` avec la partie visualisation à l'aide d'un `BranchGroup` et d'un `TransformGroup`. Ce dernier servira à appliquer éventuellement des déplacements sur le point de vue :
  - Créez un objet `ViewPlatform` et attacher le au `TransformGroup` … ;
  - Créez un objet `View` ;
  - Créez un objet `PhysicalBody` et `PhysicalEnvironment` ;
  - Attacher les objets `PhysicalBody`, `PhysicalEnvironment`, `ViewPlatform` et `Canvas3D` à l'objet `View`.
- Créez un branchement (`BranchGroup`) entre l'objet `Locale` et la partie volume ;
- Réalisez la partie volume et attachée là ;
- Compilez l'arborescence de la partie volume

Code source

```
import java.awt.BorderLayout;
import java.awt.Container;
import java.awt.GraphicsConfiguration;
import javax.swing.JFrame;
import javax.media.j3d.*;
import javax.vecmath.Vector3d;
import javax.vecmath.Point3d;
import com.sun.j3d.utils.geometry.*;
import com.sun.j3d.utils.universe.SimpleUniverse;

public class SceneVU extends Jframe
{
   //----- Données relatives à la fenêtre
   private int largeur;
   // Taille
   private int hauteur;
   private int posx; // Position
   private int posy;

   //----- Objets composants la structure principale
   private VirtualUniverse universe;
   private Locale locale;
   private View view;
   private BranchGroup racineVue; // Nœud de branchement de la vue
   private BranchGroup racineVolume; // Nœud de branchement du volume
   private TransformGroup tgVue; // Nœud de transformation attaché au
                                 // point de vue
   private TransformGroup tgVolume; // Nœud de transformation attaché au
                                    //volume
   // Consructeur
   public SceneVU (int l,int h,int px,int py)
   {
     //Instanciation de la fenêtre graphique
     this.setTitle("Visualisation 3D");
     this.largeur = l;
     this.hauteur = h;
```

```
    this.setSize(largeur,hauteur);
    this.posx = px;
    this.posy = py;
    this.setLocation(posx,posy);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    //----- Contenu de la fenêtre
    Container conteneur = getContentPane();
    conteneur.setLayout(new BorderLayout());

    //----- Création du Canvas
    GraphicsConfiguration config = SimpleUniverse.getPreferredConfiguration();
    Canvas3D c = new Canvas3D(config);
    conteneur.add("Center",c);

    //---- Création de l'univers virtuel
    this.universe = new VirtualUniverse();
    this.locale = new Locale(this.universe);

    //----- Création du nœud pour insérer les éléments de vue
    this.racineVue = new BranchGroup();

    //----- Position de l'observateur
    Transform3D t3d_oeil = new Transform3D();
    t3d_oeil.set(new Vector3d(0.0,0.0,10.0));
    this.tgVue = new TransformGroup(t3d_oeil);
    this.racineVue.addChild(this.tgVue);

    //----- Création de la plateforme de vue et attachement
    ViewPlatform vp = new ViewPlatform();
    this.tgVue.addChild(vp);

    //----- Création d'une vue
    this.view = new View();

    //----- Création de l'univers physique
    PhysicalBody body = new PhysicalBody();
    PhysicalEnvironment env = new PhysicalEnvironment();

    /*----- Liaison de la plateforme de vue, du canvas et de
      l'univers physique à la vue-----*/
    this.view.attachViewPlatform(vp);
    this.view.addCanvas3D(c);
    this.view.setBackClipDistance(100.0);
    this.view.setPhysicalBody(body);
    this.view.setPhysicalEnvironment(env);

    /*----- Création du noeud racine et de la matrice de transformation de la
       branche volume -----*/
    this.tgVolume = new TransformGroup();
    this.racineVolume = new BranchGroup();
    this.racineVolume.addChild(this.tgVolume);
    this.tgVolume.addChild(createBrancheVolume());

    //----- Ajout à Locale de Viewplatform + SceneGraph
    this.locale.addBranchGraph(racineVue)
    this.locale.addBranchGraph(racineVolume);

    //----- Rend la fenêtre visible
    this.setVisible(true);
  }
```

```
   //----- Création du volume
   private BranchGroup createBrancheVolume()
   {
      //----- Création du noeud racine
      BranchGroup racine = new BranchGroup();
      Racine = new BranchGroup();

      //----- Création du Volume
      racine.addChild(new ColorCube());

      //----- Optimisation du graphe de scène
      racine.compile();
      return racine;
   }

   public static void main(String s[])
   {
     SceneVU sc = new SceneVU(200,200,0,0)
   }
}
```

7.1.4.2. - Ecriture simplifiée d'un programme Java 3D

Une manière d'écrire un programme Java 3D de façon plus simplifiée est d'utiliser la classe `SimpleUniverse`. Cette classe permet au programmeur d'éviter la programmation de la branche de visualisation. Le constructeur d'un objet `SimpleUniverse` crée un graphe de scène contenant les objets `VirtualUniverse`, `Locale` et toute la branche de visualisation. La recette devient donc beaucoup plus simple :

- Créez une classe qui hérite de `JFrame`. Cette classe va simplement servir de fenêtre de visualisation ;
- Créez un objet `Canvas3D` que vous insérez dans votre fenêtre ;
- Créez un objet `SimpleUniverse` avec une référence sur l'objet `Canvas3D` ;
- Créez un branchement (`BranchGroup`) entre l'objet `SimpleUniverse` et la partie volume ;
- Réalisez la partie volume et attachée là ;
- Compilez l'arborescence de la partie volume.

Code source

```
import java.awt.BorderLayout;
import java.awt.Container;
import java.awt.GraphicsConfiguration;
import javax.swing.JFrame;
import javax.media.j3d.*;
import javax.vecmath.Vector3d;
import javax.vecmath.Point3d;
import com.sun.j3d.utils.geometry.*;
import com.sun.j3d.utils.universe.SimpleUniverse;

public class SceneSU extends Jframe
{
  //----- Données relatives à la fenêtre
  private int largeur; // Taille
  private int hauteur;
  private int posx; // Position
  private int posy;

  //----- Objets composants la structure principale
  private SimpleUniverse universe;
```

```
  private BranchGroup racineVolume; // Noeud de branchement du volume
  private TransformGroup tgVolume;  // Noeud de transform attaché au volume

  // Constructeur
  public SceneSU (int l, int h, int px, int py)
  {
     //----- Instanciation de la fenêtre graphique
     this.setTitle("Visualisation 3D");
     this.largeur = l;
     this.hauteur = h;
     this.setSize(largeur,hauteur);
     this.posx = px;
     this.posy = py;
     this.setLocation(posx,posy);
     this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

     //----- Contenu de la fenêtre
     Container conteneur = getContentPane();
     Conteneur.setLayout(new BorderLayout());

     //----- Création du Canvas
     GraphicsConfiguration config =SimpleUniverse.getPreferredConfiguration();
     Canvas3D c = new Canvas3D(config);
     Conteneur.add("Center",c);

     //----- Création de l'univers virtuel
     this.universe = new SimpleUniverse(c);

     //----- Position de l'observateur
     Transform3D t3d_oeil = new Transform3D();
     t3d_œil.set(new Vector3d(0.0,0.0,10.0));
     this.universe.getViewingPlatform().getViewPlatformTransform().
                                                    setTransform(t3d_oeil) ;
     //Création noeud racine et matrice de transformation du branche volume
     this.tgVolume = new TransformGroup();
     this.racineVolume = new BranchGroup();
     this.racineVolume.addChild(this.tgVolume);
     this.tgVolume.addChild(createBrancheVolume());

     //----- Ajout de la branche de volume
     this.universe.addBranchGraph(racineVolume);
     //----- Rend la fenêtre visible
     this.setVisible(true);
  }

  //----- Création du volume
  private BranchGroup createBrancheVolume()
  {
     //----- Création du noeud racine
     BranchGroup racine = new BranchGroup();
     Racine = new BranchGroup();

     //----- Création du Volume
     racine.addChild(new ColorCube());

     //----- Optimisation du graphe de scène -----*/
     racine.compile();
     return racine;
  }

  public static void main(String s[])
  {
     SceneSU sc = new SceneSU(200,200,0,0);
  }
}
```

7.2.- Les formes 3D

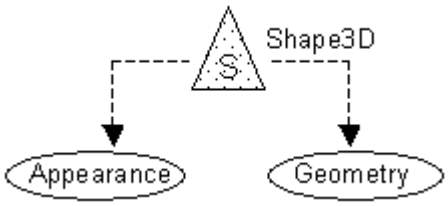
7.2.1.- Introduction

Il y a trois manières différentes de créer un nouveau volume géométrique :

- Utiliser les formes géométriques primitives fournies par l'API ;
- Créer soit même la forme et l'apparence du volume ;
- Utiliser un **loader**. Actuellement, Sun fournit dans l'API Java 3D deux loaders :
  - La classe `ObjectFile` pour charger les objets `.obj` (Wavefront) ;
  - La classe `Lw3dLoader` pour charger les objets `.lws` (Lightwave 3D).

Si vous recherchez un loader pour une autre librairie graphique, regardez chez The Java 3D Community.

Pour construire un volume (`Shape3D`), il faut définir sa géométrie (`Geometry`) et son apparence (`Appearance`). Les objets `Shape3D` ne contiennent aucune information sur la forme ou la couleur du volume. Ces informations vont être stockées dans les objets (`NodeComponent`) dont l'objet `Appareance` possède les références.



Remarque :

Tant que le `Shape3D` n'est pas vivant ou compilé, les composants de l'objet peuvent être modifiés ou consultés, sinon il faudra positionner les aptitudes appropriées.

Méthode	Rôle	Capacité du nœud
`setGeometry`	mise à jour de la forme	`ALLOW_GEOMETRY_READ`
`setAppearance`	mise à jour de l'apparence	`ALLOW_APPEARANCE_READ`
`getGeometry`	retourne la forme	`ALLOW_GEOMETRY_WRITE`
`getAppearance`	retourne l'apparence	`ALLOW_APPEARANCE_WRITE`

7.2.2.- Les primitives géométriques

L'API Java 3D contient cinq primitives géométriques simples dans le package `com.sun.j3d.utils.geometry`. Quatre de ces primitives ne présentent aucune information d'apparence et héritent de la classe `Primitive` :

- `Box`, parallélépipède de dimension 1.0 par défaut ;
- `Cone`, cône de rayon 1.0 et de hauteur 2.0 avec son axe central aligné sur l'axe Y ;
- `Cylinder`, cylindre de rayon 1.0 et de hauteur 2.0 avec son axe central aligné sur l'axe Y ;
- `Sphere`, sphère de rayon 1.0.

Méthode héritée de la classe `Primitive`	Rôle	Capacité du nœud
`setAppearance`	mise à jour de l'apparence	`ENABLE_APPEARANCE_MODIFY`
`getAppearance`	retourne l'apparence	`ENABLE_APPEARANCE_MODIFY`

Remarque :

Pour utiliser ces primitives, l'utilisateur doit obligatoirement définir une apparence par défaut (`setAppareance(new Appearance()) ;`) sinon la primitive n'apparaitra pas dans la scène. La dernière primitive (`ColorCube`) est un cube avec six faces de couleurs différentes et de côté de taille 1.0. Cette primitive hérite directement de la classe Shape3D.

7.2.3.- Construction d'une nouvelle forme

L'API Java 3D présente une classe abstraite `GeometryArray` dans laquelle peut être définie les tableaux contenant les informations sur chaque sommet (coordonnées de position, couleurs, normales de surface, coordonnées de texture (points de clipping) qui permettent de décrire les géométries de type :point, ligne, triangle et polygone.

Pour construire un objet `GeometryArray`, il faut :

- Construire un objet `GeometryArray` vide ;
- Remplir les tableaux de données ;
- Attacher la géométrie à la `Shape3D`.

Construction d'un objet vide

Lors de la construction d'un objet `GeometryArray`, il faut définir deux paramètres :

- le nombre de sommets nécessaires (`vertexCount`) ;
- le type de données stockées pour chaque sommet (`vertexFormat`).

L'unique constructeur de `GeometryArray` est : **`GeometryArray(int vertexCount, int vertexFormat)`**

Format	Description
`COORDINATES`	spécifie que l'objet contient un tableau de coordonnées des sommets
`NORMALS`	spécifie que l'objet contient un tableau de normales
`COLOR_3`	spécifie que l'objet contient un tableau de couleurs hormis la transparence
`COLOR_4`	spécifie que l'objet contient un tableau de couleurs avec transparence
`TEXTURE_COORDINATE_2`	spécifie que l'objet contient un tableau des coordonnées de texture 2D
`TEXTURE_COORDINATE_3`	spécifie que l'objet contient un tableau des coordonnées de texture 3D

Remarque :

S'il y a plusieurs formats, il faut indiquer les constantes séparées par l'opérateur COORDINATES | NORMALS | TEXTURE_COORDINATE_2 . Pour chaque constante de format spécifiée lors de la création de la géométrie, il y a un tableau de taille `vertexCount` créé afin d'accueillir les données.

Remplissage de l'objet

Après la déclaration du format du `GeometryArray` dans le constructeur, l'utilisateur doit remplir les tableaux de données correspondants. Pour cela les méthodes suivantes ont été définies :

Méthode	Rôle	Capacité
`setColor` `setColors`	permet de remplir le tableau de couleurs.	`ALLOW_COLOR_WRITE`
`setCoordinate` `setCoordinates`	permet de remplir le tableau des coordonnées des points formant la géométrie.	`ALLOW_COORDINATE_WRITE`

`setNormal` `setNormals`	permet de remplir le tableau des normales.	`ALLOW_NORMAL_WRITE`
`setTextureCoordinate` `setTextureCoordinates`	permet de remplir le tableau des coordonnées de texture.	`ALLOW_TEXCOORD_WRITE`

Remarques :

- Les méthodes présentées ci-dessus ont un grand nombre de spécifications différentes. Consultez la spécification de `GeometryArray` afin de choisir la méthode la plus appropriée à votre besoin.
- Comme on peut le voir dans ce tableau, il a été défini un ensemble d'aptitudes dans la classe `GéometryArray` qui autorise ou non la manipulation des différentes données.

7.2.4.- Les sous-classes de la classe GeometryArray

La classe `GeometryArray` a été dérivée en plusieurs classes qui offrent des algorithmes différents de construction de géométrie . Dans l'exemple présenté ci-dessus, on a utilisé une classe `QuadArray` qui permet de définir un cube comme 6 quadrilatères.

Type de géométrie	Description
`PointArray`	Ensemble de points isolés
`LineArray`	Ensemble de lignes construites en reliant les sommets du tableau deux à deux
`TriangleArray`	Ensemble de triangles construits en associant les sommets trois à trois. Chaque triangle représente une facette
`QuadArray`	Ensemble de quadrilatères construits en utilisant les sommets quatre par quatre. Les sommets du quadrilatère doivent être coplanaires

Les quatre classes ci-dessus ne permettent pas de réutiliser les sommets définis dans le tableau alors que certaines configurations géométriques invitent à la réutilisation des sommets. Par exemple, la classe **`Cube`** utilise 24 sommets pour décrire la forme, or un cube n'a que 8 sommets. Une meilleure représentation de cette classe permettrait un gain de place. L'API Java 3D a défini deux familles de classes permettant de réutiliser les sommets :

- La classe **`GeometryStripArray`** : classe abstraite à partir de laquelle des primitives de bande sont dérivées (`LineStripArray`, `TriangleStripArray` et `TriangleFanArray`) ;
- La classe **`IndexedGeometryArray`**. Les objets `IndexedGeometryArray` fournissent un niveau supplémentaire d'adressage pour éviter la redondance des informations sur les sommets. Les positions, les couleurs, les normales et les coordonnées de clipping de texture pour les sommets sont stockées dans des tableaux appelés les "**data array**". Des tableaux supplémentaires nommés "**index array**" sont ajoutés afin d'indexer les données contenues dans les data array. Les index array peuvent avoir plusieurs références au même sommet dans les data array et l'ordre des références indique l'ordre dans lequel les sommets seront traités pendant le rendu.

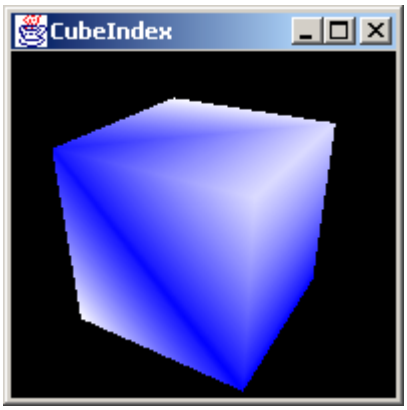
Remarques :

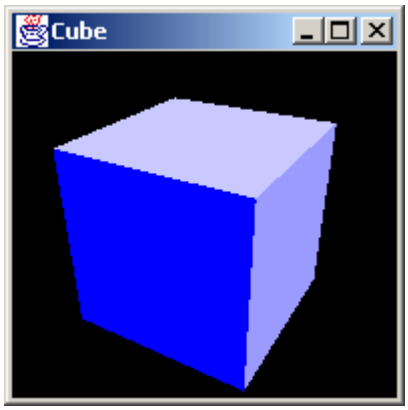
- L'indexation implique un nombre de calculs supplémentaires et peut entraîner une baisse de performance du rendu.
- Cette classe se dérive en `IndexedLineGeometry`, `IndexedTriangleArray`, ...

Exemple : la classe `CubeIndex`

La classe `CubeIndex` est un cube représenté sous forme d'un `IndexedQuadArray`.

Le rendu de CubeIndex est plus réaliste en situation d'éclairage que celui de Cube car les normales sont des normales aux sommets alors que les normales dans Cube sont des normales de surfaces.





7.3.- Les attributs d'apparence

7.3.1.- Le composant de nœud Appearance

Nous avons vu dans la partie précédente que nous pouvions attribuer une couleur à chaque sommet d'une géométrie. Mais cette information est insuffisante pour réaliser le rendu réaliste d'un objet plongé dans une scène illuminée. Il faut donc définir une apparence (`Appearance` sous classe de `NodeComponent`) qui va regrouper les caractéristiques de couleurs, de matière, d'affichage de la géométrie, de texture, de transparence .... Un objet `Appearance` ne contient aucune information sur la façon dont l'objet 3D va apparaître, mais il regroupe l'ensemble des références sur les différentes classes d'attributs d'apparence :

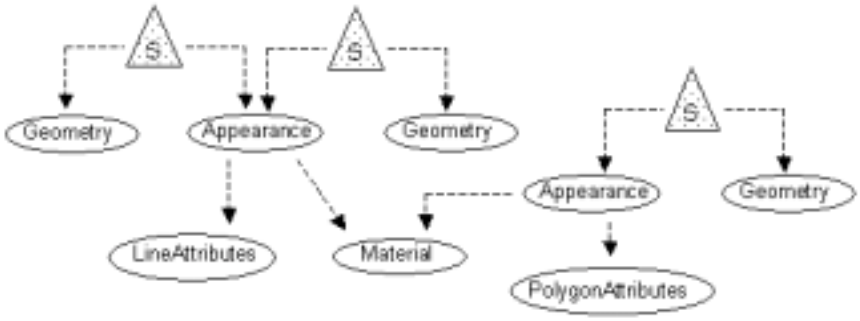
- Les attributs d'affichage de la géométrie :
  - `PointAttributes`
  - `LineAttributes`
  - `PolygonAttribute`
- Les attributs de rendu :
  - `ColoringAttributes`
  - `TransparencyAttributes`
  - `Texture & TextureArributes`
  - `Material`

Le regroupement de l'ensemble des attributs d'apparence dans un objet `Appearance` est nommé **appearance bundle** et permet une optimisation du rendu. Chaque objet `Appearance` possède une méthode lui permettant d'accéder (`get` + nom de l'attribut) et de mettre à jour (`set` + nom de l'attribut) chaque attribut. Chacune de ces actions est associée à une capacité. Le tableau ci-dessous présente les méthodes et les capacités de trois attributs :

Méthode	Rôle	Capacité
`SetPointAttributes` `getPointAttributes`	mise à jour de l'attribut `Point` retourne l'attribut `Point`	`ALLOW_POINT_ATTRIBUTES_WRITE` `ALLOW_POINT_ATTRIBUTES_READ`
`SetMaterial`	mise à jour de l'attribut	`ALLOW_MATERIAL_WRITE`

`getMaterial`	`Material` retourne l'attribut `Material`	`ALLOW_MATERIAL_READ`
`SetLineAttributes` `getLineAttributes`	mise à jour de l'attribut `Line` retourne l'attribut `Line`	`ALLOW_LINE_ATTRIBUTES_WRITE` `ALLOW_LINE_ATTRIBUTES_READ`

Le constructeur `Appearance()` de la classe `Appearance` crée un objet avec chaque référence à un composant d'objet à null. Dans ce cas tous les attributs sont initialisés avec leur valeur par défaut.
Il est important de noter que **les attributs d'apparence peuvent être partagés** par plusieurs objets. Cela permet d'améliorer la performance du rendu en évitant de refaire certains calculs.



7.3.2.- Les classes d'attributs d'apparence

La classe `PointAttributes`

Les objets `PointAttributes` spécifient le rendu des points.

Propriété	Valeur par défaut
taille du point en pixel	`1`
antialiasing	`disable`

Constructeurs :

- `PointAttributes()` crée un objet avec les propriétés par défaut ;
- `PointAttributes (int pointSize, boolean pointAntialiasing).`

L'utilisation de la technique de l'antialiasing demande plus de calculs mais les résultats sont plus satisfaisants.

Méthode	Rôle	Capacité
`setPointSize`	mise à jour de la taille des points	`ALLOW_SIZE_WRITE`
`setPointAntialiasingEnable`	active ou désactive l'antialiasing	`ALLOW_ANTIALIASING_WRITE`

La classe `LineAttributes`

Les objets `LineAttributes` spécifient le rendu des lignes.

Propriété	Valeur par défaut
largeur de la ligne en pixel	`1`
antialiasing	`disable`
dessin (pattern)	`PATTERN_SOLID`

Le tracé d'une ligne peut être solide (`PATTERN_SOLID`), en pointillé (`PATTERN_DOT`), en tiret (`PATTERN_DASH`) ou les deux (`PATTERN_DASH_DOT`).

Constructeurs :

- `LineAttributes()` crée un objet avec les propriétés par défaut ;
- `LineAttributes(float lineWidth, int linePattern, boolean lineAntialiasing).`

L'utilisation de la technique de l'antialiasing demande plus de calculs mais les résultats sont plus satisfaisants.

Méthode	Rôle	Capacité
`setLineWidth`	mise à jour de la largeur d'une ligne	`ALLOW_WIDTH_WRITE`
`setLineAntialiasingEnable`	active ou désactive l'antialiasing	`ALLOW_ANTIALIASING_WRITE`
`setLinePattern`	mise à jour du dessin des lignes	`ALLOW_PATTERN_WRITE`

La classe `PolygonAttributes`

Les objets `PolygonAttributes` spécifient le rendu des polygones.

Propriété	Valeur par défaut
mode de tramage	`POLYGON_FILL`
élimination des facettes	`CULL_BACK`

Il y a trois modes de tramage du polygone.

- `POLYGON_POINT`, le polygone est représenté par ses sommets ;
- `POLYGON_LINE`, le polygone est représenté en fil de fer ;
- `POLYGON_FILL`, le polygone est plein.

Il y a également trois modes d'élimination des facettes.

- `CULL_BACK`, élimination des faces arrière ;
- `CULL_FRONT`, élimination des faces avant ;
- `CULL_NONE`, aucune élimination.

Constructeur :

`PolygonAttributes()` crée un objet avec les propriétés par défaut

Méthode	Rôle	Capacité
`setCullFace`	mise à jour du mode d'élimination des facettes	`ALLOW_CULL_FACE_WRITE`
`setPolygonMode`	mise à jour du mode de tramage	`ALLOW_MODE_WRITE`

La classe `ColoringAttributes`

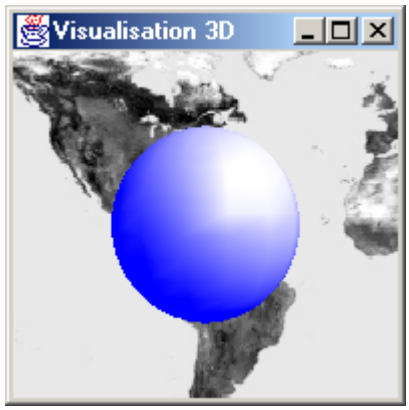
Les objets `ColoringAttributes` spécifient la couleur d'une primitive.

Propriété	Valeur par défaut
couleur	`(1,1,1) blanc`
mode d'interpolation (shading)	`SHADE_GOURAUD`

Java 3D utilise le **mode RGB** (Red, Green, Blue) pour composer les couleurs. La classe `Color3f` permet de créer n'importe quelle couleur du spectre en donnant l'influence de chaque couleur primaire. Exemples : Noir (0,0,0), Jaune (1,1,0), Vert (0,1,0), ...

Le **mode d'interpolation** permet de calculer l'intensité de la couleur en chaque point de la surface à partir des couleurs aux sommets et des normales.

- `SHADE_FLAT`, calcule l'éclairement moyen sur une surface (calcul rapide mais mauvais rendu, figure ci-dessous à gauche) ;
- `SHADE_GOURAUD`, calcule une interpolation linéaire entre les sommets de la surface (calcul plus long mais très bon rendu).



Si la couleur de la forme 3D a été définie au niveau de la géométrie, il y a conflit . Dans ce cas, la priorité est donnée à la couleur définie dans la géométrie.

Constructeurs :

- `ColoringAttributes()` crée un objet avec les propriétés par défaut ;
- `ColoringAttributes(Color3f color, int shadeModel)` ;
- `ColoringAttributes(float red, float green, float blue, int shadeModel)`.

Méthode	Rôle	Capacité
`setColor`	mise à jour de la couleur	`ALLOW_COLOR_WRITE`
`setShadeModel`	mise à jour du mode d'interpolation	`ALLOW_SHADE_MODEL_WRITE`

La classe `TransparencyAttributes`

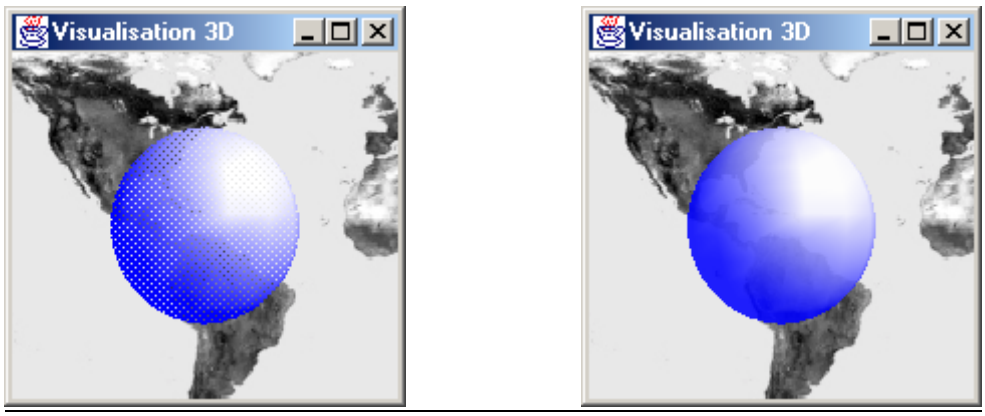
Les objets TransparencyAttributes spécifient la transparence (blending) d'une primitive.

Propriété	Valeur par défaut
opacité (alpha blending)	`0.0`
mode de transparence	`NONE`

La valeur de l'opacité d'un objet est comprise entre 0.0 (objet totalement opaque) et 1.0 (objet totalement transparent).

Le **mode de transparence** détermine la manière de calculer la transparence d'un objet.

- `NONE`, pas d'application de transparence ;
- `SCREEN_DOOR`, construction d'une trame dont certains pixels laissent passer la lumière (calcul rapide mais résultat très mauvais, figure ci-dessous à gauche) ;
- `BLENDED`, effectue un mélange des couleurs (calcul plus long mais le résultat est bien meilleur).



Constructeurs :

- `TransparencyAttributes()` crée un objet avec les propriétés par défaut ;
- `TransparencyAttributes(int tMode, float tVal)`.

Méthode	Rôle	Capacité
`setTransparency`	mise à jour de l'opacité	`ALLOW_VALUE_WRITE`
`setTransparencyMode`	mise à jour du mode de transparence	`ALLOW_MODE_WRITE`

Les classes `Texture` et `TextureAttributes`

Le tutoriel Java 3D consacre un chapitre entier à la création et l'application de texture.
La classe `Texture` permet de définir les caractéristiques d'une texture.

Propriété	Valeur par défaut
mapping	`true`
image	`null`
largeur et hauteur	`0 & 0`
format de l'image	`RGB`

L'application d'une texture est une technique simple et qui donne un rendu très réaliste. Le plus simple pour créer une texture est de charger une image à l'aide de la classe `TextureLoader` et de récupérer la texture créée. Cette classe permet de charger une image provenant d'un **fichier image** local ou distant à travers son URL.

Exemple de chargement et d'application d'une texture

```
TextureLoader textload = new TextureLoader("./image/Duke.gif", this);
Texture text = textload.getTexture();
Appearance app = new Appearance();
app.setTexture(text);
```

La classe `TextureAttributes` définit le mode d'application d'une texture sur une surface.

Propriété	Valeur par défaut
mode d'application	`REPLACE`
couleur de mélange	`(0,0,0) noir`
transformation de la texture	`null`
correction perspective	`NICEST`

Il y a quatre **modes d'application de texture** :

- `REPLACE`, remplace la couleur de la surface par la texture ;
- `DECAL`, **calque** la texture sur la surface ;
- `BLEND`, mélange la couleur de la surface avec la couleur de mélange de la texture ;
- `MODULATE`, combine la couleur de la surface avec la texture ;

La transformation permet de modifier l'aspect de la texture (étirement, écrasement, ...).

Le mode de correction perspective permet de définir la manière de calculer le rendu de la texture en présence d'un effet de perspective.

- `FASTEST`, privilégie la rapidité ;
- `NICEST`, privilégie l'aspect du rendu.

Constructeurs :

- `TextureAttributes() crée un objet avec les propriétés par défaut ;`
- `TextureAttributes(int textureMode, Transform3D transform, Color4f textureBlendColor, int perspCorrectionMode).`

Méthode	Rôle	Capacité
`setTextureMode`	mise à jour du mode d'application	`ALLOW_MODE_WRITE`
`setTextureBlendColor`	mise à jour de la couleur de mélange	`ALLOW_BLEND_COLOR_WRITE`
`setTextureTransform`	applique une transformation	`ALLOW_TRANSFORM_WRITE`
`setPerspectiveCorrectionMode`	mise à jour du rendu en perspective	

7.3.3.- L'arrière plan de la scène

L'arrière-plan d'une scène (`Background`) peut être défini à partir :

- d'une couleur ;
- d'une image ;
- d'une branche de volume dans laquelle sera définie des géométries.

Remarques :

- Nous pouvons également combiner couleur & branche de volume ou image & branche de volume ;
- Mais si nous définissons une couleur et une image d'arrière-plan alors l'image a priorité sur la couleur.

L'utilisation d'une image comme arrière-plan n'est pas recommandée si le point de vue de l'observateur bouge. En effet, l'image étant plaquée sur le fond de la scène, l'arrière-plan ne se modifie pas lors des mouvements du point de vue, les impressions de profondeur et d'horizon sont alors totalement faussées. Lors de réalisation d'un arrière-plan, il faut également définir ses limites d'activation à l'aide d'un objet `BoundingSphere`.

Constructeurs :

- `Background();`
- `Background(Color3f couleur)` définit la couleur d'arrière-plan ;
- `Background(float r, float g, float b);`
- `Backgroung(Branchgroup branch)` attache une géométrie comme arrière-plan ;
- `Background(ImageComponent2D image)` définit l'image d'arrière-plan.

Méthode	Rôle	Capacité
setColor	mise à jour de la couleur	ALLOW_COLOR_WRITE
setGeometry	mise à jour de la branche de volume	ALLOW_GEOMETRY_WRITE
setImage	mise à jour de l'image	ALLOW_IMAGE_WRITE
setApplicationBounds	mise à jour des limites d'application	ALLOW_APPLICATION_BOUNDS_WRITE

Pour ajouter un arrière-plan à la scène, il faut :

- Créer un objet `Background` ;
- Mettre à jour ses caractéristiques de couleur, géométrie ou image ;
- Définir ses limites (`Bounds`) ;
- L'attacher au graphe de scène.

Définir une couleur comme arrière-plan

Code à ajouter

```
BoundingSphere bounds = new BoundingSphere(new
Point3d(0.0,0.0,0.0),100.0);
Background bg = new Background(1.0f,0.0f,0.0f);
bg.setApplicationBounds(bounds);
racine.addChild(bg);
```

Définir une image comme arrière-plan

Code à ajouter

```
BoundingSphere bounds = new BoundingSphere(new
Point3d(0.0,0.0,0.0),100.0);
TextureLoader bgTexture = new TextureLoader("./Fond.jpg",this);
Background bg = new Background(bgTexture.getImage());
bg.setApplicationBounds(bounds);
racine.addChild(bg);
```

Définir une branche de volume comme arrière-plan

Pour définir une branche de volume comme arrière plan, il faut imaginer que nous sommes en train de peindre l'intérieur d'une sphère de rayon égal à 1 car tous les points des géometries doivent être définis à une distance de une unité. Lors du rendu les points sont projétés à l'infini pour former l'arrière plan de la scène.

Code à ajouter

```
BoundingSphere bounds = new BoundingSphere(new Point3d(0.0,0.0,0.0),100.0) ;
Background bg = new Background();
bg.setGeometry(createGeometry());
bg.setApplicationBounds(bounds);
racine.addChild(bg);
```

7.4.- Eclairage d’une scène

7.4.1.- Les différents types de lumière

7.4.1.1. - La lumière ambiante

La lumière ambiante est une lumière d'intensité constante dans tout l'espace et dans toutes les directions. Elle suffit à éclairer un objet. Mais si ce dernier est d'une couleur uni, l'observateur ne pourra pas distinguer le moindre effet de relief car celui-ci est essentiellement perçu par la différence de luminosité entre des points proches sur la surface de l'objet.

Exemple : Lors d'une journée avec couverture nuage uniforme, nous ne distinguons aucune ombre et ne sommes pas capable de donner la position du soleil. Nous sommes dans une scène dont la lumière ambiante est majoritaire. Nous avons également constaté, dans les corrigés de la partie précédente, qu'en présence de lumière ambiante un objet plein apparaît comme une figure 2D.

7.4.1.2. - La lumière diffuse

La lumière diffuse est une lumière réflechie qui dépend essentiellement de la position de la source lumineuse et du matériau de l'objet. Pour un point précis de sa surface l'objet va émettre la même quantité lumineuse dans toutes les directions. Donc peu importe la position de l'observateur, il percevra la même luminosité venant de ce point. Mais comme l'objet ne présente pas les points de sa surface sous le même angle à une source lumineuse, la quantité de lumière réfléchie diffère en fonction du point considéré. Ainsi, l'observateur va percevoir le relief de l'objet.

Exemple : Le tableau noir utilisé dans les écoles est un objet parfaitement diffus. Il permet à n'importe quel élève de la classe de percevoir de la même façon les caractères écrits au tableau tout en évitant les effets de contre jour dus à la lumière spéculaire.

7.4.1.3. - La lumière spéculaire

La lumière spéculaire est une lumière réfléchie qui traduit la capacité de reflexion du matériau. Un objet parfaitement spéculaire va réfléchir la lumière issue de la source lumineuse avec un angle opposé (par rapport à la normale à la surface de l'objet) à l'angle de réception.

Exemple : Un miroir est un objet parfaitement spéculaire.

7.4.1.4. - La lumière émise

La lumière émise traduit la couleur du matériau.

7.4.2.- Les sources lumineuses

L'API Java 3D définit quatres classes de source de lumière qui héritent de la classe abstraite `Light`. Celle-ci permet de définir les aptitudes de l'objet, la couleur de la lumière, l’état de la lumière et la zone d'influence de la source lumineuse.

Méthode	Rôle	Capacité
`SetColor`	mise à jour de la couleur de la lumière	`ALLOW_COLOR_WRITE`
`SetEnable`	mise à jour de l'état	`ALLOW_STATE_WRITE`
`SetInfluencingBoundingLeaf`	mise à jour de la zone d'influence	`ALLOW_INFLUENCING_BOUNDS_WRITE`

7.4.2.1. - Source de lumière ambiante

Cette source (`AmbientLight`) permet d'éclairer la scène de façon uniforme. Ainsi l'observateur pourra percevoir la présence des géométries.

Propriété	Valeur par défaut
couleur	`(1,1,1) blanche`
état	`on (allumé)`

Constructeurs :

- `AmbientLight();`
- `AmbientLight(boolean lightOn, Color3f color);`
- `AmbientLight(Color3f color).`

7.4.2.2. - Source de lumière directionnelle

Source de lumière (`DirectionalLight`) d'intensité constante dont les rayons sont colinéaires à une direction donnée. Le soleil peut être considéré comme une source lumineuse directionnelle car des objets terrestres proches recoivent des rayons de même direction et avec la même intensité.

Propriété	Valeur par défaut
Couleur	`(1,1,1) blanche`
Etat	`on (allumé)`
Direction	`(0.0,0.0,-1.0)`

Constructeurs :

- `DirectionLight();`
- `DirectionalLight(boolean lightOn, Color3f color, Vector3f direction);`
- `DirectionalLight(Color3f color, Vector3f direction).`

7.4.2.3. - Source de lumière ponctuelle

Source de lumière (`PointLight`) omnidirectionnelle dont l'intensite diminue en fonction de l'éloignement de la géométrie éclairé (Exemple : une ampoule au plafond).

Propriété	Valeur par défaut
Couleur	`(1,1,1) blanche`
état	`on (allumé)`
position	`(0.0,0.0,0.0)`
point d'atténuation	`(1.0,0.0,0.0)`

Constructeurs :

- `PointLight()`
- `PointLight(boolean lightOn, Color3f color, Point3f position, Point3f attenuation)`
- `PointLight(Color3f color, Point3f position, Point3f attenuation)`

7.4.2.4. - Source de lumière ponctuelle dirigée

Source de lumière (`SpotLight`) dont les rayons sont contenus dans un cône de lumière.

Propriété	Valeur par défaut
couleur	`(1,1,1) blanche`
état	`on (allumé)`
position	`(0.0,0.0,0.0)`
point d'atténuation	`(1.0,0.0,0.0)`
direction	`(0.0,0.0,-1.0)`
angle du cône de lumière	`PI`
concentration	`0`

Constructeurs :

- `SpotLight();`
- `SpotLight(boolean lightOn, Color3f color, Point3f position, Point3f attenuation, Vector3f direction, float spreadAngle, float concentration);`
- `SpotLight(Color3f color, Point3f position, Point3f attenuation, Vector3f direction, float spreadAngle, float concentration).`

7.4.2.5. - Comment insérer la source lumineuse dans votre scène ?

- Déclarez votre source lumineuse ;
- Mettez à jour ses propriétés ;
- Attribuez-lui un volume d'influence (Par exemple une `BoundingSphere`) ;
- Insérez la ou le(s) source(s) lumineuse(s) de préférence à la racine de votre graphe de volume.

7.4.3.- La classe Material

La classe `Material` regroupe les caractéristiques lumineuses d'une géométrie. Pour cela, nous devons définir les couleurs **ambiante**, **diffuse**, **spéculaire** et **émise** ainsi qu'une valeur de **brillance** (shininess) de la matière.

Les couleurs ambiante, diffuse et spéculaire sont utilsées pour calculer la lumière réfléchie par l'objet (la couleur émise ne sert qu'à déterminer la couleur de l'objet).

La couleur émise est prioritaire sur les attributs de couleur déjà définis.

La brillance est uniquement utilisée pour calculer la lumière spéculaire renvoyée par l'objet.

Propriété	Valeur par défaut
couleur ambiante	`(0.2,0.2,0.2)`
couleur diffuse	`(1.0,1.0,1.0)`
couleur spéculaire	`(1.0,1.0,1.0)`
couleur émise	`(0.0,0.0,0.0)`
brillance	`64`

Constructeurs :

- `Material();`
- `Material(Color3f ambientColor, Color3f emissiveColor, Color3f diffuseColor, Color3f specularColor, float shininess).`

On peut également utiliser les méthodes `setAmbientColor`, `setDiffuseColor` … pour spécifier chacune des couleurs

7.5.- Interaction

7.5.1.- Manipulation à l'aide de la souris

Java 3D propose un paquetage définissant le comportement de la souris à partir de la classe abstraite (`MouseBehavior`). Les trois classes filles de `MouseBehavior` permettent de réaliser les opérations de translation (`MouseTranslate`), de rotation (`MouseRotate`) et de zoom (`MouseZoom`) sur une branche de la scène.

Par défaut, chaque mouvement est associé à un bouton de la souris.

Mouvement	Action
rotation	bouton gauche de la souris
translation	bouton droit de la souris
zoom	bouton du milieu ou ALT + bouton gauche

L'utilisation de ces classes est très simple :

- Déterminez le `TransformGroup` sur lequel doit agir la souris et donnez lui les aptitudes de lecture et d' écriture ;
- Créez le `MouseBehavior` ;
- Faites référencer le `TransformGroup` par le `MouseBehavior` ;
- Définissez la zone d'influence (`Bounds`) du `MouseBehavior` ;
- Ajoutez le `MouseBehavior` à la scène.

Code à ajouter pour attacher la translation à un groupe de transformation

```
/*----- Ajout d'un comportement souris -----*/
this.tgVolume.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
this.tgVolume.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
MouseTranslate mouse = new MouseTranslate(this.tgVolume);
mouse.setSchedulingBounds(new   BoundingSphere(new   Point3d(),100.0));
this.racineVolume.addChild(mouse);
```

Vous pouvez ajouter plusieurs comportements sur un même `TransformGroup`.

7.5.2.- Manipulation à l'aide du clavier

L'API vous propose une classe `KeyNavigatorBehavior` permettant de naviguer dans la scène.
Pour cela :

- Créez un `KeyNavigatorBehavior` et associez le au `TransformGroup` de la branche de volume ;
- Définissez une zone d'influence (`Bounds`) ;
- Attachez le à la branche de volume.

Code à ajouter pour la navigation clavier

```
/*----- Ajout de la navigation à l'aide du clavier -----*/
KeyNavigatorBehavior key = new KeyNavigatorBehavior (this.universe.
                  getViewingPlatform().getViewPlatformTransform());
key.setSchedulingBounds(new BoundingSphere(new Point3d(),100.0));
this.racineVolume.addChild(key);
```

Mouvement	Action
vers la gauche	Flèche gauche
vers la droite	Flèche droite
vers le haut	pg up
vers le bas	pg down
vers l'avant	Flèche haut
vers l'arrière	flèche bas

Remarques :

- Si le `Canvas3D` (zone d'affichage de votre scène) est inséré dans un contenant (`JFrame` ou `JPanel`) qui implémente la classe `KeyListener` alors les évènements issus du clavier seront interceptés par le `Canvas3D`. Donc votre objet `KeyListener` sera inactif.

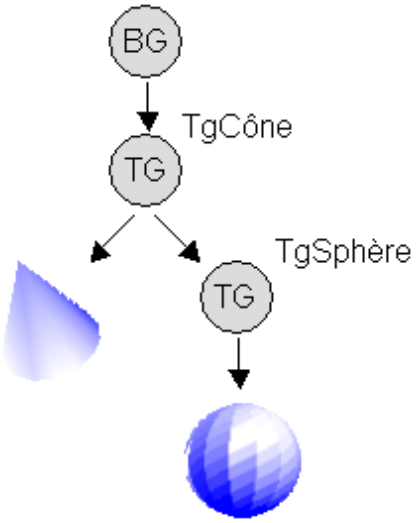
- Avant de tester vos comportements issus du clavier, pensez à donner le focus à votre `Canvas3D` en cliquant dessus.

7.5.3.- Cueillir un objet à l'aide de la souris

Java3D propose un paquetage définissant le comportement de picking (attraper, cueillir). Le picking permet de désigner un objet à l'aide de la souris. Ensuite l'utilisateur va pouvoir manipuler chaque objet de la scène et lui faire subir une modification de position ou d'orientation. Cela n'est pas possible avec les comportements de type `MouseBehavior` car on ne peut pas différencier les objets sur lesquels on souhaite agir.

7.5.3.1. - Fonctionnement du processus de picking

Lorsque l'utilisateur clique dans la fenêtre de visualisation de la scène (`Canvas3D`), un rayon est projeté dans la scène3D parallèlement aux axes de projection. Ensuite, le processus recherche l'objet intercepté par le rayon le plus proche de l'utilisateur. A partir de cet objet, le processus remonte dans le graphe de scène jusqu'à trouver un `TransformGroup` possédant la capacité `ENABLE_PICK_REPORTING`.



Par exemple : Dans le graphe ci-contre la sphère est positionnée par rapport au cône.

Si TgCône et TgSphère ont tous les deux la capacité ENABLE_PICK_REPORTING, quand l'utilisateur clique sur la sphère, celle-ci sera l'objet manipulée au travers du changement de la matrice de transformation de TgSphère.

Si seul TgCône a la capacité ENABLE_PICK_REPORTING, quand l'utilisateur clique sur la sphère alors les deux objets subissent des changements car les changements de position ou d'orientation s'appliqueront sur la matrice de transformation de TgCône.

7.5.3.2. - Les différents comportements de picking

Un objet attrappé peut subir les opérations de translation (`PickTranslateBehavior`), de rotation (`PickRotateBehavior`) et de zoom (`PickZoomBehavior`). Pour cela, il faut :

- Créez un objet de comportement de Picking ;
- Ajoutez le comportement à la scène ;
- Modifiez les capacités du ou des `TransformGroup` du graphe de scène. Un `TransformGroup` susceptible de subir des modifications à partir d'un comportement de picking devra avoir les trois aptitudes suivantes :
  - `ENABLE_PICK_REPORTING` : aptitude à être attrapé ;
  - `ALLOW_TRANSFORM_READ` : autorise la lecture de sa matrice de transformation ;
  - `ALLOW_TRANSFORM_WRITE` : autorise la mise à jour de sa matrice de transformation.

Vous pouvez ajouter plusieurs comportements de picking sur la même branche du graphe de scène.

Mouvement	Action
rotation	bouton gauche de la souris
translation	bouton droit de la souris
zoom	bouton du milieu ou ALT + bouton gauche

Constructeur :

```
PickTranslateBehavior(BranchGroup root, Canvas3D canvas, Bounds bounds)
```

Les `MouseBehavior` ont la priorité sur les `PickMouseBehavior`. Si vous attachez un `MouseTranslate` et un `PickTranslateBehavior` sur la même branche de volume, la translation issue du picking sera indisponible.

A.- Annexes

A.1.- C++ - Java : les principales différences

A.1.1.- Les pointeurs

Java n'a pas de type pointeur explicite. Les références aux objets (affectation de variables, paramètres des méthodes, éléments des tableaux …) sont implicites. Les références et les pointeurs diffèrent en ce sens qu'il n'y a pas de pointeur d'opérateur arithmétique sur les références.

A.1.2.- Les Tableaux

Les tableaux sont des classes d'objets. Leur contenu peut être atteint par des références explicites et non par pointeurs arithmétiques. Les limites des tableaux sont fixées (les débordements sont détectés à la compilation).
A l'inverse de C/C++, Java ne supporte pas les tableaux multi-dimensionnels.

A.1.3.- Les chaînes de caractères

En Java, les chaînes de caractères ne sont pas des tableaux de caractères comme en C/C++, mais des objets de la classe `String`. Tous les problèmes liés aux débordements ou au caractère de fin de chaîne (« \0 ») qui hantent les nuits des programmeurs C/C++ sont ainsi résolus.

A.1.4.- La gestion de la mémoire

La gestion de la mémoire est automatique : lors de la création d'un objet, de la mémoire lui est affectée. Un récupérateur de mémoire, le « garbage collector » est chargé de récupérer la mémoire quand l'objet n'est plus utilisé. Les fonctions malloc et free (ou delete[]) n'existe pas en Java. Pour « forcer » la libération de mémoire il faut supprimer les références à un objet (affecter le pointeur « null » aux objets …)

A.1.5.- Les types de données

Les types de données ont des tailles et des comportements adaptés aux différentes plates-formes et systèmes d'exploitation. Les types de données non signés n'existent pas en Java.

Un booléen n'est pas un entier en Java, il accepte deux valeurs : true et false.

Les types de données composés sont uniquement créés dans les définitions de classes. Les mots-clefs `struct`, `union` et `typedef` ont été enlevés au profit des classes.

La conversion des types est plus contrôlée en Java. Elle est automatique seulement s'il n'y a pas de perte d'information. Toutes les autres conversions sont explicites. La conversion des types de base en objet (ou vice versa) est impossible. Il existe des méthodes et des classes spéciales pour convertir les valeurs entre objets et types de base.

A.1.6.- Les arguments

A la différence de C/C++, Java ne supporte pas les mécanismes d'arguments optionnels ou de listes d'arguments à longueur variable. Toutes les définitions de méthode on obligatoirement un nombre déterminé d'arguments.

Les arguments des lignes de commande de Java diffèrent de celles de C/C++. En C/C++, le premier élément (argv[0]) est le nom du programme. En Java c'est le premier argument additionnel. Ainsi en Java, argv[0] correspond à argv[1] dans C/C++. Il n'est donc pas possible de récupérer le nom d'un programme Java.

A.1.7.- Autres différences

- Java n'a pas de préprocesseur. Il n'y a donc ni `#define`, ni macros.

  Dans Java, des constantes sont créées à l'aide du modificateur `final` lors de la déclaration des variables de classe et d'instance.

- Java n'a pas de classes modèles comme en C++.
- Java ne possède ni le mot-clé `const` de C, ni sa capacité à passer par la référence de `const` de manière explicite.
- Les classes de Java sont uniquement héritées avec plusieurs caractéristiques d'héritage fornies par les interfaces.
- Dans Java, toutes les fonctions sont implémentées en tant que méthodes. De plus, elles sont toutes liées aux classes.
- Le mot-clé `goto` n'existe pas dans Java.

A.2.- Problèmes avec les méthodes de la classe Thread

A.2.1. - Why is Thread.stop deprecated ?

Because it is inherently unsafe. Stopping a thread causes it to unlock all the monitors that it has locked. (The monitors are unlocked as the `ThreadDeath` exception propagates up the stack.) If any of the objects previously protected by these monitors were in an inconsistent state, other threads may now view these objects in an inconsistent state. Such objects are said to be *damaged*. When threads operate on damaged objects, arbitrary behavior can result. This behavior may be subtle and difficult to detect, or it may be pronounced. Unlike other unchecked exceptions, `ThreadDeath` kills threads silently; thus, the user has no warning that his program may be corrupted. The corruption can manifest itself at any time after the actual damage occurs, even hours or days in the future.

A.2.2. - What about Thread.stop(Throwable) ?

In addition to all of the problems noted above, this method may be used to generate exceptions that its target thread is unprepared to handle (including checked exceptions that the thread could not possibly throw, were it not for this method). For example, the following method is behaviorally identical to Java's `throw` operation, but circumvents the compiler's attempts to guarantee that the calling method has declared all of the checked exceptions that it may throw:

```
static void sneakyThrow(Throwable t) {
    Thread.currentThread().stop(t);
}
```

A.2.3. - What should I use instead of Thread.stop ?

Most uses of `stop` should be replaced by code that simply modifies some variable to indicate that the target thread should stop running. The target thread should check this variable regularly, and return from its run method in an orderly fashion if the variable indicates that it is to stop running. (This is the approach that JavaSoft's Tutorial has always recommended.) To ensure prompt communication of the stop-request, the variable must be `volatile` (or access to the variable must be synchronized).
For example, suppose your applet contains the following `start`, `stop` and `run` methods:

```
private Thread blinker;

public void start() {
    blinker = new Thread(this);
    blinker.start();
}

public void stop() {
    blinker.stop();  // UNSAFE!
}

public void run() {
    Thread thisThread = Thread.currentThread();
    while (true) {
        try {
            thisThread.sleep(interval);
        }
        catch (InterruptedException e) {
        }
        repaint();
    }
}
```

You can avoid the use of `Thread.stop` by replacing the applet's `stop` and `run` methods with:

```
    private volatile Thread blinker;

    public void stop() {
        blinker = null;
    }

    public void run() {
        Thread thisThread = Thread.currentThread();
        while (blinker == thisThread) {
            try {
                thisThread.sleep(interval);
            }
            catch (InterruptedException e){
            }
            repaint();
        }
    }
```

A.2.4. - Why are Thread.suspend and Thread.resume deprecated ?

`Thread.suspend` is inherently deadlock-prone. If the target thread holds a lock on the monitor protecting a critical system resource when it is suspended, no thread can access this resource until the target thread is resumed. If the thread that would resume the target thread attempts to lock this monitor prior to calling `resume`, deadlock results. Such deadlocks typically manifest themselves as "frozen" processes.

A.2.5. - What should I use instead of Thread.suspend and Thread.resume ?

As with `Thread.stop`, the prudent approach is to have the "target thread" poll a variable indicating the desired state of the thread (active or suspended). When the desired state is suspended, the thread waits using `Object.wait`. When the thread is resumed, the target thread is notified using `Object.notify`.
For example, suppose your applet contains the following mousePressed event handler, which toggles the state of a thread called `blinker`:

```
    private boolean threadSuspended;

    Public void mousePressed(MouseEvent e) {
        e.consume();

        if (threadSuspended)
            blinker.resume();
        else
            blinker.suspend();  // DEADLOCK-PRONE!

        threadSuspended = !threadSuspended;
    }
```

You can avoid the use of `Thread.suspend` and `Thread.resume` by replacing the event handler above with:

```
    public synchronized void mousePressed(MouseEvent e) {
        e.consume();
        threadSuspended =! threadSuspended;

        if (!threadSuspended)
            notify();
    }
```

and adding the following code to the "run loop" :

```
     synchronized(this) {
        while (threadSuspended)
             wait();
     }
```

The `wait` method throws the `InterruptedException`, so it must be inside a `try ... catch` clause. It's fine to put it in the same clause as the `sleep`. The check should follow (rather than precede) the `sleep` so the window is immediately repainted when the the thread is "resumed." The resulting `run` method follows:

```
    public void run() {
        while (true) {
            try {
                Thread.currentThread().sleep(interval);
                synchronized(this) {
                    while (threadSuspended)
                        wait();
                }
            }
            catch (InterruptedException e) {}
            repaint();
        }
    }
```

Note that the `notify` in the `mousePressed` method and the `wait` in the `run` method are inside `synchronized` blocks. This is required by the language, and ensures that `wait` and `notify` are properly serialized. In practical terms, this eliminates race conditions that could cause the "suspended" thread to miss a `notify` and remain suspended indefinitely.
While the cost of synchronization in Java is decreasing as the platform matures, it will never be free. A simple trick can be used to remove the synchronization that we've added to each iteration of the "run loop." The synchronized block that was added is replaced by a slightly more complex piece of code that enters a synchronized block only if the thread has actually been suspended :

```
     if (threadSuspended) {
        synchronized(this) {
           while (threadSuspended)
                wait();
        }
     }
```

The resulting `run` method is :

```
    public void run() {
        while (true) {
            try {
                Thread.currentThread().sleep(interval);

                if (threadSuspended) {
                    synchronized(this) {
                        while (threadSuspended)
                            wait();
                    }
                }
            } catch (InterruptedException e) {}
            repaint();
        }
    }
```

In the absence of explicit synchronization, `threadSuspended` must be made `volatile` to ensure prompt communication of the suspend-request.

A.3.- Quelques exemples

A.3.1.- Les Threads

Pour illustrer l'utilisation des threads, voici un exemple d'un chronomètre affichant les 1/10 de seconde : 0:00:00:0

Ci-dessous le programme Java correspondant :

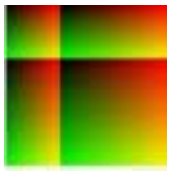
```
import java.awt.*;
import java.applet.Applet;

public class Chrono extends Applet implements Runnable
{
  private Thread chronometre;
  private int    dixiemeseconde = 0;

  // Méthode appelée par le système au démarrage de l'applet
  public void start ()
  {
    // Au début de l'applet, création et démarrage du chronomètre
    chronometre = new Thread (this);
    chronometre.start ();
  }

  public void run ()
  {
    try
    {
      while (chronometre.isAlive ())
      {
        // Dessine le compteur (appel indirect à la méthode paint ()),
        repaint ();
        dixiemeseconde++; // augmente le compteur de 1
        // Arrête le thread pendant 1/10 s (100 ms)
        Thread.sleep (100);
      }
    }
    catch (InterruptedException e) { }
  }

  // Méthode appelée par le système à l'arrêt de l'applet
  public void stop ()
  {
    // A la fin de l'applet, arrêt du chronometre
    chronometre.stop ();
  }

  // Méthode appelée par le système pour mettre à jour le dessin de l'applet
  public void paint (Graphics gc)
  {
    // Dessine le temps écoulé sous forme de hh:mm:ss:d en noir et helvetica
    gc.setColor (Color.black);
    gc.setFont  (new Font ("Helvetica", Font.BOLD, size ().height));
    gc.drawString (dixiemeseconde / 36000
            + ":" + (dixiemeseconde / 6000) % 6 + (dixiemeseconde / 600) % 10
            + ":" + (dixiemeseconde / 100) % 6  + (dixiemeseconde / 10) % 10
            + ":" + dixiemeseconde % 10, 2, size ().height - 2);
  }
}
```

Ce programme est une d'applet qui implémente l'interface `Runnable`. Comme décrit au chapitre sur les applets, la méthode `paint ()` de la classe `Applet` est appelée pour mettre à jour le dessin apparaissant dans la fenêtre d'une applet. Dans cet exemple, elle est surchargée pour dessiner le chronomètre.
Quand l'applet est créée, une instance de la classe `Chrono` est allouée et la méthode `start ()` créant le thread chronometre est appelée. Si on exécute cette applet, on se rend compte que le chronomètre à tendance à retarder. En effet, à chaque tour de boucle `while` (), le thread est arrêté pendant un dixième de seconde grâce à l'appel `Thread.sleep (100)` après le redessin de l'applet avec la méthode `repaint ()` dans `run ()`. Le fait de redessiner le chronomètre prend un faible délai qui s'additionne au 1/10 de seconde d'arrêt du thread chronomètre. Une programmation plus précise devrait notamment tenir compte de ce délai pour le soustraire de la valeur de 100 millisecondes passée à la méthode `sleep ()`. La classe `System` déclare une méthode `currentTimeMillis ()`, donnant le temps courant, qui peut aider à résoudre ce problème.

A.3.2.- Les images

Exemple de génération d'image

Les méthodes `drawImage ()` de la classe `Graphics` permettent d'afficher une image à un point donné en la redimensionnant éventuellement, comme le montre l'applet suivante, qui crée une image d'un nuancier, et l'affiche à 4 tailles différentes :



```
import java.applet.Applet;
import java.awt.*;

public class MultiImages extends Applet
{
  private Image image;

  public void reshape (int x, int y, int width, int height)
  {
     // Effectuer le comportement par défaut du changement de taille
     super.reshape (x, y, width, height);

     // Création de l'image que quand le peer de l'applet existe
     if (getPeer () != null)
     {
        Dimension taille = size ();
        taille.width  /= 3;
        taille.height /= 3;
        // Création d'une image de dimensions 3 fois plus petites que l'applet
        image = createImage (taille.width, taille.height);
        Graphics gc = image.getGraphics ();

        // Remplissage d'une image avec un nuancier de rouge et vert
        for (int i = 0; i < taille.height; i++) {
           for (int j = 0; j < taille.width; j++) {
              gc.setColor (new Color ((float)i / taille.height, // Rouge
                                      (float)j / taille.width,  // Vert
                                      0));                      // Bleu
              gc.fillRect (i, j, 1, 1);
           }
        }
     }
   } // fin reshape
```

```
  public void paint (Graphics gc)
  {
    if (image != null)
    {
      // Dessin de l'image à sa taille et agrandie
      gc.drawImage (image, 0, 0, this);
      gc.drawImage (image, image.getWidth (this), 0,
                           image.getWidth (this) * 2,
                           image.getHeight (this), this);
      gc.drawImage (image, 0, image.getHeight (this),
                           image.getWidth (this),
                           image.getHeight (this) * 2, this);
      gc.drawImage (image, image.getWidth (this), image.getHeight (this),
                           image.getWidth (this) * 2,
                           image.getHeight (this) * 2, this);
    }
  }
} // fin applet
```

Deux exemples de chargement d'image (avec et sans utilisation de `ImageObserver`)

L'applet suivante utilise la méthode `drawImage()` pour charger et afficher une image :

```
import java.applet.Applet;
import java.awt.*;

public class ImageSimple extends Applet
{
  private Image image;

  public void init ()
  {
    // Création d'une image
    image = getImage (getCodeBase (), "monval.jpg");
  }

  public void paint (Graphics gc)
  {
    if (image != null)
      // Affichage de l'image (image chargée automatiquement)
      gc.drawImage (image, 0, 0, this);
  }
}
```

L'applet suivante utilise l'interface `ImageObserver` pour attendre la fin du chargement d'une image et l'afficher :

```
import java.applet.Applet;
import java.awt.*;
import java.awt.image.*;

public class ChargementImage extends Applet implements ImageObserver
{
  private Image   image;
  private boolean chargementTermine = false;

  public void init ()
  {
    // Création d'une image et lancement de son chargement
    image = getImage (getCodeBase (), "brookbr.jpg");
    prepareImage (image, this);
  }
```

```
  public void paint (Graphics gc)
  {
    // Si le chargement de l'image est terminé, affichage de l'image
    // sinon affichage d'une chaîne de caractères d'attente
    if (chargementTermine)
      gc.drawImage (image, 0, 0, this);
    else
      gc.drawString ("Chargement en cours...", 10, size ().height - 10);
  }

  // Méthode appelée pour communiquer les étapes du chargement de l'image
  public boolean imageUpdate (Image image, int infoFlags,
                              int x, int y, int width, int height)
  {
    // Si le chargement est terminé, redessin de l'applet
    if ((infoFlags & ALLBITS) != 0)
    {
      chargementTermine = true;
      repaint ();
    }
    return (infoFlags & (ALLBITS | ABORT)) == 0;
  }
}
```

A la différence de l'applet ImageSimple, l'image de cette applet n'est affichée qu'une fois que l'image est entièrement chargée.

Exemple d'enchaînement d'images téléchargées

Cette applet utilise les 8 images de taille égale (ci-dessus). Elles sont extraites de l'image téléchargée grâce à l'utilisation de la classe de filtre `CropImageFilter`, puis un thread provoque leur affichage l'une après l'autre à intervalle régulier pour donner l'effet animé.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.image.*;

 public class AnimationFleche extends Applet implements Runnable
 {
  private Thread threadAnimation     = null;
  private Image  imagesAnimation [ ] = new Image [8];
  private int    imageCourante       = 0;

  public void init ()
  {
    try {
      // Création de l'image globale mémorisant 8 dessins de flèches
      Image multiImages = getImage (getCodeBase (), "fleches.gif");
      // Création d'un MediaTracker pour récupérer les 8 images
      MediaTracker imageTracker = new MediaTracker (this);
      for (int i = 0; i < imagesAnimation.length; i++)
      {
        // Chacune des 8 images est extraite de l'image principale
        imagesAnimation [i] = createImage (new FilteredImageSource
                                    (multiImages.getSource (),
                                     new CropImageFilter (i * 50, 0, 50, 50)));
        imageTracker.addImage (imagesAnimation [i], 0);
      }
```

```
      imageTracker.waitForID (0);
      // En cas d'erreur, déclenchement d'une exception
      if (imageTracker.isErrorAny ())
        throw new IllegalArgumentException ("Images non chargees");
    }
    catch (Exception e) { }
  }

  public void start ()
  {
    // Création et démarrage d'un thread d'animation
    threadAnimation = new Thread (this);
    threadAnimation.start ();
  }

  public void stop ()
  {
    threadAnimation.stop ();
  }

  public void run ()
  {
    while (threadAnimation.isAlive ())
      try
      {
        // Redessin de l'applet et passage à l'image suivante
        repaint ();
        imageCourante = ++imageCourante % 8;
        // Attente de 70 ms avant de passer à l'image suivante
        Thread.sleep (70);
      }
      catch (InterruptedException exception) { }
  }

  // Méthode outrepassée pour qu'elle dessine directement l'image
  public void update (Graphics gc)
  {
    paint (gc);
  }

  public void paint (Graphics gc)
  {
    // Dessin de l'image courante
    gc.drawImage (imagesAnimation [imageCourante], 0, 0, this);
  }
}
```