

JAVA 8

Les fondamentaux du langage Java

Ce livre s'adresse à tout informaticien désirant **développer sous Java**. Que le lecteur soit débutant ou qu'il ait déjà une première expérience avec un autre langage il trouvera dans cet ouvrage **toutes les bases nécessaires** pour se familiariser rapidement avec un des langages les plus utilisés au monde.

Les trois premiers chapitres présentent les bases du langage, de la **programmation objet** et les nouveautés de la version 8. Le lecteur découvrira notamment les nouvelles API **de gestion des dates**, les **expressions Lambda** et leur application dans la **gestion des collections**. Les chapitres suivants étudient le développement **d'applications graphiques** avec la **bibliothèque Swing** et la création **d'applets** permettant d'enrichir facilement le contenu de pages web. Le développement d'applications client/serveur est également présenté avec l'**API JDBC** assurant l'**accès aux bases de données**. Le déploiement étant une étape importante du succès d'une application, le dernier chapitre présente la **distribution** d'une application avec la solution classique des **fichiers d'archives sécurisés** ou l'utilisation plus souple de la technologie **Java Web Start**.

De **nombreux exercices avec leurs corrigés** vous permettront de valider vos connaissances et de mettre en pratique immédiatement les notions étudiées.

Le livre ne nécessite pas d'outils de développement spécifiques. Un éditeur de texte et les outils disponibles gratuitement sur le site d'Oracle sont suffisants pour mener à bien l'apprentissage de ce langage passionnant et en plein essor.

Des éléments complémentaires sont en téléchargement sur cette page.

Les chapitres du livre :

Avant-propos – Présentation – Bases du langage – Programmation objet – Applications graphiques – Les applets – Accès aux bases de données – Déploiement d'applications

Thierry GROUSSARD

Analyste et développeur pendant plus de 10 ans, **Thierry GROUSSARD** s'est ensuite orienté vers la formation et plus particulièrement dans le domaine du développement. Sa connaissance approfondie des besoins de l'entreprise et ses qualités pédagogiques rendent cet ouvrage particulièrement adapté à l'apprentissage et à la mise en pratique du développement sous Java.

Avant-propos

Lorsque les ingénieurs de Sun Microsystems ont développé en 1991 le langage Java, ils ne pensaient certainement pas que vingt ans plus tard il serait l'un des langages les plus utilisés au monde. À l'origine prévu pour le développement d'applications destinées à des systèmes embarqués, il est maintenant présent dans tous les domaines de l'informatique. Il apparaît comme un des langages les plus demandés dans la majorité des offres d'emploi dans le domaine du développement.

C'est un langage dont la syntaxe est simple mais rigoureuse. Il permet donc de prendre rapidement de bonnes habitudes lorsque l'on débute. C'est certainement pour cette raison qu'il est le langage le plus utilisé dans l'enseignement.

Le but de ce livre est de vous faire découvrir les bases de ce langage pour vous permettre ensuite d'évoluer vers le développement d'applications importantes utilisant les nombreuses technologies disponibles avec ce langage (JEE, JME...).

La lecture de ce livre ne nécessite pas de connaissances préalables en développement. Les chapitres Présentation et Bases du langage vous présentent les notions de base de tout langage informatique : les variables, les opérateurs, les conditions, les boucles...

Après l'apprentissage de ces bases, le chapitre Programmation objet vous présente les principes et la mise en œuvre de la programmation orientée objet (POO). Les notions présentées dans ce chapitre sont capitales pour pouvoir par la suite aborder la conception d'applications graphiques.

Les chapitres Applications graphiques et Les applets vous proposent d'étudier la conception d'applications graphiques autonomes avec la bibliothèque SWING, puis le développement d'applications s'exécutant dans le contexte d'un navigateur web avec la technologie des applets.

Vos futures applications nécessiteront certainement la manipulation d'informations présentes dans une base de données. Le chapitre Accès aux bases de données consacré à ce sujet vous apportera une aide précieuse pour mener à bien cette tâche. Il vous familiarisera avec l'utilisation de JDBC qui est la technologie utilisée par Java pour la gestion de l'accès à une base de données.

Le déploiement est bien sûr l'ultime étape de l'élaboration d'une application mais ne doit évidemment pas être négligé. Deux technologies de déploiement disponibles sont abordées dans le dernier chapitre de cet ouvrage, pour vous permettre de simplifier l'installation de vos applications sur les postes clients.

Cet ouvrage n'a toutefois pas pour vocation de se substituer à la documentation fournie par Oracle qui doit rester votre référence pour l'obtention d'informations telles que la liste des méthodes ou propriétés présentes dans une classe.

Historique

1. Pourquoi Java ?

Bill Joy, ingénieur chez Sun Microsystems, et son équipe de chercheurs travaillaient sur le projet "Green" qui consistait à développer des applications destinées à une large variété de périphériques et systèmes embarqués (notamment téléphones cellulaires et téléviseurs interactifs).

Convaincus par les avantages de la programmation orientée objet (POO), ils choisissaient de développer avec le langage C++ éprouvé pour ses performances.

Mais, par rapport à ce genre de projet, C++ a rapidement montré ses lacunes et ses limites. En effet, de nombreux problèmes d'incompatibilité se sont posés par rapport aux différentes architectures matérielles (processeurs, taille mémoire) et aux systèmes d'exploitation rencontrés, ainsi qu'au niveau de l'adaptation de l'interface graphique des applications et de l'interconnexion entre les différents appareils.

En raison des difficultés rencontrées avec C++, il était préférable de créer un nouveau langage autour d'une nouvelle plate-forme de développement. Deux développeurs de chez Sun, James Gosling et Patrick Naughton se sont attelés à cette tâche.

La création de ce langage et de cette plate-forme s'est inspirée des fonctionnalités intéressantes offertes par d'autres langages tels que C++, Eiffel, SmallTalk, Objective C, Cedar/ Mesa, Ada, Perl. Le résultat est une plate-forme et un langage idéaux pour le développement d'applications sécurisées, distribuées et portables sur de nombreux périphériques et systèmes embarqués interconnectés en réseau mais également sur Internet (clients légers), et sur des stations de travail (clients lourds).

D'abord surnommé C++-- (C++ sans ses défauts) puis OAK, mais il s'agissait d'un nom déjà utilisé dans le domaine informatique, il fut finalement baptisé Java, mot d'argot voulant dire café, en raison des quantités de café ingurgité par les programmeurs et notamment par ses concepteurs. Et ainsi, en 1991, est né le langage Java.

2. Objectifs de la conception de Java

Par rapport aux besoins exprimés, il fallait un langage et une plate-forme simples et performants, destinés au développement et au déploiement d'applications sécurisées, sur des systèmes hétérogènes dans un environnement distribué, devant consommer un minimum de ressources et fonctionner sur n'importe quelle plate-forme matérielle et logicielle.

La conception de Java a apporté une réponse efficace à ces besoins :

- Langage d'une syntaxe simple, orienté objet et interprété, permettant d'optimiser le temps et le cycle de développement (compilation et exécution).
- Les applications sont portables sans modification sur de nombreuses plates-formes matérielles et systèmes d'exploitation.
- Les applications sont robustes car la gestion de la mémoire est prise en charge par le moteur d'exécution de Java (*Java Runtime Environment*), et il est plus facile d'écrire des programmes sans erreur par rapport au C++, en raison d'un mécanisme de gestion des erreurs plus évolué et plus strict.
- Les applications et notamment les applications graphiques sont performantes en raison de la mise en œuvre et de la prise en charge du fonctionnement de multiples processus légers (thread et multithreading).
- Le fonctionnement des applications est sécurisé, notamment dans le cas d'applet Java où le moteur d'exécution de Java veille à ce qu'aucune manipulation ou opération dangereuse ne soit effectuée par l'applet.

3. Essor de Java

Malgré la création de Java, les développements du projet "Green" n'ont pas eu les retombées commerciales escomptées et le projet fut mis de côté.

À cette époque, l'émergence d'Internet et des architectures client/serveur hétérogènes et distribuées a apporté une certaine complexité au développement des applications.

Les caractéristiques de Java se trouvent alors également fort intéressantes pour ce type d'applications.

En effet :

- un programme Java étant peu encombrant, son téléchargement à partir d'un site Internet prend peu de temps.
- un programme Java est portable et peut donc être utilisé sans modification sous n'importe quelle plate-forme (Windows, Macintosh, Unix, Linux...).

Java se trouve alors un nouveau domaine d'application sur le réseau mondial Internet, ainsi que sur les réseaux locaux dans une architecture intranet et client/serveur distribuée.

Pour présenter au monde les possibilités de Java, deux programmeurs de SUN, Patrick Naughton et Jonathan Peayne ont créé et présenté au salon SunWorld en mai 1995 un navigateur Web entièrement programmé en Java du nom de HotJava. Celui-ci permet l'exécution de programmes Java, nommés applets, dans les pages au format HTML.

En août 1995, la société Netscape, très intéressée par les possibilités de Java, signe un accord avec Sun lui permettant d'intégrer Java et l'implémentation des applets dans son navigateur Web (Netscape Navigator). En janvier 1996, Netscape version 2 arrive sur le marché en intégrant la plate-forme Java.

C'est donc Internet qui a assuré la promotion de Java.

Fort de cette réussite, Sun décide de promouvoir Java auprès des programmeurs en mettant à disposition gratuitement sur son site Web dès novembre 1995, une plate-forme de développement dans une version bêta du nom de JDK 1.0 (*Java Development Kit*).

Peu après, Sun crée une filiale du nom de JavaSoft (<http://java.sun.com>), dont l'objectif est de continuer à développer le langage.

Depuis, Java n'a fait qu'évoluer très régulièrement pour donner un langage et une plate-forme très polyvalents et sophistiqués, et de grandes compagnies telles que Borland/Inprise, IBM, Oracle, pour ne citer qu'eux, ont misé très fortement sur Java.

Début 2009, IBM effectue une tentative de rachat de Sun. Aucun accord n'étant trouvé sur le montant de la transaction, le projet de rachat échoue. Peu de temps après, Oracle fait à son tour une proposition de rachat qui cette fois se concrétise.

Java est aujourd'hui le premier langage objet enseigné dans les écoles et universités en raison de sa rigueur et de sa richesse fonctionnelle.

La communauté des développeurs Java représente plusieurs millions de personnes et est plus importante en nombre que la communauté des développeurs C++ (pourtant une référence).

Caractéristiques de Java

Java est à la fois un langage et une plate-forme de développement.

Cette partie vous présente ces deux aspects, elle vous donnera un aperçu des caractéristiques de Java et vous aidera à évaluer l'importance de l'intérêt porté à Java.

1. Le langage de programmation Java

Sun caractérise Java par le fait qu'il est simple, orienté objet, distribué, interprété, robuste, sécurisé, indépendant des architectures, portable, performant, multithread et dynamique.

Ces caractéristiques sont issues du livre blanc écrit en mai 1996 par James Gosling et Henry Mc Gilton et disponible à l'adresse suivante : <http://www.oracle.com/technetwork/java/langenv-140151.html>

Nous allons détailler chacune de ces caractéristiques.

a. Simple

La syntaxe de Java est similaire à celle du langage C et C++, mais elle omet des caractéristiques sémantiques qui rendent C et C++ complexes, confus et non sécurisés :

- En Java, il y a seulement trois types primitifs : les numériques (entiers et réels), le type caractère et le type booléen. Les numériques sont tous signés.
- En Java, les tableaux et les chaînes de caractères sont des objets, ce qui en facilite la création et la manipulation.
- En Java, le programmeur n'a pas à s'occuper de la gestion de la mémoire. Un système nommé le "ramasse-miettes" (*garbage collector*) s'occupe d'allouer la mémoire nécessaire lors de la création des objets et de la libérer lorsque les objets ne sont plus référencés dans le contexte courant du programme (quand aucune variable n'y fait référence).
- En Java, pas de préprocesseur et pas de fichier d'en-tête. Les instructions define du C sont remplacées par des constantes en Java et les instructions typedef du C sont remplacées par des classes en Java.
- En C et C++, on définit des structures et des unions pour représenter des types de données complexes. En Java, on crée des classes avec des variables d'instance pour représenter des types de données complexes.
- En C++ , une classe peut hériter de plusieurs autres classes, ce qui peut poser des problèmes d'ambiguïté. Afin d'éviter ces problèmes, Java n'autorise que l'héritage simple mais apporte un mécanisme de simulation d'héritage multiple par l'implémentation d'une ou de plusieurs interfaces.
- En Java, il n'existe pas la célèbre instruction goto, tout simplement parce qu'elle apporte une complexité à la lecture des programmes et que bien souvent, on peut se passer de cette instruction en écrivant du code plus propre. De plus, en C et C++, le goto est généralement utilisé pour sortir de boucles imbriquées. En Java, nous utiliserons les instructions break et continue qui permettent de sortir d'un ou plusieurs niveaux d'imbrication.
- En Java, il n'est pas possible de surcharger les opérateurs, tout simplement pour éviter des problèmes d'incompréhension du programme. On préférera créer des classes avec des méthodes et des variables d'instance.
- Et pour finir, en Java, il n'y a pas de pointeurs mais plutôt des références sur des objets ou des cases d'un tableau (référencées par leur indice), tout simplement parce qu'il s'est avéré que la manipulation des pointeurs est une grosse source de bugs dans les programmes C et C++.

b. Orienté objet

Mis à part les types de données primitifs, tout est objet en Java. Et de plus, si besoin est, il est possible d'encapsuler les types primitifs dans des objets, des classes préfabriquées sont déjà prévues à cet effet.

Java est donc un langage de programmation orienté objet conçu sur le modèle d'autres langages (C++, Eiffel, SmallTalk, Objective C, Cedar/Mesa, Ada, Perl), mais sans leurs défauts.

Les avantages de la programmation objet sont : une meilleure maîtrise de la complexité (diviser un problème complexe en une suite de petits problèmes), un réemploi plus facile, une meilleure facilité de correction et d'évolution.

Java est fourni de base avec un ensemble de classes qui permettent de créer et manipuler toutes sortes d'objets (interface graphique, accès au réseau, gestion des entrées/sorties...).

c. Distribué

Java implémente les protocoles réseau standard, ce qui permet de développer des applications client/serveur en architecture distribuée, afin d'invoquer des traitements et/ou de récupérer des données sur des machines distantes.

Pour cela, Java fournit de base deux API permettant de créer des applications client/serveur distribuées :

- RMI (*Remote Method Invocation*), qui permet de faire communiquer des objets Java s'exécutant sur différentes machines virtuelles Java et même sur différentes machines physiques.
- CORBA (*Common Object Request Broker Architecture*), basé sur le travail de l'OMG (<http://www.omg.org>), qui permet de faire communiquer des objets Java, C++ , Lisp, Python, Smalltalk, COBOL, Ada, s'exécutant sur différentes machines physiques.

d. Interprétré

Un programme Java n'est pas exécuté, il est interprété par la machine virtuelle ou JVM (*Java Virtual Machine*), ce qui le rend un peu plus lent. Mais cela apporte des avantages, notamment celui de ne pas être obligé de recompiler un programme Java d'un système à un autre car il suffit, pour chacun des systèmes, de posséder sa propre machine virtuelle Java.

Du fait que Java est un langage interprétré, vous n'avez pas à faire l'édition des liens (obligatoire en C++) avant d'exécuter un programme. En Java, il n'y a donc que deux étapes : la compilation puis l'exécution. L'opération d'édition des liens est réalisée par la machine virtuelle au moment de l'exécution du programme.

e. Robuste

Java est un langage fortement typé et très strict. Par exemple, la déclaration des variables doit obligatoirement être explicite en Java.

Le code est vérifié (syntaxe, types) à la compilation et également au moment de l'exécution, ce qui permet de réduire les bugs et les problèmes d'incompatibilité de versions.

De plus, la gestion des pointeurs est entièrement prise en charge par Java et le programmeur n'a aucun moyen d'y accéder, ce qui évite des écrasements inopportunus de données en mémoire et la manipulation de données corrompues.

f. Sécurisé

Vu les domaines d'application de Java, il est très important qu'il y ait un mécanisme qui veille à la sécurité des applications et des systèmes. C'est le moteur d'exécution de Java (JRE) qui s'occupe entre autres de cette tâche.

Le JRE s'appuie notamment sur le fichier texte `java.policy` qui contient des informations sur le paramétrage de la sécurité.

En Java, c'est le JRE qui gère la planification mémoire des objets et non le compilateur comme c'est le cas en C++.

Comme en Java il n'y a pas de pointeurs mais des références sur des objets, le code compilé contient des identifiants sur les objets qui sont ensuite traduits en adresses mémoire par le JRE, cette partie étant complètement opaque pour les développeurs.

Au moment de l'exécution d'un programme Java, le JRE utilise un processus nommé le ClassLoader qui s'occupe du chargement du bytecode (ou langage binaire intermédiaire) contenu dans les classes Java. Le bytecode est ensuite analysé afin de contrôler qu'il n'a pas fait de création ou de manipulation de pointeurs en mémoire et également qu'il n'y a pas de violation d'accès.

Comme Java est un langage distribué, les principaux protocoles d'accès au réseau sont implémentés (FTP, HTTP, Telnet...). Le JRE peut donc être paramétré afin de contrôler l'accès au réseau de vos applications :

- Interdire tous les accès.
- Autoriser l'accès seulement à la machine hôte d'où provient le code de l'application. C'est le cas par défaut pour les applets Java.
- Autoriser l'accès à des machines sur le réseau externe (au-delà du firewall), dans le cas où le code de l'application provient également d'un hôte sur le réseau externe.
- Autoriser tous les accès. C'est le cas par défaut pour les applications de type client lourd.

g. Indépendant des architectures

Le compilateur Java ne produit pas de code spécifique pour un type d'architecture.

En fait, le compilateur produit du bytecode (langage binaire intermédiaire) qui est indépendant de toute architecture matérielle, de tout système d'exploitation et de tout dispositif de gestion de l'interface utilisateur graphique (GUI).

L'avantage de ce bytecode est qu'il peut facilement être interprété ou transformé dynamiquement en code natif pour des besoins de performance.

Il suffit de disposer de la machine virtuelle dédiée à sa plate-forme pour faire fonctionner un programme Java. C'est elle qui s'occupe de traduire le bytecode en code natif.

h. Portable

Ce qui fait tout d'abord que Java est portable, c'est qu'il s'agit d'un langage interprétré.

De plus, contrairement au langage C et C++, les types de données primaires (numériques, caractère et booléen) de Java ont la même taille, quelle que soit la plate-forme sur laquelle le code s'exécute.

Les bibliothèques de classes standard de Java facilitent l'écriture du code qui peut ensuite être déployé sur différentes plates-formes sans adaptation.

i. Performant

Même si un programme Java est interprétré, ce qui est plus lent qu'un programme natif, Java met en œuvre un processus d'optimisation de l'interprétation du code, appelé JIT (*Just In Time*) ou HotSpot, qui permet de compiler à la volée le bytecode Java en code natif, ce qui permet d'atteindre les mêmes performances qu'un programme écrit en langage C ou C++.

j. Multitâche

Java permet de développer des applications mettant en œuvre l'exécution simultanée de plusieurs threads (ou processus légers). Ceci permet d'effectuer plusieurs traitements simultanément, afin

d'accroître la rapidité des applications, soit en partageant le temps CPU, soit en partageant les traitements entre plusieurs processeurs.

k. Dynamique

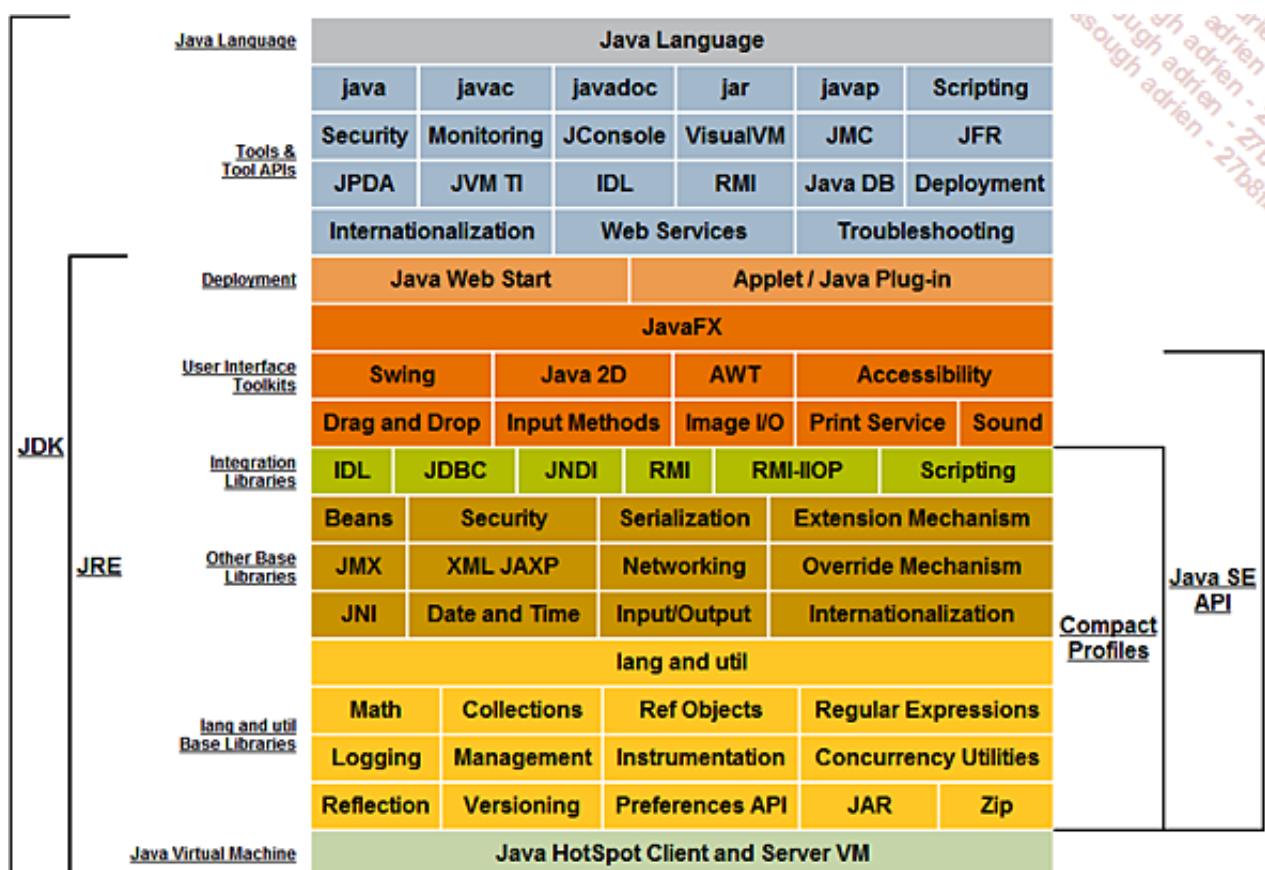
En Java, nous l'avons dit, le programmeur n'a pas à faire l'édition des liens (obligatoire en C et C++). Il est donc possible de modifier une ou plusieurs classes sans avoir à effectuer une mise à jour de ces modifications pour l'ensemble du programme. La vérification de l'existence des classes se fait au moment de la compilation et l'appel du code de ces classes ne se fait qu'au moment de l'exécution du programme. Ce procédé permet de disposer d'applications allégées en taille mémoire.

2. La plate-forme Java

Par définition, une plate-forme est un environnement matériel ou logiciel sur lequel peut s'exécuter un programme. La plupart des plates-formes actuelles sont la combinaison d'une machine et d'un système d'exploitation (ex : PC + Windows).

La plate-forme Java diffère par le fait qu'elle ne se compose que d'une partie logicielle qui s'exécute sur de nombreuses plates-formes matérielles et différents systèmes d'exploitation.

Le schéma suivant est issu du site web d'Oracle sur le langage Java et présente les différents composants de la plate-forme Java :



Comme le montre le schéma, elle est composée des éléments suivants :

- la machine virtuelle Java (JVM),
- l'interface de programmation d'application Java (API Java), qui est décomposée en trois catégories (API de bases, API d'accès aux données et d'intégration avec l'existant, API de gestion de l'interface avec l'utilisateur),
- les outils de déploiement des applications,

- les outils d'aide au développement.

Voyons en détail ces différents éléments.

a. La machine virtuelle Java (JVM)

La machine virtuelle est la base de la plate-forme Java, elle est nécessaire pour l'exécution des programmes Java. La JVM est disponible pour de nombreux types d'ordinateurs et systèmes d'exploitation.

La machine virtuelle Java s'occupe :

- du chargement des classes et du bytecode qu'elles contiennent : quand un programme invoque la création d'objets ou invoque des membres d'une classe, c'est la JVM qui s'occupe du chargement du bytecode qui doit être interprété.
- de la gestion de la mémoire : la JVM s'occupe entièrement de la gestion des pointeurs et donc de chaque référence faite à un objet. Ce procédé permet également à la JVM de s'occuper de la libération automatique de la mémoire (ramasse-miettes) dès qu'un objet n'est plus référencé dans le programme, c'est-à-dire quand aucune variable n'y fait référence.
- de la sécurité : c'est l'une des opérations les plus complexes effectuées par la JVM. Au chargement du programme, elle vérifie qu'il n'est pas fait appel à de la mémoire non initialisée, que des conversions de types illégales ne sont pas effectuées, que le programme ne manipule pas des pointeurs en mémoire. Dans le cas d'applets Java, la JVM interdit au programme l'accès aux périphériques de la machine sur laquelle l'applet s'exécute et autorise l'accès au réseau uniquement vers l'hôte qui diffuse l'applet.
- de l'interfaçage avec du code natif (par exemple, code écrit en langage C) : la plupart des API de base de Java font appel à du code natif qui est fourni avec le JRE, afin d'interagir avec le système hôte. Vous pouvez également utiliser ce procédé pour des accès à des périphériques ou à des fonctionnalités qui ne sont pas implémentées directement ou voir pas du tout en Java.

Le fait que Java soit interprété apporte des avantages et des inconvénients. Depuis toujours, on reproche à Java d'être moins performant que des langages natifs, ce qui était surtout le cas pour les applications avec interface utilisateur graphique. Afin de combler cette lacune et de perdre cette mauvaise image injustifiée, les développeurs de chez Oracle ont énormément travaillé sur l'optimisation de la JVM.

Avec la version 1.2, on avait un compilateur JIT (*Just In Time*) qui permettait d'optimiser l'interprétation du bytecode en modifiant sa structure pour le rapprocher du code natif. Depuis la version 1.3, la JVM intègre un processus nommé HotSpot (client et serveur) qui optimise davantage l'interprétation du code et d'une manière générale les performances de la JVM. HotSpot apporte un gain de performance allant de 30 % à 40 % selon le type d'application (on le remarque énormément au niveau des interfaces graphiques).

b. L'API Java

L'API Java contient une collection de composants logiciels préfabriqués qui fournissent un grand nombre de fonctionnalités.

L'API Java dans sa version 8 est organisée en plus de 220 packages, l'équivalent des librairies en langage C. Chaque package contient les classes et interfaces préfabriquées et directement réutilisables. Vous avez donc à votre disposition environ 4300 classes et interfaces.

La plate-forme Java fournit des API de base. De nombreuses extensions peuvent être ajoutées et sont disponibles sur le site Java d'Oracle : gestion des images en 3D, des ports de communication de l'ordinateur, de la téléphonie, des courriers électroniques...

L'API Java peut être décomposée en trois catégories :

Les API de base

Les API de base permettent de gérer :

- les éléments essentiels comme les objets, les chaînes de caractères, les nombres, les entrées/sorties, les structures et collections de données, les propriétés système, la date et l'heure, et plus encore...
- les applets Java dans l'environnement du navigateur Web.
- le réseau, avec les protocoles standard tels que FTP, HTTP, UDP, TCP/IP plus les URL et la manipulation des sockets.
- l'internationalisation et l'adaptation des programmes Java, en externalisant les chaînes de caractères contenues dans le code dans des fichiers de propriétés (.properties). Ce procédé permet d'adapter le fonctionnement des applications par rapport à des paramètres changeants (nom serveur, nom d'utilisateur, mot de passe...) et d'adapter la langue utilisée dans les interfaces graphiques par rapport aux paramètres régionaux de la machine.
- l'interfaçage avec du code natif, en permettant de déclarer que l'implémentation d'une méthode est faite au sein d'une fonction d'une DLL par exemple.
- la sécurité, en permettant :
 - de crypter/décrypter les données (*JCE - Java Cryptography Extension*),
 - de mettre en œuvre une communication sécurisée via SSL et TLS (*JSSE - Java Secure Socket Extension*),
 - d'authentifier et de gérer les autorisations des utilisateurs dans les applications (*JAAS - Java Authentication and Authorization Service*),
 - d'échanger des messages en toute sécurité entre des applications communiquant via un service comme Kerberos (*GSS-API - Generic Security Service - Application Programming Interface*),
 - de créer et valider des listes de certificats nommées Certification Paths (*Java Certification Path API*).
- la création de composants logiciels du nom de JavaBeans réutilisables et capables de communiquer avec d'autres architectures de composants tels que ActiveX, OpenDoc, LiveConnect.
- la manipulation de données XML (*eXtensible Markup Language*) à l'aide des API DOM (*Document Object Model*) et SAX (*Simple API for XML*). Les API de base permettent aussi d'appliquer des transformations XSLT (*eXtensible Style Sheet Transformation*) à partir de feuilles de style XSL sur des données XML.
- la génération de fichiers de journalisation (logs) permettant d'avoir un compte rendu du fonctionnement des applications (activité, erreurs, bugs...).
- la manipulation de chaînes de caractères avec des expressions régulières.
- les erreurs système et applicatives avec le mécanisme des exceptions chaînées.
- les préférences utilisateur ou système, en permettant aux applications de stocker et récupérer des données de configuration dans différents formats.

Les API d'accès aux données et d'intégration avec l'existant

Les API d'intégration permettent de gérer :

- des applications client/serveur dans une architecture distribuée, en permettant la communication en local ou par le réseau entre des objets Java fonctionnant dans des contextes de JVM différents, grâce à l'API RMI (*Remote Method Invocation*).
- des applications client/serveur dans une architecture distribuée, en permettant la communication en local ou par le réseau entre des objets Java et des objets compatibles CORBA tels que C++, Lisp, Python, Smalltalk, COBOL, Ada, grâce au support de l'API CORBA (*Common Object Request*

Broker Architecture), basé sur le travail de l'OMG (<http://www.omg.org>).

- l'accès à pratiquement 100 % des bases de données, via l'API JDBC (*Java DataBase Connectivity*).
- l'accès aux données stockées dans des services d'annuaire au protocole LDAP (*Lightweight Directory Access Protocol*) comme par exemple l'Active Directory de Windows, via l'API JNDI (*Java Naming and Directory Interface*).

Les API de gestion de l'interface des applications avec l'utilisateur

Les API de gestion de l'interface utilisateur permettent de gérer :

- la conception des interfaces graphiques avec l'API AWT (*Abstract Window Toolkit*) d'ancienne génération, ou l'API SWING de nouvelle génération.
- le son, avec la manipulation, la lecture et la création de fichiers son de différents formats (.wav ou .midi).
- la saisie de données textuelles par d'autres moyens que le clavier, comme par exemple des mécanismes de reconnaissance vocale ou de reconnaissance d'écriture, avec l'API Input Method Framework.
- les opérations graphiques de dessin avec l'API Java 2D et de manipulation d'images avec l'API Java Image I/O.
- l'accessibilité des applications aux personnes handicapées avec l'API Java Accessibility qui permet de s'interfacer par exemple avec des systèmes de reconnaissance vocale ou des terminaux en braille.
- le déplacement ou transfert de données lors d'une opération glisser/déposer (*Drag and Drop*).
- des travaux d'impression de données sur tout périphérique d'impression.

c. Les outils de déploiement des applications

La plate-forme Java fournit deux outils permettant d'aider au déploiement des applications :

- Java Web Start : destiné à simplifier le déploiement et l'installation des applications Java autonomes. Les applications sont disponibles sur un serveur, les utilisateurs peuvent en lancer l'installation sur leur machine via la console Java Web Start et tout se fait alors automatiquement. Ce qui est intéressant, c'est qu'ensuite à chaque lancement d'une application, Java Web Start vérifie si une mise à jour est disponible sur le serveur et procède automatiquement à son installation.
- Java Plug-in : destiné à permettre le fonctionnement des applets Java avec la machine virtuelle 8. En effet, lorsque vous accédez, via votre navigateur web, à une page html qui contient une applet, c'est la machine virtuelle du navigateur qui est chargée de la faire fonctionner. Le problème, c'est que les machines virtuelles des navigateurs supportent d'anciennes versions de Java. Afin de ne pas être limité au niveau des fonctionnalités et donc de ne pas rencontrer des problèmes d'incompatibilité entre les navigateurs, vous pouvez installer le Java Plug-in sur les postes clients. Le Java Plug-in consiste à installer un moteur d'exécution Java 8 (le JRE étant composé d'une JVM et de l'ensemble des API) et à faire en sorte que les navigateurs Web utilisent cette JRE et non la leur.

d. Les outils d'aide au développement

La plupart des outils d'aide au développement sont contenus dans le répertoire bin sous le répertoire racine de l'installation du J2SE.

Les principaux outils d'aide au développement permettent :

- de compiler (javac.exe) vos codes source .java en fichier .class.
- de générer la documentation (javadoc.exe) automatique de vos codes source (nom de classe,

package, hiérarchie d'héritage, liste des variables et méthodes) avec le même style de présentation que la documentation officielle des API standard fournies par Sun.

- de lancer l'exécution (java.exe) des applications autonomes Java.
- de visualiser, à l'aide d'une visionneuse (appletviewer.exe), l'exécution d'une applet Java dans une page HTML.

Deux autres technologies sont également intéressantes. Elles sont destinées à des outils de développement tiers afin qu'ils puissent les intégrer :

- JPDA (*Java Platform Debugger Architecture*), qui permet d'intégrer un outil de débogage au sein de son IDE de développement, apportant des fonctionnalités telles que les points d'arrêts, le pas à pas, l'inspection des variables et expressions...
- JVMPPI (*Java Virtual Machine Profiler Interface*), qui permet d'effectuer des analyses et de générer des états sur le fonctionnement des applications (mémoire utilisée, objets créés, nombre et fréquence d'invocation des méthodes, temps de traitement...) afin d'observer le bon fonctionnement des applications et de repérer où sont les "goulets d'étranglement".

3. Cycle de conception d'un programme Java

Pour développer une application Java, il faut d'abord se procurer la plate-forme J2SE de développement (SDK - *Software Development Kit*) dédiée à sa machine et à son système d'exploitation, dont vous trouverez la liste sur le site Java d'Oracle : <http://www.oracle.com/technetwork/java/index.html>

Ensuite, vous pouvez utiliser les API standard de Java pour écrire vos codes sources. En Java, la structure de base d'un programme est la classe et chaque classe doit être contenue dans un fichier portant l'extension java. Plusieurs classes peuvent être contenues dans un même fichier .java, mais une seule de ces classes peut être déclarée publique. Et c'est le nom de cette classe déclarée publique qui donne son nom au fichier .java.

Au cours du développement, vous pouvez procéder à la phase de compilation en utilisant l'outil javac.exe. Vous obtenez comme résultat au moins un fichier portant le même nom mais avec l'extension .class. Le fichier .class compilé reste tout de même indépendant de toute plate-forme ou système d'exploitation.

Ensuite, c'est l'interpréteur (java.exe) qui exécute les programmes Java. Pour l'exécution des applets, l'interpréteur est incorporé au navigateur Internet compatible Java. Pour l'exécution d'applications Java autonomes, il est nécessaire de lancer l'exécution de la machine virtuelle fournie soit avec la plate-forme de développement Java (SDK), soit avec le kit de déploiement d'applications Java (JRE - *Java Runtime Environment*).

Installation du SDK version Win32 pour l'environnement Windows

1. Téléchargement

Dans un premier temps, il vous faut télécharger la dernière version du SDK pour l'environnement Windows (Win32) à partir du site web d'Oracle :<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

Actuellement, le fichier à télécharger se nomme jdk-8u5-windows-i586.exe et fait 152 Mo. Dans tous les cas, téléchargez toujours la dernière version disponible.

Puisque vous êtes sur le site web d'Oracle, profitez-en pour télécharger un autre élément qui s'avère indispensable pour programmer en Java : la documentation des API standard.

Actuellement, le fichier à télécharger se nomme jdk-8u5-apidocs.zip et fait 85 Mo. Pour pouvoir le décompresser sur votre machine, il vous faut 300 Mo d'espace disque disponible. Cela fait beaucoup de lecture !

2. Installation

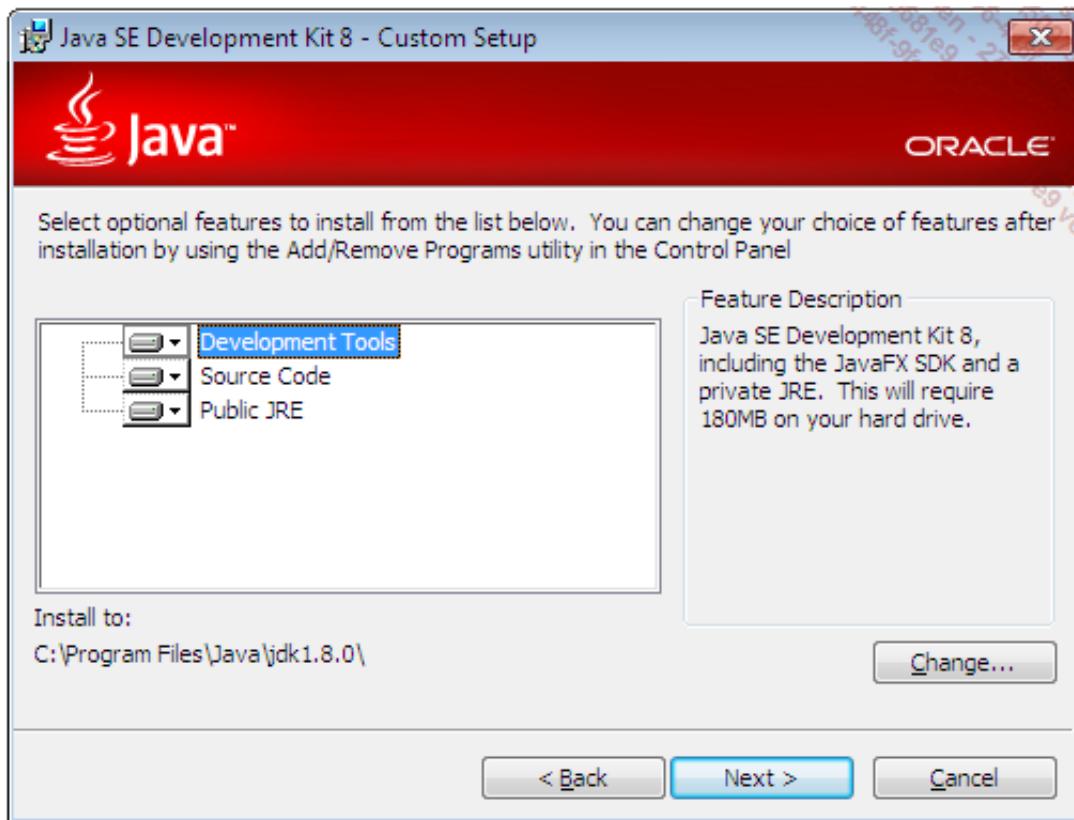
Avant d'installer le SDK sur votre ordinateur, assurez-vous qu'il n'y a aucun autre outil de développement Java d'installé, ceci afin d'éviter les problèmes de conflits de configuration.

- Pour commencer l'installation, double cliquez sur le fichier d'installation précédemment téléchargé : jdk-8u5-windows-i586.exe.

Tout d'abord une boîte de message **Welcome** apparaît vous indiquant que vous êtes sur le point d'installer le SDK et vous demande si vous voulez poursuivre l'installation.

- Cliquez sur **Next**.

Une nouvelle fenêtre apparaît, **Custom Setup** qui vous permet de sélectionner les éléments du SDK à installer et le répertoire de destination de l'installation.



- Après avoir fait vos choix ou avoir laissé la sélection par défaut, cliquez sur **Next**. Le programme installe alors les fichiers sur votre ordinateur. Après quelques instants, la boîte de dialogue suivante vous informe sur le succès de l'installation.



3. Configuration

Il reste à configurer le système, en indiquant dans quel répertoire sont stockés les outils tels que java.exe

(machine virtuelle) appletviewer.exe (visionneuse d'applets) ou encore javac.exe (compilateur). Pour ce faire, il faut modifier la variable d'environnement PATH pour ajouter le chemin d'accès vers le répertoire bin du jdk. Si vous avez conservé les options par défaut lors de l'installation, ce chemin doit être C:\Program Files\Java\jdk1.8.0\bin.

4. Test de la configuration du SDK

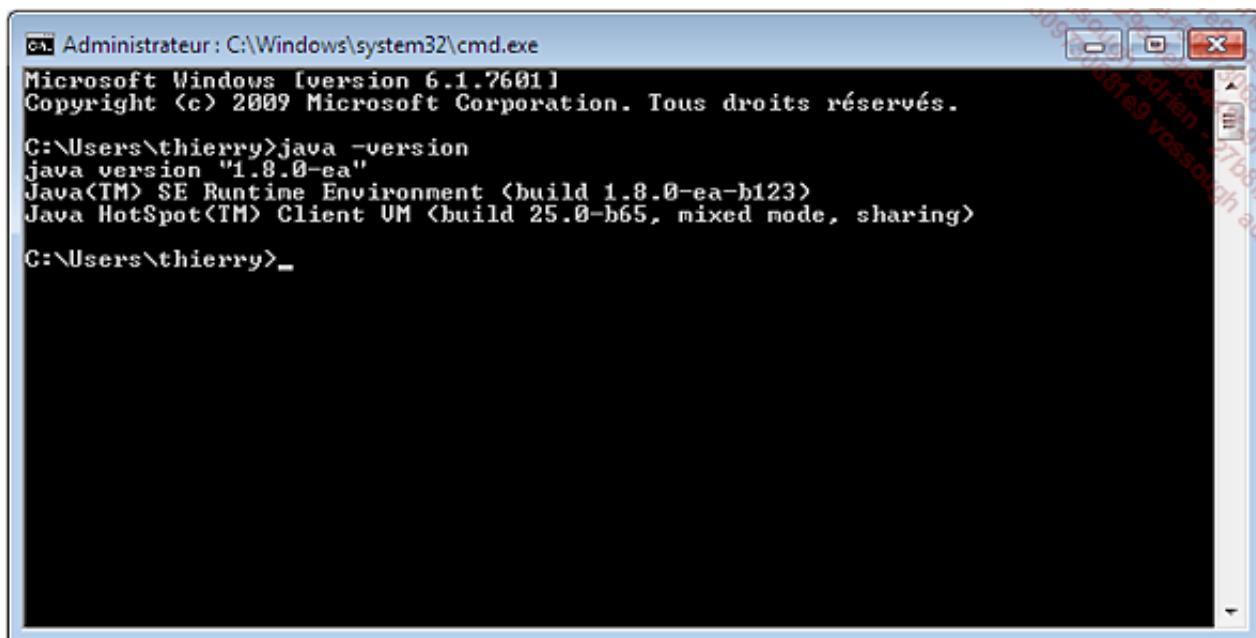
Vous allez tester si l'ordinateur a bien pris en compte les modifications que vous venez d'apporter à la variable PATH et donc vérifier s'il trouve le chemin où sont situés les outils du SDK.

Pour tester la configuration du SDK, utilisez une fenêtre **Invite de commandes**.

- À l'invite de commandes, saisissez la commande suivante qui va permettre de déterminer si l'installation du SDK est correcte ou non :

```
java -version
```

Vous devez voir le message suivant apparaître en réponse à la ligne que vous avez saisie :



The screenshot shows a Windows Command Prompt window titled "Administrator: C:\Windows\system32\cmd.exe". The window displays the following text:

```
Microsoft Windows [version 6.1.7601]
Copyright © 2009 Microsoft Corporation. Tous droits réservés.

C:\Users\thierry>java -version
java version "1.8.0-ea"
Java(TM) SE Runtime Environment (build 1.8.0-ea-b123)
Java HotSpot(TM) Client VM (build 25.0-b65, mixed mode, sharing)

C:\Users\thierry>_
```

Cette commande affiche des informations concernant la version de la machine virtuelle Java.

Si vous obtenez un message du style : 'java' n'est pas reconnu en tant que commande interne ou externe, un programme exécutable ou un fichier de commandes, cela signifie que le répertoire où sont stockés les outils du SDK n'a pas été trouvé par votre système.

Dans ce cas, vérifiez si la variable PATH contient bien les modifications que vous avez apportées et que vous n'avez pas fait d'erreur de syntaxe en spécifiant le chemin du répertoire bin.

5. Installation de la documentation du SDK et des API standard

À l'aide d'un utilitaire de décompression tel que WinZip, ouvrez le fichier que vous avez précédemment téléchargé. Extrayez tous les fichiers qu'il contient vers la racine d'installation du SDK, c'est-à-dire par défaut sous : C:\Program Files\Java\jdk1.8.0

Il faut prévoir 270 Mo d'espace disque disponible pour installer la documentation.

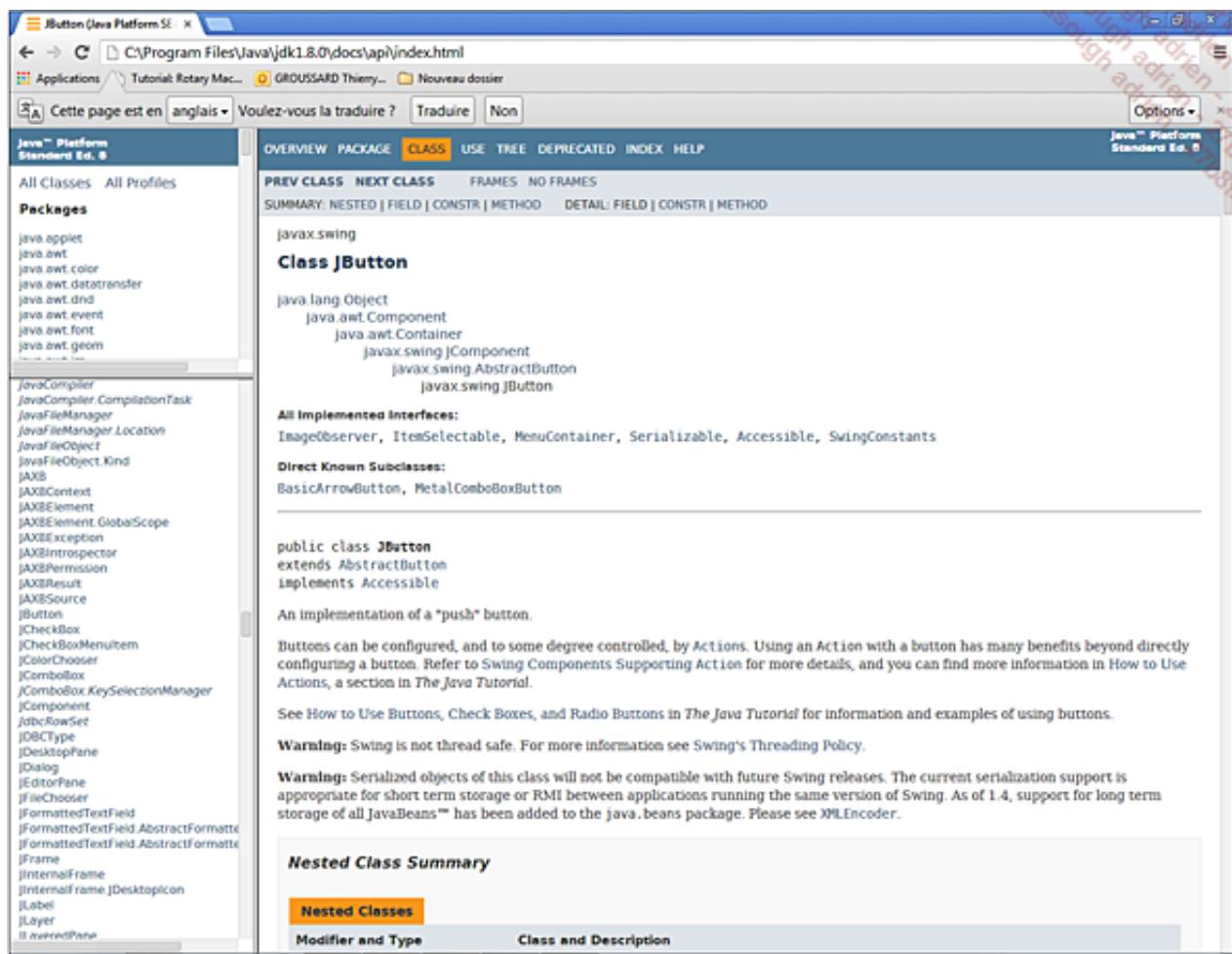
Une fois tous les fichiers extraits, fermez l'utilitaire. Sous l'explorateur Windows, dans le

répertoire C:\Program Files\Java\jdk1.8.0, vous devez avoir un nouveau répertoire docs. C'est le répertoire qui contient l'ensemble de la documentation du SDK au format HTML.

Dans ce répertoire docs, double cliquez sur le fichier index.html. Ce fichier contient des liens hypertextes vers l'ensemble de la documentation Java, qui est soit installée sur votre ordinateur, soit accessible sur un site Web.

Le plus important de la documentation se trouve dans le sous-répertoire api, en double cliquant sur le fichier index.html. Ce fichier contient les spécifications de l'API Java, ou plus précisément la description de l'ensemble des classes de la librairie Java. Sans cette documentation, vous ne pourrez pas développer efficacement en Java.

Il est recommandé de placer sur votre bureau un raccourci vers ce document.



Cette page est organisée en trois fenêtres :

- la fenêtre en haut à gauche contient la liste des packages (plus de 220).
- la fenêtre en bas à gauche contient la liste des classes contenues dans le package sélectionné dans la fenêtre précédente.
- la plus grande fenêtre contient la description d'une interface ou d'une classe sélectionnée dans la fenêtre précédente. La description d'une classe est organisée de la manière suivante :
 - un schéma de la hiérarchie des super-classes de l'interface ou de la classe en cours.
 - une explication sur l'utilisation de la classe ou de l'interface.
 - *Field Summary* : liste des attributs.
 - *Constructor Summary* : liste des constructeurs de la classe.
 - *Method Summary* : liste des méthodes.

- *Field Details* : description détaillée des attributs.
- *Constructor Details* : description détaillée des constructeurs de la classe.
- *Method Details* : description détaillée des méthodes de la classe.

Les différentes étapes de création d'un programme Java

1. Création des fichiers source

Dans un premier temps, il vous faut créer un ou plusieurs fichiers de code source, selon l'importance de votre programme.

Tout code Java est contenu à l'intérieur d'une classe qui est elle-même contenue dans un fichier portant l'extension `.java`.

Plusieurs classes peuvent exister dans un même fichier `.java` mais une seule peut être déclarée publique, et c'est cette classe qui donne son nom au fichier.

Comme beaucoup d'autres langages de programmation, les fichiers source Java sont des fichiers de texte sans mise en forme.

Un simple éditeur de texte capable d'enregistrer au format texte ASCII, tel que le Bloc-notes de Windows ou VI sous Unix, est suffisant pour écrire des sources Java.

Une fois le code de votre fichier source écrit, il faut l'enregistrer avec l'extension `.java` qui est l'extension des fichiers source.

Si vous utilisez le Bloc-notes de Windows, faites attention lors de l'enregistrement de votre fichier que le Bloc-notes n'ajoute pas une extension `.txt` au nom de votre fichier. Pour éviter ce problème, nommez votre fichier avec l'extension `.java`, le tout placé entre guillemets.

Il y a toutefois mieux qu'un simple éditeur de texte pour développer. Vous pouvez, moyennant le coût d'une licence, utiliser des outils commerciaux ou encore mieux utiliser des produits open source comme l'excellent **Eclipse**. Il s'agit au départ d'un projet IBM mais de nombreuses sociétés se sont jointes à ce projet (Borland, Oracle, Merant...). C'est un outil de développement Java excellent et gratuit auquel on peut lier d'autres applications tiers via un système de plug-in. Oracle propose également **NetBeans** un outil très efficace et simple d'utilisation.

2. Compiler un fichier source

Une fois votre fichier source réalisé et enregistré avec l'extension `.java`, il faut compiler votre fichier.

Pour compiler un fichier source Java, il faut utiliser l'outil en ligne de commande `javac` fourni avec le SDK.

- Ouvrez une fenêtre **Invite de commandes**.
- À l'invite de commandes, placez-vous dans le dossier contenant votre fichier source (`.java`), à l'aide de la commande `cd` suivie d'un espace puis du nom du dossier qui contient votre fichier source.
- Une fois que vous êtes dans le bon dossier, vous pouvez lancer la compilation de votre fichier à l'aide de la commande suivante à l'invite de commandes : `javac <nomFichier>.java`.

`javac` : compilateur Java en ligne de commande, fourni avec le JDK.

`<nomFichier>` : nom du fichier source Java.

`.java` : extension qui indique que le fichier est une source Java.

Si vous voulez compiler plusieurs fichiers source en même temps, il suffit de saisir la commande précédente, puis d'ajouter les autres fichiers à compiler en les séparant par un espace.

`javac <nomFichier1>.java <nomFichier2>.java`

Si au bout de quelques secondes vous voyez apparaître de nouveau l'invite de commandes, c'est que votre fichier ne contient pas d'erreur et qu'il a été compilé. En effet le compilateur n'affiche pas de message quand la compilation se déroule correctement.

Le résultat de la compilation d'un fichier source Java est la création d'un fichier binaire portant le même nom que le fichier source mais avec l'extension .class.

Un fichier binaire .class contient le pseudo-code Java qui peut être interprété par la machine virtuelle Java. Si par contre, vous voyez apparaître une suite de messages dont le dernier vous indique un nombre d'erreurs, c'est que votre fichier source contient des erreurs et que javac n'a donc pas réussi à le compiler.

```
C:\thierry\java\melangeChaine>javac Principale.java
Principale.java:20: ']' expected
    String[ mots;
           ^
Principale.java:20: not a statement
    String[ mots;
           ^
Principale.java:30: ';' expected
                                System.out.print(melange(mots[i]));
                                         ^
3 errors
C:\thierry\java\melangeChaine>
```

Dans ce cas, il vous faut corriger votre fichier source.

Pour vous aider à trouver les erreurs dans votre ou vos fichiers source, javac vous fournit des informations très utiles :

<nomFichier.java> : <numLigne> : <message> <ligne de code>

<nomFichier>

Nom du fichier source Java qui contient une erreur.

<numLigne>

Numéro de la ligne de votre fichier source où javac a décelé une erreur.

<message>

Message indiquant le type de l'erreur.

<ligne>

Ligne de code contenant une erreur, javac indique par une flèche où est située l'erreur dans la ligne.

Après avoir corrigé votre code, recompilez votre fichier. Si javac vous indique toujours des erreurs, renouvez l'opération de correction puis de recompilation du fichier jusqu'à obtenir la création du fichier binaire .class.

Par défaut les fichiers compilés sont créés dans le même répertoire que vos fichiers source. Vous pouvez

indiquer à l'outil javac de les créer dans un autre répertoire à l'aide de l'option -d "directory".

3. Exécuter une application

Une application Java est un programme autonome, semblable aux programmes que vous connaissez, mais qui, pour être exécuté, nécessite l'emploi d'un interpréteur Java (la machine virtuelle Java) qui charge la méthode `main()` de la classe principale de l'application.

Pour lancer l'exécution d'une application Java, il faut utiliser l'outil en ligne de commande `java` fourni avec le JDK.

- Ouvrez une fenêtre **Invite de commandes**. Placez-vous dans le répertoire qui contient le ou les fichiers binaires (.class) de votre application, puis saisissez la commande avec la syntaxe suivante :

```
java <fichierMain> <argumentN> <argumentN+1>
```

`java` : outil en ligne de commande qui lance l'exécution de la machine virtuelle Java.

`<fichierMain>` : est obligatoirement le nom du fichier binaire (.class) qui contient le point d'entrée de l'application, la méthode `main()`. Important : ne mettez pas l'extension .class après le nom du fichier car ceci est fait implicitement par la machine virtuelle Java.

`<argumentN> <argumentN+1>` : éventuels arguments de ligne de commande à passer à l'application à l'exécution de celle-ci.

Si vous avez lancé l'exécution correctement (syntaxe correcte, avec le fichier contenant la méthode `main()`), vous devez voir apparaître les éventuels messages que vous avez insérés dans votre code. Si par contre vous voyez apparaître un message d'erreur semblable à `Exception in thread "main" java.lang.NoClassDefFoundError : ...` c'est que votre programme ne peut pas être exécuté.

Plusieurs raisons peuvent en être la cause :

- Le nom du fichier à exécuter ne porte pas le même nom que la classe (différence entre majuscules et minuscules).
- Vous avez saisi l'extension .class après le nom du fichier à exécuter sur la ligne de commande.
- Le fichier que vous exécutez ne contient pas de méthode `main()`.
- Vous essayez d'exécuter un fichier binaire (.class) qui est situé dans un autre répertoire que celui d'où vous lancez l'exécution.

Votre première application Java

1. Squelette d'une application

Une application Java est un programme autonome qui peut être exécuté sur n'importe quelle plate-forme disposant d'une machine virtuelle Java.

Tout type d'application peut être développé en Java : interface graphique, accès aux bases de données, applications client/serveur, multithreading...

Une application est composée au minimum d'un fichier .class qui doit lui-même contenir au minimum le point d'entrée de l'application, la méthode main().

Exemple de la structure minimum d'une application

```
public class MonApplication {  
    public static void main(String arguments[]) {  
        /* corps de la méthode principale */  
    }  
}
```

Si l'application est importante, il est possible de créer autant de classes que nécessaire. Les classes qui ne contiennent pas la méthode main() sont nommées classes auxiliaires.

La méthode main() est le premier élément appelé par la machine virtuelle Java au lancement de l'application.

Le corps de cette méthode doit contenir les instructions nécessaires pour le lancement de l'application, c'est-à-dire la création d'instances de classe, l'initialisation de variables et l'appel de méthodes.

Idéalement, la méthode main() peut ne contenir qu'une seule instruction.

La déclaration de la méthode main() se fait toujours suivant la syntaxe suivante :

```
public static void main(String <identificateur>[ ] ) {...}
```

public

Modificateur d'accès utilisé pour rendre la méthode accessible à l'ensemble des autres classes et objets de l'application, et également pour que l'interpréteur Java puisse y accéder de l'extérieur au lancement de l'application.

static

Modificateur d'accès utilisé pour définir la méthode main() comme étant une méthode de classe. La machine virtuelle Java peut donc appeler cette méthode sans avoir à créer une instance de la classe dans laquelle elle est définie.

void

Mot clé utilisé pour indiquer que la méthode est une procédure qui ne retourne pas de valeur.

main

Identificateur de la méthode.

String <identificateur>[] :

Paramètre de la méthode, c'est un tableau de chaînes de caractères. Ce paramètre est utilisé pour

passer des arguments en ligne de commande au lancement de l'application. Dans la plupart des programmes, le nom utilisé pour <identificateur> est argument ou args, pour indiquer que la variable contient des arguments pour l'application.

2. Arguments en ligne de commande

a. Principes et utilisation

Une application Java étant un programme autonome, il peut être intéressant de lui fournir des paramètres ou des options qui vont déterminer le comportement ou la configuration du programme au lancement de celui-ci.

- Les arguments en ligne de commande sont stockés dans un tableau de chaînes de caractères. Si vous voulez utiliser ces arguments sous un autre format, vous devez faire une conversion de type, du type String vers le type désiré lors de l'utilisation de l'argument.

Dans quels cas faut-il utiliser les arguments en ligne de commande ?

Les arguments en ligne de commande sont à utiliser au lancement d'une application dès qu'une ou des données utilisées à l'initialisation de votre programme peuvent prendre des valeurs variables en fonction de l'environnement. Par exemple :

- nom du port de communication utilisé dans le cas d'une communication avec un périphérique matériel.
- adresse IP d'une machine sur le réseau dans le cas d'une application client/serveur.
- nom d'utilisateur et mot de passe dans le cas d'une connexion à une base de données avec gestion des permissions d'accès.

Par exemple, dans le cas d'une application qui accède à une base de données, il faut en général fournir un nom d'utilisateur et un mot de passe pour ouvrir une session d'accès à la base. Des utilisateurs différents peuvent accéder à la base de données, mais avec des permissions différentes. Il peut donc exister plusieurs sessions différentes. Il n'est pas envisageable de créer une version de l'application pour chaque utilisateur.

De plus, ces informations sont susceptibles d'être modifiées. Il n'est donc pas judicieux de les intégrer dans votre code, car tout changement vous obligera à modifier votre code source et à le recompiler et à détenir une version pour chaque utilisateur.

La solution à ce problème réside dans les arguments en ligne de commande.

Il suffit dans votre code d'utiliser le tableau d'arguments de la méthode main qui contient les variables (nom et mot de passe) de votre application.

Ensuite, selon l'utilisateur du programme, il faut au lancement de l'application par l'instruction java, faire suivre le nom de la classe principale par la valeur des arguments de ligne de commande de l'application.

b. Passage d'arguments à une application Java au moment de l'exécution

Le passage d'arguments à une application Java se fait au lancement de l'application par l'intermédiaire de la ligne de commande. L'exemple de programme suivant montre comment utiliser le passage d'arguments en ligne de commande dans une application Java.

```
/* Déclaration de la classe principale de l'application */
public class MaClasse
{
    /* Déclaration de la méthode point d'entrée de l'application*/
```

```
public static void main(String args[])
{
    /* Affichage des arguments de la ligne de commande */
    for (int i = 0 ; i < args.length; i++)

        System.out.printIn("Argument " +i + " = " + args[i]);

    }

    /* Conversion de deux arguments de la ligne de commande de
String vers int, puis addition des valeurs entières, et
affichage du résultat */

    int somme;
    somme=(Integer.parseInt(args[3]))+(Integer.parseInt(args[4]));
    System.out.println("Argument 3 + Argument 4 = " + somme);
}
}
```

Après compilation, le programme s'exécute avec la ligne de commande suivante :

```
java MaClasse éditions ENI "éditions ENI" 2 5
```

L'exécution du programme affiche les informations suivantes :

Argument	0	=	éditions
Argument	1	=	ENI
Argument	2	=	éditions ENI
Argument	3	=	2
Argument	4	=	5
Argument	3	+	Argument 4 = 7

Les variables, constantes et énumérations

1. Les variables

Les variables vont vous permettre de mémoriser pendant l'exécution de votre application différentes valeurs utiles pour le fonctionnement de votre application. Une variable doit obligatoirement être déclarée avant son utilisation dans le code. Lors de la déclaration d'une variable nous allons fixer ses caractéristiques. En fonction de l'emplacement de sa déclaration une variable appartiendra à une des catégories suivantes :

- Déclarée à l'intérieur d'une classe la variable est une variable d'instance. Elle n'existera que si une instance de la classe est disponible. Chaque instance de classe aura son propre exemplaire de la variable.
- Déclarée avec le mot clé `static` à l'intérieur d'une classe la variable est une variable de classe. Elle est accessible directement par le nom de la classe et n'existe qu'en un seul exemplaire.
- Déclarée à l'intérieur d'une fonction la variable est une variable locale. Elle n'existe que pendant l'exécution de la fonction et n'est accessible que par le code de celle-ci.
- Les paramètres des fonctions peuvent être considérés comme des variables locales. La seule différence réside dans l'initialisation de la variable qui est effectuée lors de l'appel de la fonction.

a. Nom des variables

Voyons les règles à respecter pour nommer les variables.

- Le nom d'une variable commence obligatoirement par une lettre.
- Il peut être constitué de lettres, de chiffres ou du caractère souligné (`_`).
- Il peut contenir un nombre quelconque de caractères (pratiquement il vaut mieux se limiter à une taille raisonnable).
- Il y a une distinction entre minuscules et majuscules (la variable AGEDUCAPITAINe est différente de la variable ageducapitaine).
- Les mots clés du langage ne doivent pas être utilisés comme nom de variable.
- Par convention les noms de variable sont orthographiés en lettres minuscules sauf la première lettre de chaque mot si le nom de la variable en comporte plusieurs (ageDuCapitaine).

b. Type des variables

En spécifiant un type pour une variable nous indiquons quelles informations nous allons pouvoir stocker dans cette variable et les opérations que nous allons pouvoir effectuer avec.

Deux catégories de types de variables sont disponibles :

- Les types valeur : la variable contient réellement les informations.
- Les types référence : la variable contient l'adresse mémoire où se trouvent les informations.

Le langage Java dispose de sept types de base qui peuvent être classés en trois catégories.

Les types numériques entiers

Types entiers signés			
byte	-128	127	8 bits
short	-32768	32767	16 bits

int	-2147483648	2147483647	32 bits
long	-9223372036854775808	9223372036854775807	64 bits

Lorsque vous choisissez un type pour vos variables entières vous devez prendre en compte les valeurs minimale et maximale que vous envisagez de stocker dans la variable afin d'optimiser la mémoire utilisée par la variable. Il est en effet inutile d'utiliser un type long pour une variable dont la valeur n'excédera pas 50, un type byte est dans ce cas suffisant. L'économie de mémoire semble dérisoire pour une variable unique mais devient appréciable lors de l'utilisation de tableaux de grande dimension.

Tous les types entiers sont signés. Il est cependant possible de travailler avec des valeurs entières non signées en utilisant les classes `Integer` et `Long`. Ceci permet d'étendre la valeur positive maximale admissible pour un type `int` jusqu'à 4294967296 et jusqu'à 18446744073709551616 pour un type `long`. Il faut toutefois prendre quelques précautions. Par exemple, le code suivant ne pourra pas être compilé.

```
distance=new Integer(3000000000);
```

Le compilateur vérifie que la valeur littérale fournie au constructeur ne dépasse pas les limites admissibles pour un type `int` et génère ici une erreur. Pour pouvoir s'affranchir de cette limitation, il faut utiliser la méthode statique `parseUnsignedInt` qui accepte comme paramètre une chaîne de caractères.

```
int distance;
distance=Integer.parseUnsignedInt("3000000000");
```

L'utilisation ultérieure de cette variable devra bien sûr tenir compte de la spécificité de son type non signé. L'affichage de son contenu devra être effectué avec la méthode statique `toUnsignedString`. Le code suivant permet de mettre en évidence cette spécificité.

```
System.out.println("affichage en tant que int :" + distance);
System.out.println("affichage en tant que int non signé :"
+Integer.toUnsignedString(distance));
```

Ce code affiche les informations suivantes sur la console :

```
affichage en tant que int :-1294967296
affichage en tant que int non signé :3000000000
```

Les types décimaux

float	1.4E-45	3.4028235E38	4 octets
double	4.9E-324	1.7976931348623157E308	8 octets

Tous les types décimaux sont signés et peuvent donc contenir des valeurs positives ou négatives.

Le type caractère

Le type `char` est utilisé pour stocker un caractère unique. Une variable de type `char` utilise deux octets pour stocker le code Unicode du caractère. Dans jeu de caractère Unicode les 128 premiers caractères sont identiques au jeu de caractère ASCII, les caractères suivants, jusqu'à 255, correspondent aux caractères spéciaux de l'alphabet latin (par exemple les caractères accentués), le reste est utilisé pour les symboles ou les caractères d'autres alphabets. Les caractères spécifiques ou ceux ayant une signification particulière pour le langage Java sont représentés par une séquence d'échappement. Elle est constituée du caractère \ suivi d'un autre caractère indiquant la signification de la séquence d'échappement. Le tableau suivant présente la liste des séquences d'échappement et leurs significations.

séquence	signification
\t	Tabulation
\b	BackSpace

\n	Saut de ligne
\r	Retour chariot
\f	Saut de page
\'	Simple quote
\"	Double quote
\\\	Antislash

Les caractères unicode non accessibles au clavier sont eux aussi représentés par une séquence d'échappement constituée des caractères \u suivis de la valeur hexadécimale du code unicode du caractère. Le symbole euro est par exemple représenté par la séquence \u20AC.

Pour pouvoir stocker des chaînes de caractères il faut utiliser le type `String` qui représente une suite de zéro à n caractères. Ce type n'est pas un type de base mais une classe. Cependant pour faciliter son emploi, il peut être utilisé comme un type de base du langage. Les chaînes de caractères sont invariables, car lors de l'affectation d'une valeur à une variable de type chaîne de caractères de l'espace est réservé en mémoire pour le stockage de la chaîne. Si par la suite cette variable reçoit une nouvelle valeur un nouvel emplacement lui est assigné en mémoire. Heureusement ce mécanisme est transparent pour nous et la variable fera toujours automatiquement référence à la valeur qui lui a été assignée. Avec ce mécanisme les chaînes de caractères peuvent avoir une taille variable. L'espace occupé en mémoire est automatiquement ajusté en fonction de la longueur de la chaîne de caractères. Pour affecter une chaîne de caractères à une variable il faut saisir le contenu de la chaîne entre " et " comme dans l'exemple ci-dessous.

Exemple

```
nomDuCapitaine = "Crochet";
```

De nombreuses fonctions de la classe `String` permettent la manipulation des chaînes de caractères et seront détaillées plus loin dans ce chapitre.

Le type boolean

Le type `boolean` permet d'avoir une variable qui peut prendre deux états vrai/faux, oui/non, on/off.

L'affectation se fait directement avec les valeurs `true` ou `false` comme dans l'exemple suivant :

```
boolean disponible,modifiable;
disponible=true;
modifiable=false;
```

Il est impossible d'affecter une autre valeur à une variable de type `boolean`.

c. Valeurs par défaut

L'initialisation des variables n'est pas toujours obligatoire. C'est par exemple le cas pour les variables d'instance qui sont initialisées avec les valeurs par défaut suivantes.

Type	Valeur par défaut
byte	0
short	0
int	0
long	0
float	0.0

double	0.0
char	\u0000
boolean	false
String	null

Par contre les variables locales doivent être initialisées avant d'être utilisées. Le compilateur effectue d'ailleurs une vérification lorsqu'il rencontre l'utilisation d'une variable locale et déclenche une erreur si la variable n'a pas été initialisée.

d. Valeurs littérales

Les valeurs numériques entières peuvent être utilisées avec leur représentation décimale, octale, hexadécimale ou binaire. Les quatre lignes de code suivantes ont le même effet.

```
i=243;
i=0363;
i=0xF3;
i=0b11110011;
```

Les valeurs numériques réelles peuvent être exprimées avec la notation décimale ou la notation scientifique.

```
surface=2356.8f;
surface=2.3568e3f;
```

Vous pouvez insérer des caractères _ dans les valeurs numériques littérales pour faciliter la lecture. Les deux syntaxes suivantes sont équivalentes.

```
prix=1_234_876_567;
prix =1234876567;
```

Les valeurs littérales sont également typées. Les valeurs numériques entières sont par défaut considérées comme des types int. Les valeurs numériques réelles sont, elles, considérées comme des types double. Cette assimilation est parfois source d'erreurs de compilation lors de l'utilisation du type float. Les lignes suivantes génèrent une erreur de compilation car le compilateur considère que vous tentez d'affecter à une variable de type float une valeur de type double et qu'il y a dans ce cas un risque de perte d'information.

```
float surface;
surface=2356.8;
```

Pour résoudre ce problème il faut forcer le compilateur à considérer la valeur littérale réelle comme un type float en la faisant suivre par le caractère f ou F.

```
float surface;
surface=2356.8f;
```

e. Conversions de types

Les conversions de types consistent à transformer une variable d'un type dans un autre type. Les conversions peuvent se faire vers un type supérieur ou vers un type inférieur. Si une conversion vers un type inférieur est utilisée, il risque d'y avoir une perte d'information. Par exemple la conversion d'un type double vers un type long fera perdre la partie décimale de la valeur. C'est pour cette raison que le compilateur exige dans ce cas que vous indiquiez explicitement que vous souhaitez effectuer cette opération. Pour cela vous devez préfixer l'élément que vous souhaitez convertir avec le type que vous voulez obtenir en plaçant celui-ci entre parenthèses.

Vous perdez dans ce cas la partie décimale, mais c'est parfois le but de ce genre de conversion.

```

float surface;
surface=2356.8f;
int approximation;
approximation=(int)surface;

```

Les conversions vers un type supérieur sont sans risque de perte d'information et peuvent donc se faire directement par une simple affectation.

Le tableau suivant résume les conversions possibles et si elles doivent être explicites (😞) ou si elles sont implicites (😊).

		Type de données à obtenir						
		byte	short	int	long	float	double	char
Type de données d'origine	byte		😊	😊	😊	😊	😊	😊
	short	😞		😊	😊	😊	😊	😊
	int	😞	😞		😊	😊	😊	😊
	long	😞	😞	😞		😊	😊	😞
	float	😞	😞	😞	😞		😊	😞
	double	😞	😞	😞	😞	😞		😞
	char	😞	😊	😊	😊	😊	😊	

Les conversions à partir de chaînes de caractères et vers des chaînes de caractères sont plus spécifiques.

Conversion vers une chaîne de caractères

Les fonctions de conversion vers le type chaîne de caractères sont accessibles par l'intermédiaire de la classe `String`. La méthode de classe `valueOf` assure la conversion d'une valeur d'un type de base vers une chaîne de caractères.

Barrières
High adrien - 27b
Isough adrien - 27b

<code>static String</code>	<code>valueOf(boolean b)</code> Returns the string representation of the boolean argument.
<code>static String</code>	<code>valueOf(char c)</code> Returns the string representation of the char argument.
<code>static String</code>	<code>valueOf(char[] data)</code> Returns the string representation of the char array argument.
<code>static String</code>	<code>valueOf(char[] data, int offset, int count)</code> Returns the string representation of a specific subarray of the char array argument.
<code>static String</code>	<code>valueOf(double d)</code> Returns the string representation of the double argument.
<code>static String</code>	<code>valueOf(float f)</code> Returns the string representation of the float argument.
<code>static String</code>	<code>valueOf(int i)</code> Returns the string representation of the int argument.
<code>static String</code>	<code>valueOf(long l)</code> Returns the string representation of the long argument.
<code>static String</code>	<code>valueOf(Object obj)</code> Returns the string representation of the Object argument.

Dans certaines situations l'utilisation de ces fonctions est optionnelle car la conversion est effectuée implicitement. C'est le cas par exemple lorsqu'une variable d'un type de base est concaténée avec une chaîne de caractères. Les deux versions de code suivantes ont le même effet.

Version 1

```
double prixHt;  
prixHt=152;  
String recap;  
recap="le montant de la commande est : " + prixHt*1.196;
```

Version 2

```
double prixHt;  
prixHt=152;  
String recap;  
recap="le montant de la commande est : " +String.valueOf(prixHt*1.196);
```

Conversion depuis une chaîne de caractères

Il arrive fréquemment qu'une valeur numérique soit disponible dans une application sous forme d'une chaîne de caractères (saisie de l'utilisateur, lecture d'un fichier...).

Pour pouvoir être manipulée par l'application, elle doit être convertie en un type numérique. Ce type de conversion est accessible par l'intermédiaire des classes équivalentes aux types de base. Elles permettent la manipulation de valeurs numériques sous forme d'objets. Chaque type de base possède sa classe associée.

Type de base	Classe correspondante
byte	Byte
short	Short
int	Integer

long	Long
float	Float
double	Double
boolean	Boolean
char	Character

Ces classes sont appelées classes Wrapper car elles sont utilisées pour « emballer » dans un objet les types de base du langage. Elles peuvent être utilisées comme des classes normales en créant une instance à partir d'un des constructeurs disponibles. Cette solution peut être contournée grâce au mécanisme appelé « autoboxing » du compilateur.

Ce mécanisme permet l'affectation d'un type de base du langage à une variable du type wrapper correspondant. Les deux lignes de code suivantes sont équivalentes.

```
Integer entier=new Integer(10);
Integer entier=10;
```

Le mécanisme inverse, appelé bien sûr « unboxing », permet la conversion automatique d'un type wrapper vers un type de base. La variable entier de l'exemple précédent peut être affectée à une variable de type int.

```
int x;
x=entier;
```

Ces classes fournissent une méthode parse... acceptant comme paramètre une chaîne de caractères et permettant d'en obtenir la conversion dans le type de base associé à la classe.

Classe	Méthode
Byte	public static byte parseByte(String s)
Short	public static short parseShort(String s)
Integer	public static int parseInt(String s)
Long	public static long parseLong(String s)
Float	public static float parseFloat(String s)
Double	public static double parseDouble(String s)
Boolean	public static boolean parseBoolean(String s)

Pour retenir comment procéder pour effectuer une conversion, il faut appliquer le principe très simple : la méthode à utiliser se trouve dans la classe correspondante au type de données que l'on souhaite obtenir.

f. Déclaration des variables

La déclaration d'une variable est constituée du type de la variable suivi du nom de la variable. La syntaxe de base est donc la suivante :

```
int compteur;
double prix;
String nom;
```

Des modificateurs d'accès et une valeur initiale peuvent également être précisés lors de la déclaration.

```
private int compteur=0;
protected double prix=123.56;
public nom=null;
```

La déclaration d'une variable peut apparaître n'importe où dans le code. Il suffit simplement que la déclaration précède l'utilisation de la variable. Il est conseillé de regrouper les déclarations de variables en début de classe ou en début de fonction afin de faciliter la relecture du code.

La déclaration de plusieurs variables de même type peut être regroupée sur une seule ligne en séparant les noms des variables par une virgule.

```
protected double prixHt=123.56, prixTtc,fraisPort;
```

g. Portée des variables

La portée d'une variable est la portion de code à partir de laquelle on peut manipuler cette variable. Elle est fonction de l'emplacement où est située la déclaration.

Cette déclaration peut être faite dans le bloc de code d'une classe, le bloc de code d'une fonction ou un bloc de code à l'intérieur d'une fonction. Seul le code du bloc où est déclarée la variable peut l'utiliser. Si le même bloc de code est exécuté plusieurs fois pendant l'exécution de la fonction, cas d'une boucle while par exemple, la variable sera créée à chaque passage dans la boucle. L'initialisation de la variable est dans ce cas obligatoire. Il ne peut pas y avoir deux variables portant le même nom avec la même portée. Vous avez cependant la possibilité de déclarer une variable interne à une fonction, ou un paramètre d'une fonction avec le même nom qu'une variable déclarée au niveau de la classe. La variable déclarée au niveau de la classe est dans ce cas masquée par la variable interne à la fonction.

h. Niveau d'accès des variables

Le niveau d'accès d'une variable se combine avec la portée de la variable et détermine quelle portion de code a le droit de lire et d'écrire dans la variable. Un ensemble de mots clés permettent de contrôler le niveau d'accès. Ils s'utilisent lors de la déclaration de la variable et doivent être placés devant le type de la variable. Ils sont uniquement utilisables pour la déclaration d'une variable à l'intérieur d'une classe. Leur utilisation à l'intérieur d'une fonction est interdite.

private : la variable est utilisable uniquement par le code de la classe où elle est définie.

protected : la variable est utilisable dans la classe où elle est définie, dans les sous-classes de cette classe et dans les classes qui font partie du même package.

public : la variable est accessible à partir de n'importe quelle classe indépendamment du package.

aucun modificateur : la variable est accessible à partir de toutes les classes faisant partie du même package.

static : ce mot clé est associé à un des mots clé précédents pour transformer une déclaration de variable d'instance en déclaration de variable de classe (utilisable sans qu'une instance de la classe existe).

i. Durée de vie des variables

La durée de vie d'une variable nous permet de spécifier pendant combien de temps durant l'exécution de l'application le contenu d'une variable sera disponible.

Pour une variable déclarée dans une fonction la durée de vie correspond à la durée d'exécution de la fonction. Dès la fin de l'exécution de la procédure ou fonction, la variable est éliminée de la mémoire. Elle est recréée lors du prochain appel de la fonction. Une variable déclarée à l'intérieur d'une classe est utilisable tant qu'une instance de la classe est disponible. Les variables déclarées avec le mot clé static sont accessibles pendant toute la durée de fonctionnement de l'application.

2. Les constantes

Dans une application il arrive fréquemment que l'on utilise des valeurs numériques ou chaînes de caractères qui ne seront pas modifiées pendant le fonctionnement de l'application. Il est conseillé, pour faciliter la lecture du code, de définir ces valeurs sous forme de constantes.

La définition d'une constante se fait en ajoutant le mot clé `final` devant la déclaration d'une variable. Il est obligatoire d'initialiser la constante au moment de sa déclaration (c'est le seul endroit où il est possible de faire une affectation à la constante).

```
final double TAUXTVA=1.196;
```

La constante peut être alors utilisée dans le code à la place de la valeur littérale qu'elle représente.

```
prixTtc=prixHt*TAUXTVA;
```

Les règles concernant la durée de vie et la portée des constantes sont identiques à celles concernant les variables.

La valeur d'une constante peut également être calculée à partir d'une autre constante.

```
final double TOTAL=100;
final double DEMI=TOTAL/2;
```

De nombreuses constantes sont déjà définies au niveau du langage Java. Elles sont définies comme membres `static` des nombreuses classes du langage. Par convention les noms des constantes sont orthographiés entièrement en majuscules.

3. Les énumérations

Une énumération va nous permettre de définir un ensemble de constantes qui sont liées entre elles. La déclaration se fait de la manière suivante :

```
public enum Jours
{
    DIMANCHE,
    LUNDI,
    MARDI,
    MERCREDI,
    JEUDI,
    VENDREDI,
    SAMEDI
}
```

La première valeur de l'énumération est initialisée à zéro. Les constantes suivantes sont ensuite initialisées avec un incrément de un. La déclaration précédente aurait donc pu s'écrire :

```
public class Jours
{
    public static final int DIMANCHE=0;
    public static final int LUNDI=1;
    public static final int MARDI=2;
    public static final int MERCREDI=3;
    public static final int JEUDI=4;
    public static final int VENDREDI=5;
    public static final int SAMEDI=6;
}
```

C'est approximativement ce que fait le compilateur lorsqu'il analyse le code de l'énumération.

En fait la déclaration d'une énumération est une déclaration de classe « déguisée ». Cette classe hérite implicitement de la classe `java.lang.Enum`. Les éléments définis dans l'énumération sont les seules instances possibles de cette classe. Comme n'importe quelle classe, elle peut contenir des attributs, des constructeurs et des méthodes. L'exemple de code suivant présente ces possibilités.

```
public enum Daltons
{
```

```

JOE (1.40, 52),
WILLIAM (1.68, 72),
JACK (1.93, 83),
AVERELL (2.13, 89);

private final double taille;
private final double poids;

private Daltons(double taille, double poids)
{
    this.taille = taille;
    this.poids = poids;
}
private double taille() { return taille; }
private double poids() { return poids; }

double imc()
{
    return poids/(taille+taille);
}

}

```

Le constructeur est utilisé de manière implicite pour initialiser les constantes de chacun des éléments de l'énumération. Le constructeur d'une énumération doit obligatoirement être déclaré `private`. Plusieurs méthodes définies dans la classe de base (`java.lang.Enum`) permettent d'obtenir des informations sur les éléments de l'énumération. La méthode `toString` retourne une chaîne de caractères représentant le nom de la constante de l'énumération.

```

Daltons d;
d=Daltons.JACK;
System.out.println(d.toString());

```

La méthode `valueOf` effectue l'opération inverse en fournissant un des éléments de l'énumération dont le nom est indiqué par la chaîne de caractères passée en paramètre.

```

d=Daltons.valueOf("JOE");
System.out.println("poids : "+ d.poids());
System.out.println("taille : "+ d.taille());

```

La méthode `values` retourne sous forme d'un tableau toutes les valeurs possible de l'énumération.

```

System.out.println("les frères Dalton");
for(Daltons d: Dalton.values())
{
    System.out.println(d.toString());
}

```

Une fois définie, une énumération peut être utilisée comme un nouveau type de données. Vous pouvez donc déclarer une variable avec pour type votre énumération.

```
Jours repere;
```

Il est alors possible d'utiliser la variable en lui affectant une des valeurs définies dans l'énumération.

```
repere=Jours.LUNDI;
```

Lorsque vous faites référence à un élément de votre énumération, vous devez le faire précéder du nom de l'énumération comme dans l'exemple précédent. L'affectation à la variable d'autre chose qu'une des valeurs contenues dans l'énumération est interdite et provoque une erreur de compilation.

La déclaration d'une énumération ne peut pas se faire dans une procédure ou une fonction. Elle peut par contre être déclarée dans une classe mais il faudra dans ce cas préfixer le nom de l'énumération par le nom de la classe dans laquelle elle est définie lors de son utilisation. Pour que l'énumération soit autonome il suffit simplement de la déclarer dans son propre fichier.

La portée d'une énumération suit les mêmes règles que celle des variables (utilisation des mots clés `public`, `private`, `protected`).

Une variable de type énumération peut facilement être utilisée dans une structure `switch ... case`, il n'est dans ce cas pas nécessaire de faire précéder les membres de l'énumération du nom de l'énumération.

```
public static void testJour(Jours j)
{
    switch (j)
    {
        case LUNDI:
        case MARDI:
        case MERCREDI:
        case JEUDI:
            System.out.println("c'est dur de travailler");
            break;
        case VENDREDI:
            System.out.println("bientôt le week end !");
            break;
        case SAMEDI:
            System.out.println("enfin !");
            break;
        case DIMANCHE:
            System.out.println("et ça recommence !");
            break;
    }
}
```

4. Les tableaux

Les tableaux vont nous permettre de faire référence à un ensemble de variables de même type par le même nom et d'utiliser un index pour les différencier. Un tableau peut avoir une ou plusieurs dimensions. Le premier élément d'un tableau a toujours pour index zéro. Le nombre de cases du tableau est spécifié au moment de la création du tableau. Le plus grand index d'un tableau est donc égal au nombre de cases moins un. Après sa création les caractéristiques d'un tableau ne peuvent plus être modifiées (nombre de cases, type d'éléments stockés dans le tableau). La manipulation d'un tableau doit être décomposée en trois étapes :

- Déclaration d'une variable permettant de manipuler le tableau.
- Création du tableau (allocation mémoire).
- Stockage et manipulation des éléments du tableau.

Déclaration du tableau

La déclaration se fait comme une variable classique sauf que l'on doit ajouter à la suite du type de données ou du nom de la variable les caractères [et]. Il est préférable, pour une meilleure lisibilité du code, d'associer les caractères [et] au type de données. La ligne suivante déclare une variable de type tableau d'entiers.

```
int[] chiffreAffaire;
```

Création du tableau

Après la déclaration de la variable il faut créer le tableau en obtenant de la mémoire pour stocker ces éléments. C'est à ce moment que nous indiquons la taille du tableau. Les tableaux étant assimilés à des objets c'est donc l'opérateur `new` qui va être utilisé pour créer une instance du tableau. La valeur fournie par l'opérateur `new` est stockée dans la variable déclarée au préalable.

```
chiffreAffaire=new int[12];
```

Cette déclaration va créer un tableau avec douze cases numérotées de 0 à 11. La taille du tableau est définitive, il n'est donc pas possible d'agrandir ou de rétrécir un tableau déjà créé.

Une autre solution est disponible pour la création d'un tableau. Elle permet simultanément la déclaration de la variable, la création du tableau et l'initialisation de son contenu. La syntaxe est la suivante :

```
int[] chiffreAffaire={1234,563,657,453,986,678,564,234,786,123,534,975};
```

Il n'y a dans ce cas pas besoin de préciser de taille pour le tableau. Le dimensionnement se fera automatiquement en fonction du nombre de valeurs placées entre les accolades.

Utilisation du tableau

Les éléments des tableaux sont accessibles de la même manière qu'une variable classique. Il suffit juste d'ajouter l'index de l'élément que l'on veut manipuler.

```
chiffreAffaire[0]=12456;
```

Le contenu d'une case de tableau peut être utilisé exactement de la même façon qu'une variable du même type. Il faut être vigilant en manipulant un tableau et ne pas tenter d'accéder à une case du tableau qui n'existe pas sous peine d'obtenir une exception du type `ArrayIndexOutOfBoundsException`.

Tableaux à plusieurs dimensions

Les tableaux à plusieurs dimensions sont en fait des tableaux contenant d'autres tableaux. La syntaxe de déclaration est semblable à celle d'un tableau mis à part que l'on doit spécifier autant de paires de crochets que vous souhaitez avoir de dimensions.

```
int[][] matrice;
```

La création est également semblable à celle d'un tableau à une dimension hormis que vous devez indiquer une taille pour chacune des dimensions.

```
matrice=new int[2][3];
```

L'accès à un élément du tableau se fait de manière identique en indiquant les index permettant d'identifier la case du tableau concernée.

```
matrice[0][0]=99;
```

La syntaxe permettant l'initialisation d'un tableau à plusieurs dimensions au moment de sa déclaration est un petit peu plus complexe.

```
int[][] grille={{11,12,13},{21,22,23},{31,32,33}};
```

Cet exemple crée un tableau à deux dimensions de trois cases sur trois cases.

La création avec cette technique de tableaux de grande taille à plusieurs dimensions risque d'être périlleuse.

Manipulations courantes avec des tableaux

Lorsque l'on travaille avec les tableaux, certaines opérations doivent être fréquemment réalisées. Ce paragraphe décrit les opérations les plus courantes réalisées sur les tableaux. La plupart d'entre elles sont disponibles grâce à la classe `java.util.Arrays` fournissant de nombreuses méthodes static de manipulation de tableaux.

Obtenir la taille d'un tableau : il suffit d'utiliser la propriété `length` du tableau pour connaître le nombre d'éléments qu'il peut contenir. Dans le cas d'un tableau multidimensionnel, il faut se souvenir qu'il s'agit en fait de tableaux de tableaux. La propriété `length` indique alors le nombre d'éléments sur la première

dimension.

Pour obtenir la même information sur les autres dimensions, il faut utiliser la propriété `length` de chaque case du tableau de niveau inférieur.

```
matrice=new int[8][3];
System.out.println("le tableau comporte " + matrice.length +
    " cases sur " + matrice[0].length +
    " cases");
```

Rechercher un élément dans un tableau : la fonction `binarySearch` permet d'effectuer une recherche dans un tableau. Elle accepte comme paramètres le tableau dans lequel se fait la recherche et l'élément recherché dans le tableau. La valeur renournée correspond à l'index où l'élément a été trouvé dans le tableau ou une valeur négative si l'élément ne se trouve pas dans le tableau. Pour que cette fonction fonctionne correctement le tableau doit être au préalable trié.

```
int[] chiffreAffaire={1234,563,657,453,986,678,564,234,786,123,534,975};
Arrays.sort(chiffreAffaire);
System.out.println(Arrays.binarySearch(chiffreAffaire, 123));
```

Trier un tableau : la fonction `sort` assure le tri du tableau qu'elle reçoit en paramètre. Le tri se fait par ordre alphabétique pour les tableaux de chaîne de caractères et par ordre croissant pour les tableaux de valeurs numériques.

```
int[] chiffreAffaire={1234,563,657,453,986,678,564,234,786,123,534,975};
Arrays.sort(chiffreAffaire);
for (int i=0;i<chiffreAffaire.length;i++)
{
    System.out.print(chiffreAffaire[i] + "\t");
}
```

Affiche le résultat suivant :

```
123    234    453    534    563    564    657    678    786    975    986    1234
```

La fonction `parallelSort` effectue elle aussi le tri du tableau mais en utilisant un algorithme exploitant les capacités d'une machine multiprocesseur.

Afficher un tableau : la fonction `toString` permet d'obtenir une représentation sous forme d'une chaîne de caractères du tableau passé en paramètre.

```
System.out.println(Arrays.toString(chiffreAffaire));
```

Affiche le résultat suivant :

```
[123, 234, 453, 534, 563, 564, 657, 678, 786, 975, 986, 1234]
```

La fonction `deepToString` effectue la même opération mais pour un tableau à plusieurs dimensions.

```
int[][] grille={{11,12,13},{21,22,23},{31,32,33}};
System.out.println(Arrays.deepToString(grille));
```

Affiche le résultat suivant :

```
[[11, 12, 13], [21, 22, 23], [31, 32, 33]]
```

Copier un tableau : deux fonctions sont disponibles pour la copie de tableaux.

La fonction `copyOf` copie un tableau entier avec la possibilité de modifier la taille du tableau. La fonction `copyOfRange` effectue une copie d'une partie d'un tableau.

```
int[] copieChiffreAffaire;
copieChiffreAffaire=Arrays.copyOf(chiffreAffaire, 24);
System.out.println(Arrays.toString(copieChiffreAffaire));
```

Affiche le résultat suivant :

```
[1234, 563, 657, 453, 986, 678, 564, 234, 786, 123, 534, 975, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0]
```

```
int[] premierTrimestre;
premierTrimestre=Arrays.copyOfRange(chiffreAffaire, 0, 3);
System.out.println(Arrays.toString(premierTrimestre));
```

Affiche le résultat suivant :

```
[1234, 563, 657]
```

Remplir un tableau : la fonction `fill` est utilisable pour remplir toutes les cases d'un tableau avec la même valeur.

5. Les chaînes de caractères

Les variables de type `String` permettent la manipulation de chaînes de caractères par votre application.

Nous allons regarder comment réaliser les opérations les plus courantes sur les chaînes de caractères.

Création d'une chaîne de caractères

La méthode la plus simple pour créer une chaîne de caractères consiste à considérer le type `String` comme un type de base du langage et non comme un type objet. C'est dans ce cas l'affectation d'une valeur à la variable qui va provoquer la création d'une instance de la classe `String`. La création d'une chaîne de caractères comme un objet est bien sûr également possible en utilisant l'opérateur `new` et un des nombreux constructeurs disponibles dans la classe `String`. L'exemple de code suivant présente les deux solutions.

```
String chaine1="eni";
String chaine2=new String("eni");
```

Après sa création une chaîne de caractères ne peut plus être modifiée. L'affectation d'une autre valeur à la variable provoque la création d'une nouvelle instance de la classe `String`. La classe `String` contient de nombreuses méthodes permettant la modification de chaînes de caractères. À l'utilisation, nous avons l'impression que la fonction modifie le contenu de la chaîne initiale mais en fait c'est une nouvelle instance contenant le résultat qui est renvoyée par la fonction.

Affectation d'une valeur à une chaîne

Nous avons vu que pour affecter une valeur à une chaîne il faut la spécifier entre les caractères " et ", un problème se pose si nous voulons que le caractère " fasse partie de la chaîne. Pour qu'il ne soit pas interprété comme caractère de début ou de fin de chaîne il faut le protéger par une séquence d'échappement comme dans l'exemple ci-dessous :

```
String Chaine;
Chaine=" il a dit : \" ça suffit ! \\"";
```

Nous obtenons à l'affichage : il a dit : "ça suffit ! "

Pour les exemples suivants, nous allons travailler avec deux chaînes :

```
chaine1 = "l'hiver sera pluvieux";
chaine2 = "l'hiver sera froid";
```

Extraction d'un caractère particulier

Pour obtenir le caractère présent à une position donnée d'une chaîne de caractères, il faut utiliser la fonction `charAt` en fournissant comme argument l'index du caractère que l'on souhaite obtenir. Le premier caractère a l'index zéro comme pour un tableau. Cette fonction retourne un caractère (`char`).

```
System.out.println("le troisième caractère de la chaine1 est " +
chaine1.charAt(2));
```

Obtenir la longueur d'une chaîne

Pour déterminer la longueur d'une chaîne, la fonction `length` de la classe `String` est disponible.

```
System.out.println("la chaine1 contient " + chaine1.length() + " caractères");
```

Découpage de chaîne

La fonction `substring` de la classe `String` retourne une portion de chaîne en fonction de la position de départ et de la position de fin qui lui sont passées comme paramètres. La chaîne obtenue commence par le caractère situé à la position de départ et se termine au caractère précédent la position de fin.

```
System.out.println("un morceau de la chaine1 : " + chaine1.substring(2,8));
```

Nous obtenons à l'affichage :

```
un morceau de la chaine1 : hiver
```

Comparaison de chaînes

Lorsqu'on fait une comparaison de deux chaînes, on est tenté d'utiliser le double égal (`==`), comme précédemment. Cet opérateur fonctionne correctement sur les types de base mais il ne faut pas perdre de vue que les chaînes de caractères sont des types objet. Il faut donc utiliser les méthodes de la classe `String` pour effectuer des comparaisons de chaînes de caractères. La méthode `equals` effectue une comparaison de la chaîne avec celle qui est passée comme paramètre. Elle retourne un `boolean` égal à `true` si les deux chaînes sont identiques et bien sûr un `boolean` égal à `false` dans le cas contraire. Cette fonction fait une distinction entre minuscules et majuscules lors de la comparaison. La fonction `equalsIgnoreCase` effectue un traitement identique mais sans tenir compte de cette distinction.

```
if (chaine1.equals(chaine2))
{
    System.out.println("les deux chaines sont identiques");
}
else
{
    System.out.println("les deux chaines sont différentes");
}
```

 Ne vous laissez pas tromper par les apparences. Dans certains cas, l'opérateur `==` est bien capable de réaliser une comparaison correcte de chaînes de caractères. Le code ci-dessous fonctionne correctement et fournit bien le résultat attendu et considère que les deux chaînes sont identiques.

```
String s1="toto";
String s2="toto";
if (s1==s2)
```

```

    {
        System.out.println("chaines identiques");
    }
    else
    {
        System.out.println("chaines différentes");
    }
}

```

En fait, pour économiser de l'espace en mémoire, Java n'utilise dans ce cas qu'une seule instance de la classe String pour les variables s1 et s2 car le contenu des deux chaînes est identique.

Les deux variables s1 et s2 référencent donc la même zone mémoire et l'opérateur == constate donc l'égalité.

Si par contre nous utilisons le code suivant qui demande explicitement la création d'une instance de la classe String pour chacune des variables s1 et s2, l'opérateur == ne constate bien sûr plus l'égalité des chaînes.

```

String s1=new String("toto");
String s2=new String("toto");
if (s1==s2)
{
    System.out.println("chaines identiques");
}
else
{
    System.out.println("chaines différentes");
}

```

Pour réaliser un classement, vous devez par contre utiliser la méthode compareTo de la classe String ou la fonction compareToIgnoreCase. Avec ces deux solutions il faut passer comme paramètres la chaîne à comparer. Le résultat de la comparaison est retourné sous forme d'un entier inférieur à zéro si la chaîne est inférieure à celle reçue comme paramètre, égal à zéro si les deux chaînes sont identiques, et supérieur à zéro si la chaîne est supérieure à celle reçue comme paramètre.

```

if (chaine1.compareTo(chaine2)>0)
{
    System.out.println("chaine1 est supérieure à chaine2");
}
else
if (chaine1.compareTo(chaine2)<0)
{
    System.out.println("chaine1 est inférieure à chaine2");
}
else
{
    System.out.println("les deux chaines sont identiques");
}

```

Les fonctions startsWith et endsWith permettent de tester si la chaîne débute par la chaîne reçue en paramètre ou si la chaîne se termine par la chaîne reçue en paramètre. La fonction endsWith peut par exemple être utilisée pour tester l'extension d'un nom de fichier.

```

String nom="Code.java";
if (nom.endsWith(".java"))
{
    System.out.println("c'est un fichier source java");
}

```

Suppression des espaces

La fonction trim permet de supprimer les espaces situés avant le premier caractère significatif et après le dernier caractère significatif d'une chaîne.

```
String chaine="        eni      ";
System.out.println("longueur de la chaîne : " + chaine.length());
System.out.println("longueur de la chaîne nettoyée : " + chaine.trim()
.length());
```

Changer la casse

Tout en majuscules :

```
System.out.println(chaine1.toUpperCase());
```

Tout en minuscules :

```
System.out.println(chaine1.toLowerCase());
```

Recherche dans une chaîne

La méthode `indexOf` de la classe `String` permet la recherche d'une chaîne à l'intérieur d'une autre. Le paramètre correspond à la chaîne recherchée. La fonction retourne un entier indiquant la position à laquelle la chaîne a été trouvée ou -1 si la chaîne n'a pas été trouvée. Par défaut la recherche commence au début de la chaîne, sauf si vous utilisez une autre version de la fonction `indexOf` qui, elle, attend deux paramètres, le premier paramètre étant pour cette version, la chaîne recherchée et le deuxième la position de départ de la recherche.

```
String recherche;
int position;
recherche = "e";
position = chaine1.indexOf(recherche);
while (position > 0)
{
    System.out.println("chaîne trouvée à la position " + position);
    position = chaine1.indexOf(recherche, position+1);
}
System.out.println("fin de la recherche");
```

Nous obtenons à l'affichage :

```
chaîne trouvée à la position 5
chaîne trouvée à la position 9
chaîne trouvée à la position 18
fin de la recherche
```

Remplacement dans une chaîne

Il est parfois souhaitable de pouvoir rechercher la présence d'une chaîne à l'intérieur d'une autre, comme dans l'exemple précédent, mais également remplacer les portions de chaînes trouvées. La fonction `replace` permet de spécifier une chaîne de substitution pour la chaîne recherchée. Elle attend deux paramètres :

- la chaîne recherchée
- la chaîne de remplacement

```
String chaine3;
chaine3= chaine1.replace("hiver", "ete");
System.out.println(chaine3);
```

Nous obtenons à l'affichage :

Formatage d'une chaîne

La méthode `format` de la classe `String` permet d'éviter de longues et fastidieuses opérations de conversion et de concaténation. Le premier paramètre attendu par cette fonction est une chaîne de caractères spécifiant sous quelle forme on souhaite obtenir le résultat. Cette chaîne contient un ou plusieurs motifs de formatage représentés par le caractère `%` suivi d'un caractère spécifique indiquant sous quelle forme doit être présentée l'information. Il doit ensuite y avoir autant de paramètres qu'il y a de motifs de formatage. La chaîne renvoyée est construite par le remplacement de chacun des motifs de formatage par la valeur du paramètre correspondant, le remplacement se faisant dans l'ordre d'apparition des motifs. Le tableau suivant présente les principaux motifs de formatage disponibles.

Motif	Description
<code>%b</code>	Insertion d'un booléen
<code>%s</code>	Insertion d'une chaîne de caractères
<code>%d</code>	Insertion d'un nombre entier
<code>%o</code>	Insertion d'un entier affiché en octal
<code>%x</code>	Insertion d'un entier affiché en hexadécimal
<code>%f</code>	Insertion d'un nombre décimal
<code>%e</code>	Insertion d'un nombre décimal affiché au format scientifique
<code>%n</code>	Insertion d'un saut de ligne

L'exemple de code suivant :

```
boolean b=true;
int i=56;
double d=19.6;
String s="chaine";
System.out.println(String.format("boolean : %b %n" +
    "chaîne de caractères : %s %n" +
    "entier : %d %n" +
    "entier en hexadécimal : %x %n" +
    "entier en octal : %o %n" +
    "décimal : %f %n" +
    "décimal au format scientifique : %e%n",
    b,s,i,i,i,d,d));
```

affiche ce résultat sur la console :

```
boolean : true
chaîne de caractères : chaine
entier : 56
entier en hexadécimal : 38
entier en octal : 70
décimal : 19,600000
décimal au format scientifique : 1,960000e+01
```

6. Date et heure

La gestion de date et d'heure a longtemps été la bête noire des développeurs Java. La classe `GregorianCalendar` était disponible pour répondre aux problèmes de manipulation de date et d'heure. De nombreuses fonctionnalités étaient prévues mais leur utilisation relevait parfois du casse-tête. Il est vrai que le problème est complexe. Travailler en base 60 pour les secondes et les minutes puis en base 24 pour les heures n'est pas très simple. Mais la palme revient à la gestion des mois qui n'ont pas tous le même nombre de jours, voire pire puisque certains mois ont un nombre de jours variable suivant les

années. Les ordinateurs utilisent une technique différente, en ne travaillant pas directement avec des dates et heures mais en nombre de secondes ou de millisecondes depuis une date de référence (généralement le 1^{er} janvier 1970 à 0 heure). Ce mode de représentation n'est cependant pas très pratique pour un humain. La valeur 61380284400000 n'est pas très évocatrice, par contre 25/12/2014 est beaucoup plus parlant. C'est pourquoi de nombreuses fonctions permettent le passage d'un format à l'autre.

Dans la version 8 de Java, la gestion des dates et des heures a été complètement repensée. Au lieu de n'avoir qu'une ou deux classes dédiées à cette gestion et avec lesquelles il fallait jongler, de nombreuses classes spécialisées ont fait leur apparition.

LocalDate	Représente une date (jour mois année) sans heure.
LocalDateTime	Représente une date et une heure sans prise en compte du fuseau horaire.
LocalTime	Représente une heure sans prise en compte du fuseau horaire.
OffsetDateTime	Représente une date et une heure avec le décalage UTC.
OffsetTime	Représente une heure avec le décalage UTC.
ZonedDateTime	Représente une date et une heure avec le fuseau horaire correspondant.
Duration	Représente une durée exprimée en heures minutes secondes.
Period	Représente une durée exprimée en jours mois années.
MonthDay	Représente un jour et un mois sans année.
YearMonth	Représente un mois et une année sans jour.

Toutes ces classes proposent une série de méthodes permettant la manipulation de leurs éléments. Ces méthodes respectent une convention de nommage facilitant l'identification de leur usage.

- `of` : retourne une instance de la classe initialisée avec les différentes valeurs passées comme paramètres.

```
LocalDate noel;
noel=LocalDate.of(2014, 12, 25);
```

from : conversion entre les différents types. En cas de conversion vers un type moins complet, il y a perte d'informations.

```
LocalDateTime maintenant;
maintenant=LocalDateTime.now();
// transformation en LocalDate
// avec perte de l'heure
LocalDate aujourd'hui;
aujourd'hui=LocalDate.from(maintenant);
```

parse : transforme la chaîne de caractères passée comme paramètre vers le type correspondant. •

```
LocalTime horloge;
horloge=LocalTime.parse("22:45:03");
```

withxxxxxx : retourne une nouvelle instance en modifiant la composante indiquée par xxxx • par la valeur passée comme paramètre.

```
LocalTime horloge;
horloge=LocalTime.parse("22:45:03");
LocalTime nouvelleHeure;
nouvelleHeure=horloge.withHour(9);
```

- plusxxxxx et minusxxxx : retourne une nouvelle instance de la classe après ajout ou retrait du nombre d'unités indiqué par le paramètre. xxxxxx indique ce qui est ajouté ou retranché.

```
LocalDate paques;
paques=LocalDate.of(2014,4,20);
LocalDate ascension;
ascension=paques.plusDays(39);
```

atxxxxxx : combine l'objet reçu comme paramètre avec l'objet courant et retourne le résultat de • cette association. On peut par exemple combiner un objet LocalDate et un objet LocalTime pour obtenir un objet LocalDateTime.

```
LocalDate jourMatch;
jourMatch=LocalDate.of(2014,7,13);

LocalTime heureMatch;
heureMatch=LocalTime.of(21,00);

LocalDateTime fin;
fin=jourMatch.atTime(heureMatch);
```

Le petit exemple de code ci-dessous illustre quelques opérations sur les dates en comptant le nombre de jours fériés tombant un samedi ou un dimanche.

```
MonthDay[] fetes;
fetes=new MonthDay[8];
fetes[0]=MonthDay.of(1,1);
fetes[1]=MonthDay.of(5,1);
fetes[2]=MonthDay.of(5,8);
fetes[3]=MonthDay.of(7,14);
fetes[4]=MonthDay.of(8,15);
fetes[5]=MonthDay.of(11,1);
fetes[6]=MonthDay.of(11,11);
fetes[7]=MonthDay.of(12,25);

int nbJours;
int annee;
LocalDate jourTest;
for (annee=2014;annee<2030;annee++)
{
    nbJours=0;
    for(MonthDay test:fetes)
    {
        jourTest=test.atYear(annee);
        if (jourTest.getDayOfWeek()==DayOfWeek.SATURDAY
||jourTest.getDayOfWeek()==DayOfWeek.SUNDAY)
        {
            nbJours++;
        }
    }
    System.out.println("en " + annee + " il y a " + nbJours
+ " jour(s) ferie(s) un samedi ou un dimanche");
}
```

Les opérateurs

Les opérateurs sont des mots clés du langage permettant l'exécution d'opérations sur le contenu de certains éléments, en général des variables, des constantes, des valeurs littérales, ou des retours de fonctions. La combinaison d'un ou de plusieurs opérateurs et d'éléments sur lesquels les opérateurs vont s'appuyer se nomme une expression. Ces expressions sont évaluées au moment de l'exécution en fonction des opérateurs et des valeurs qui sont associées.

Deux types d'opérateurs sont disponibles :

- Les opérateurs unaires qui ne travaillent que sur un seul opérande.
- Les opérateurs binaires qui nécessitent deux opérandes.

Les opérateurs unaires peuvent être utilisés avec la notation préfixée, dans ce cas l'opérateur est placé avant l'opérande, et la notation postfixée avec dans ce cas l'opérateur placé après l'opérande. La position de l'opérateur détermine le moment où celui-ci est appliqué sur la variable. Si l'opérateur est préfixé il s'applique sur l'opérande avant que celui-ci ne soit utilisé dans l'expression. Avec la notation postfixée l'opérateur n'est appliqué sur la variable qu'après utilisation de celle-ci dans l'expression. Cette distinction peut avoir une influence sur le résultat d'une expression.

```
int i;  
i=3;  
System.out.println(i++);
```

Affiche 3 car l'incrémentation est exécutée après utilisation de la variable par l'instruction `println`.

```
int i;  
i=3;  
System.out.println(++i);
```

Affiche 4 car l'incrémentation est exécutée avant l'utilisation de la variable par l'instruction `println`.

Si la variable n'est pas utilisée dans une expression, les deux versions conduisent au même résultat.

La ligne de code suivante :

```
i++;
```

est équivalente à la ligne de code :

```
++i;
```

Les opérateurs peuvent être répartis en sept catégories.

1. Les opérateurs unaires

Opérateur	Action
-	Valeur négative
~	Complément à un
++	Incrémantation
--	Décrémantation
!	Négation

L'opérateur ! n'est utilisable que sur des variables de type boolean ou sur des expressions produisant un type boolean (comparaison).

2. Les opérateurs d'affectation

Le seul opérateur disponible dans cette catégorie est l'opérateur `=`. Il permet d'affecter à une variable une valeur. Le même opérateur est utilisé quel que soit le type de la variable (numérique, chaîne de caractères...).

Cet opérateur peut être combiné avec un opérateur arithmétique, logique ou binaire.

La syntaxe suivante :

```
x+=2;
```

est équivalente à :

```
x=x+2;
```

3. Les opérateurs arithmétiques

Les opérateurs arithmétiques permettent d'effectuer des calculs sur le contenu des variables.

Opérateur	Opération réalisée	Exemple	Résultat
<code>+</code>	Addition pour des valeurs numériques ou concaténation pour des chaînes	<code>6+4</code>	10
<code>-</code>	Soustraction	<code>12-6</code>	6
<code>*</code>	Multiplication	<code>3*4</code>	12
<code>/</code>	Division	<code>25/3</code>	8.3333333333
<code>%</code>	Modulo (reste de la division entière)	<code>25 mod 3</code>	1

4. Les opérateurs bit à bit

Ces opérateurs effectuent des opérations sur des entiers uniquement (Byte, Short, Integer, Long). Ils travaillent au niveau du bit sur les variables qu'ils manipulent.

Opérateur	Opération réalisée	Exemple	Résultat
<code>&</code>	Et binaire	<code>45 & 255</code>	45
<code> </code>	Ou binaire	<code>99 46</code>	111
<code>^</code>	Ou exclusif	<code>99 ^ 46</code>	77
<code>>></code>	Décalage vers la droite (division par 2)	<code>26>>1</code>	13
<code><<</code>	Décalage vers la gauche (multiplication par 2)	<code>26<<1</code>	52

5. Les opérateurs de comparaison

Les opérateurs de comparaison sont utilisés dans les structures de contrôle d'une application (`if, while...`). Ils renvoient une valeur de type `boolean` en fonction du résultat de la comparaison effectuée. Cette valeur sera ensuite utilisée par la structure de contrôle.

Opérateur	Opération réalisée	Exemple	Résultat
<code>==</code>	Egalité	<code>2 == 5</code>	false

!=	Inégalité	2 != 5	true
<	Inférieur	2 < 5	true
>	Supérieur	2 > 5	false
<=	Inférieur ou égal	2 <= 5	true
>=	Supérieur ou égal	2 >= 5	false
instanceof	Comparaison du type de la variable avec le type indiqué	O1 instanceof Client	True si la variable O1 référence un objet créé à partir de la classe Client ou d'une sous-classe

6. L'opérateur de concaténation

L'opérateur **+** déjà utilisé pour l'addition est également utilisé pour la concaténation de chaînes de caractères. Le fonctionnement de l'opérateur est déterminé par le type des opérandes. Si un des opérandes est du type **String** alors l'opérateur **+** effectue une concaténation avec éventuellement une conversion implicite de l'autre opérande en chaîne de caractères.

Petit inconvénient de l'opérateur **+**, il ne brille pas par sa rapidité pour les concaténations. En fait ce n'est pas réellement l'opérateur qui est en cause mais la technique utilisée par Java pour gérer les chaînes de caractères (elles ne peuvent pas être modifiées après création). Si vous avez de nombreuses concaténations à exécuter sur une chaîne, il est préférable d'utiliser la classe **StringBuffer**.

Exemple

```

long duree;
String lievre;
String tortue="";
long debut, fin;
debut = System.currentTimeMillis();
for (int i = 0; i <= 10000; i++)
{
    tortue = tortue + " " + i;
}
fin = System.currentTimeMillis();
duree = fin-debut;
System.out.println("durée pour la tortue : " + duree + "ms");
debut = System.currentTimeMillis();
StringBuffer sb = new StringBuffer();
for (int i = 0; i <= 10000; i++)
{
    sb.append(" ");
    sb.append(i);
}
lievre = sb.toString();
fin = System.currentTimeMillis();
duree = fin-debut;
System.out.println("durée pour le lièvre : " + duree + "ms");
if (lievre.equals(tortue))
{
    System.out.println("les deux chaînes sont identiques");
}

```

Résultat de la course :

```

durée pour la tortue : 953ms
durée pour le lièvre : 0ms
les deux chaînes sont identiques

```

Ce résultat se passe de commentaire !

7. Les opérateurs logiques

Les opérateurs logiques permettent de combiner les expressions dans des structures conditionnelles ou des structures de boucle.

Opérateur	Opération	Exemple	Résultat
&	Et logique	if ((test1) & (test2))	vrai si test1 et test2 est vrai
	Ou logique	if ((test1) (test2))	vrai si test1 ou test2 est vrai
^	Ou exclusif	if ((test1) ^ (test2))	vrai si test1 ou test2 est vrai mais pas si les deux sont vrais simultanément
!	Négation	if (! Test)	Inverse le résultat du test
&&	Et logique	if((test1) && (test2))	Idem et logique mais test2 ne sera évalué que si test1 est vrai
	Ou logique	if ((test1) (test2))	Idem ou logique mais test2 ne sera évalué que si test1 est faux

Il faudra être prudent avec les opérateurs && et || car l'expression que vous testerez en second (test2 dans notre cas) pourra parfois ne pas être exécutée. Si cette deuxième expression modifie une variable, celle-ci ne sera modifiée que dans les cas suivants :

- premier test vrai dans le cas du &&.
- premier test faux dans le cas du ||.

8. Ordre d'évaluation des opérateurs

Lorsque plusieurs opérateurs sont combinés dans une expression, ils sont évalués dans un ordre bien précis. Les incrémentations et décrémentations préfixées sont exécutées en premier puis les opérations arithmétiques, les opérations de comparaison, les opérateurs logiques et enfin les affectations.

Les opérateurs arithmétiques ont entre eux également un ordre d'évaluation dans une expression. L'ordre d'évaluation est le suivant :

- Négation (-)
- Multiplication et division (*, /)
- Division entière (\)
- Modulo (Mod)
- Addition et soustraction (+, -), concaténation de chaînes (+)

Si un ordre d'évaluation différent est nécessaire dans votre expression, il faut placer les portions à évaluer en priorité entre parenthèses comme dans l'expression suivante :

```
X= (z * 4) ^ (y * (a + 2));
```

Vous pouvez utiliser autant de niveaux de parenthèses que vous le souhaitez dans une expression. Il importe cependant que l'expression contienne autant de parenthèses fermantes que de parenthèses ouvrantes sinon le compilateur générera une erreur.

Les structures de contrôle

Les structures de contrôle permettent de modifier l'ordre d'exécution des instructions dans votre code. Deux types de structures sont disponibles :

- Les structures de décision : elles aiguilleront l'exécution du code en fonction des valeurs que pourra prendre une expression de test.
- Les structures de boucle : elles feront exécuter une portion de code un certain nombre de fois, jusqu'à ce qu'une condition soit remplie ou tant qu'une condition est remplie.

1. Structures de décision

Deux solutions sont possibles.

a. Structure if

Quatre syntaxes sont utilisables pour l'instruction `if`.

```
if (condition)instruction;
```

Si la condition est vraie alors l'instruction est exécutée. La condition doit être une expression qui, une fois évaluée, doit fournir un boolean `true` ou `false`. Avec cette syntaxe, seule l'instruction située après le `if` sera exécutée si la condition est vraie. Pour pouvoir faire exécuter plusieurs instructions en fonction d'une condition il faut utiliser la syntaxe ci-après.

```
if (condition)
{
    Instruction 1;
    ...
    Instruction n;
}
```

Dans ce cas le groupe d'instructions situé entre les accolades sera exécuté si la condition est vraie.

Vous pouvez également spécifier une ou plusieurs instructions qui elles seront exécutées si la condition est fausse.

```
if (condition)
{
    Instruction 1;
    ...
    Instruction n;
}
else
{
    Instruction 1;
    ...
    Instruction n;
}
```

Vous pouvez également imbriquer les conditions avec la syntaxe.

```
if (condition1)
{
    Instruction 1
    ...
    Instruction n
}
else if (Condition 2)
{
    Instruction 1
    ...
}
```

```

        Instruction n
    }
else if (Condition 3)
{
    Instruction 1
    ...
    Instruction n
}
else
{
    Instruction 1
    ...
    Instruction n
}

```

Dans ce cas, on teste la première condition. Si elle est vraie alors le bloc de code correspondant est exécuté sinon on teste la suivante et ainsi de suite. Si aucune condition n'est vérifiée, le bloc de code spécifié après le `else` est exécuté. L'instruction `else` n'est pas obligatoire dans cette structure. Dans ce cas, il se peut qu'aucune instruction ne soit exécutée si aucune des conditions n'est vraie.

Il existe également un opérateur conditionnel permettant d'effectuer un `if ... else` en une seule instruction.

```
condition ? expression1 : expression2;
```

Cette syntaxe est équivalente à celle-ci :

```
If (condition)
expression1;
else
expression2;
```

b. Structure switch

La structure `switch` permet un fonctionnement équivalent mais offre une meilleure lisibilité du code. La syntaxe est la suivante :

```

Switch (expression)
{
    Case valeur1:
        Instruction 1
        ...
        Instruction n
        Break;
    Case valeur2:
        Instruction 1
        ...
        Instruction n
        Break;
    Default:
        Instruction 1
        ...
        Instruction n
}

```

La valeur de l'expression est évaluée au début de la structure (par le `switch`) puis la valeur obtenue est comparée avec la valeur spécifiée dans le premier `case`.

Si les deux valeurs sont égales, alors le bloc de code 1 est exécuté.

Sinon, la valeur obtenue est comparée avec la valeur du `case` suivant, s'il y a correspondance, le bloc de code est exécuté et ainsi de suite jusqu'au dernier `case`.

Si aucune valeur concordante n'est trouvée dans les différents `case` alors le bloc de code spécifié dans le `default` est exécuté. Chacun des blocs de code doit se terminer par l'instruction `break`.

Si ce n'est pas le cas l'exécution se poursuivra par le bloc de code suivant jusqu'à ce qu'une instruction `break` soit rencontrée ou jusqu'à la fin de la structure `switch`. Cette solution peut être utilisée pour pouvoir exécuter un même bloc de code pour différentes valeurs testées.

La valeur à tester peut être contenue dans une variable mais elle peut également être le résultat d'un calcul. Dans ce cas, le calcul n'est effectué qu'une seule fois au début du `switch`. Le type de la valeur testée peut être numérique entière, caractère, chaîne de caractères ou énumération. Il faut bien sûr que le type de la variable testée corresponde au type des valeurs dans les différents `case`.

Si l'expression est de type chaîne de caractères, la méthode `equals` est utilisée pour vérifier l'égalité avec les valeurs des différents `case`. La comparaison fait donc une distinction entre minuscules et majuscules.

```
BufferedReader br;
    br=new BufferedReader(new InputStreamReader(System.in));
    String reponse="";
    reponse=br.readLine();
    switch (reponse)
    {
        case "oui":
        case "OUI":
            System.out.println("réponse positive");
            break;
        case "non":
        case "NON":
            System.out.println("réponse négative");
            break;
        default:
            System.out.println("mauvaise réponse");
    }
```

2. Les structures de boucle

Trois structures sont à notre disposition :

```
while (condition)
do ... while (condition)
for
```

Elles ont toutes pour but de faire exécuter un bloc de code un certain nombre de fois en fonction d'une condition.

a. Structure while

Cette structure exécute un bloc de façon répétitive tant que la condition est `true`.

```
while (condition)
{
    Instruction 1
    ...
    Instruction n
}
```

La condition est évaluée avant le premier passage dans la boucle. Si elle est `false` à cet instant alors le bloc de code n'est pas exécuté. Après chaque exécution du bloc de code la condition est à nouveau évaluée pour vérifier si une nouvelle exécution du bloc de code est nécessaire. Il est recommandé que l'exécution du bloc de code contienne une ou plusieurs instructions susceptibles de faire évoluer la condition. Si ce n'est pas le cas la boucle s'exécutera sans fin. Il ne faut surtout pas placer de caractère ; après le `while` car dans ce cas, le bloc de code n'est plus associé à la boucle.

```
int i=0;
while (i<10)
{
    System.out.println(i);
```

```
i++;
```

```
}
```

b. Structure do ... while

```
do
{
    Instruction 1
    ...
    Instruction n
}
while (condition);
```

Cette structure a un fonctionnement identique à la précédente sauf que la condition est examinée après l'exécution du bloc de code. Elle nous permet de garantir que le bloc de code sera exécuté au moins une fois puisque la condition sera testée pour la première fois après la première exécution du bloc de code. Si la condition est `true` alors le bloc est exécuté une nouvelle fois jusqu'à ce que la condition soit `false`. Vous devez faire attention à ne pas oublier le point-virgule après le `while` sinon le compilateur détecte une erreur de syntaxe.

```
do
{
    System.out.println(i);
    i++;
}
while(i<10);
```

c. Structure for

Lorsque vous connaissez le nombre d'itérations à réaliser dans une boucle il est préférable d'utiliser la structure `for`. Pour pouvoir utiliser cette instruction, une variable de compteur doit être déclarée. Cette variable peut être déclarée dans la structure `for` ou à l'extérieur, elle doit dans ce cas être déclarée avant la structure `for`.

La syntaxe générale est la suivante :

```
for(initialisation;condition;instruction d'itération)
{
    Instruction 1
    ...
    Instruction n
}
```

La partie `initialisation` est exécutée une seule fois lors de l'entrée dans la boucle. La partie `condition` est évaluée lors de l'entrée dans la boucle puis à chaque itération. Le résultat de l'évaluation de la condition détermine si le bloc de code est exécuté, il faut pour cela que la condition soit évaluée comme `true`. Après l'exécution du bloc de code l'instruction d'itération est à son tour exécutée. Puis la condition est à nouveau testée et ainsi de suite tant que la condition est évaluée comme `true`.

Voici ci-dessous deux boucles `for` en action pour afficher une table de multiplication.

```
int k;
for(k=1;k<10;k++)
{
    for (int l = 1; l < 10; l++)
    {
        System.out.print(k * l + "\t");
    }
    System.out.println();
}
```

Nous obtenons le résultat suivant :

1	2	3	4	5	6	7	8	9
2	4	6	8	10	12	14	16	18
3	6	9	12	15	18	21	24	27
4	8	12	16	20	24	28	32	36
5	10	15	20	25	30	35	40	45
6	12	18	24	30	36	42	48	54
7	14	21	28	35	42	49	56	63
8	16	24	32	40	48	56	64	72
9	18	27	36	45	54	63	72	81

Une autre syntaxe de la boucle `for` permet de faire exécuter un bloc de code pour chaque élément contenu dans un tableau ou dans une instance de classe implémentant l'interface `Iterable`. La syntaxe générale de cette instruction est la suivante :

```
for (type variable : tablo)
{
    Instruction 1
    ...
    Instruction n
}
```

Il n'y a pas de notion de compteur dans cette structure puisqu'elle effectue elle-même les itérations sur tous les éléments présents dans le tableau ou la collection.

La variable déclarée dans la structure sert à extraire un à un les éléments du tableau ou de la collection pour que le bloc de code puisse les manipuler. Il faut bien sûr que le type de la variable soit compatible avec le type des éléments stockés dans le tableau ou la collection. La variable doit obligatoirement être déclarée dans la structure `for` et non à l'extérieur. Elle ne sera utilisable qu'à l'intérieur de la structure. Par contre vous n'avez pas à vous soucier du nombre d'éléments car la structure est capable de gérer elle-même le déplacement dans le tableau ou la collection. Voici un petit exemple pour clarifier la situation !

Avec une boucle classique :

```
String[] tablo={"rouge","vert","bleu","blanc"};
int cpt;
for (cpt = 0; cpt < tablo.length; cpt++)
{
    System.out.println(tablo[cpt]);
}
```

Avec la boucle `for` d'itération :

```
String[] tablo={"rouge","vert","bleu","blanc"};
for (String s : tablo)
{
    System.out.println(s);
}
```



Le code placé à l'intérieur de cette structure `for` ne doit pas modifier le contenu de la collection.

Il est donc interdit d'ajouter ou de supprimer des éléments pendant le parcours de la collection. Le problème ne se pose pas avec un tableau. La taille d'un tableau étant fixe, il est bien impossible d'y ajouter ou d'y supprimer un élément. Le code suivant met en évidence cette limitation lors du parcours d'une `ArrayList`. L'ajout d'un élément à l'`ArrayList` en cours d'itération déclenche une exception de type `ConcurrentModificationException`.

```
ArrayList<String> lst;
st=new ArrayList<String>();
lst.add("client 1");
lst.add("client 2");
lst.add("client 3");
```

```

lst.add("client 5");

for(String st:lst)
{
    System.out.println(st);
    if(st.endsWith("3"))
    {
        lst.add("client 4");
    }
}

```

d. Interruption d'une structure de boucle

Trois instructions peuvent modifier le fonctionnement normal des structures de boucle.

break

Si cette instruction est placée à l'intérieur du bloc de code d'une structure de boucle elle provoque la sortie immédiate de ce bloc de code. L'exécution se poursuit par l'instruction placée après le bloc de code. Cette instruction doit en général être exécutée de manière conditionnelle, sinon les instructions situées après à l'intérieur de la boucle ne seront jamais exécutées.

Dans le cas de boucles imbriquées, il est possible d'utiliser l'instruction `break` associée avec une étiquette. L'exemple de code ci-dessous effectue le parcours d'un tableau à deux dimensions et s'arrête dès qu'une case contenant la valeur 0 est rencontrée.

```

int[][] points = {
    { 10,10, },
    { 0,10 },
    { 45,24 }};
int x=0,y=0;
boolean trouve=false;
recherche:
    for (x = 0; x <points.length; x++)
    {
        for (y = 0; y < points[x].length;y++)
        {
            if (points[x][y] == 0)
            {
                trouve = true;
                break recherche;
            }
        }
    }
if (trouve)
{
    System.out.println("resultat trouvé dans la case "
+ x + "-" + y);
}
else
{
    System.out.println("recherche infructueuse");
}

```

Continue

Cette instruction permet d'interrompre l'exécution de l'itération courante d'une boucle et de continuer l'exécution à l'itération suivante après vérification de la condition de sortie de boucle. Comme pour l'instruction `break` elle doit être exécutée de manière conditionnelle et accepte également l'utilisation d'une étiquette.

Voici un exemple de code utilisant une boucle sans fin et ses deux instructions pour afficher les nombres impairs jusqu'à ce que l'utilisateur saisisse un retour chariot.

```

import java.io.IOException;
public class TestStructures {
    static boolean stop;
    public static void main(String[] args)
    {
        new Thread()
        {
            public void run()
            {
                int c;
                try
                {
                    c=System.in.read();
                    stop=true;
                }
                catch (IOException e)
                {
                    e.printStackTrace();
                }
            }
        }.start();
        long compteur=0;
        while(true)
        {
            compteur++;
            if (compteur%2==0)
                continue;
            if (stop)
                break;
            System.out.println(compteur);
        }
    }
}

```

Return

L'instruction `return` est utilisée pour sortir immédiatement de la méthode en cours d'exécution et poursuivre l'exécution par l'instruction suivant celle qui a appelé cette méthode. Si elle est placée dans une structure de boucle, elle provoque bien sûr la sortie immédiate de la boucle puis de la méthode dans laquelle se trouve la boucle. L'utilisation de cette instruction dans une fonction dont le type de retour est autre que `void` oblige à fournir à l'instruction `return` une valeur compatible avec le type de retour de la fonction.

Exercices

Exercice 1

Créer un tableau de dix chaînes de caractères puis remplir ce tableau avec des adresses e-mail. Calculer ensuite, à partir des informations présentes dans le tableau, la part de marché de chacun des fournisseurs d'accès.

Indice : dans une adresse e-mail, le nom du fournisseur d'accès est la partie située après le caractère @ de l'adresse e-mail.

Exercice 2

Générer trois nombres aléatoires compris entre 0 et 1000, puis vérifier si vous avez deux nombres pairs suivis par un nombre impair. Si ce n'est pas le cas, recommencer jusqu'à ce vous ayez la combinaison pair, pair, impair. Afficher ensuite le nombre d'essais nécessaires pour obtenir cette combinaison.

Indice : la classe Math propose la méthode statique random qui génère un nombre aléatoire compris entre 0 et 1.

Exemple : double nb=Math.random();

Exercice 3

Générer un nombre aléatoire compris entre 0 et 1000. Demander ensuite à l'utilisateur de deviner le nombre choisi par l'ordinateur. Il doit saisir un nombre compris entre 0 et 1000 lui aussi. Comparer le nombre saisi avec celui choisi par l'ordinateur et afficher sur la console « c'est plus » ou « c'est moins » selon le cas. Recommencer jusqu'à ce que l'utilisateur trouve le bon nombre. Afficher alors le nombre d'essais nécessaires pour trouver la bonne réponse.

Indice : pour récupérer les caractères saisis au clavier, nous avons à notre disposition le flux System.in. Malheureusement, celui-ci ne propose que des fonctions rudimentaires pour la récupération des saisies de l'utilisateur (lecture caractère par caractère). Pour une utilisation plus confortable, il vaut mieux utiliser un objet Scanner. Nous aurons ainsi à notre disposition une série de fonctions permettant la récupération d'entiers, de float, de chaînes de caractères... Ces fonctions sont nommées nextxxxx où xxxx doit être remplacé par le type de données que l'on souhaite obtenir, par exemple nextInt pour un entier, nextLine pour une chaîne de caractères, etc.

```
String chaine;
Scanner sc;
sc=new Scanner(System.in);
chaine=sc.nextLine();
```

Exercice 4

Ajouter au jeu de l'exercice 3 l'affichage du temps mis par l'utilisateur pour obtenir la bonne réponse.

Corrections

Correction de l'exercice 1

Décomposons le problème.

Il faut en premier lieu créer et remplir le tableau contenant les adresses e-mail à traiter. C'est le tableau `adresses` qui joue ce rôle et qui est initialisé des lignes 13 à 23.

Il faut ensuite extraire de ces adresses la partie nom de fournisseur. Le résultat est stocké dans un nouveau tableau `listeFournisseurs`. Ce traitement est effectué par les lignes 25 à 29.

Ce tableau est ensuite analysé pour extraire les différents noms de fournisseur et pour les stocker dans un nouveau tableau `nomsFournisseurs`. Les lignes 31 à 48 effectuent ce travail en parcourant le tableau `listeFournisseurs` et en vérifiant si chaque élément est déjà présent dans le tableau `nomsFournisseurs` grâce à la fonction `rechercheFournisseur`. Elle retourne l'index de la case où le fournisseur a été trouvé ou -1 en cas d'échec de la recherche. Dans ce cas, le nom du fournisseur est ajouté au tableau `nomsFournisseurs`. Le contenu de la case correspondante du tableau `nbClient` est ensuite incrémenté.

Pour terminer, on parcourt le tableau `nomsFournisseurs` et on récupère dans le tableau `nbClient` le nombre de fois où ce fournisseur est apparu puis on calcule et on affiche le pourcentage (lignes 49 à 63).

```
1. package exercices.chapitre2.exercice1;
2.
3. public class Principale
4. {
5.     static String[] adresses;
6.     static String[] listeFournisseurs;
7.     static String[] nomsFournisseurs;
8.     static int[] nbClient;
9.     static int position;
10.    static String fournisseur;
11.    public static void main(String[] args)
12.    {
13.        adresses=new String[10];
14.        adresses[0]="jpp@sfr.fr";
15.        adresses[1]="tom@gmail.com";
16.        adresses[2]="fred@sfr.fr";
17.        adresses[3]="victor@sfr.fr";
18.        adresses[4]="chris@sfr.fr";
19.        adresses[5]="robert@orange.fr";
20.        adresses[6]="paul@sfr.fr";
21.        adresses[7]="lise@gmail.com";
22.        adresses[8]="thierry@eni.fr";
23.        adresses[9]="marie@eni.fr";
24.
25.        listeFournisseurs =new String[10];
26.        for (int i=0;i<adresses.length;i++)
27.        {
28.            listeFournisseurs[i]=adresses[i].substring(adresses[i].indexOf
('@@')+1);
29.        }
30.
31.        nbClient=new int[10];
32.        nomsFournisseurs=new String[10];
33.        for(int i=0;i<listeFournisseurs.length;i++)
34.        {
35.            int resultat;
36.            resultat=rechercheFournisseur(listeFournisseurs[i]);
37.            if(resultat==-1)
38.            {
39.                nomsFournisseurs[position]=listeFournisseurs[i];
40.                nbClient[position]++;
41.                position++;
42.            }
```

```

43.         else
44.         {
45.             nbClient[resultat]++;
46.         }
47.
48.     }
49.     for(int i=0;i<position;i++)
50.     {
51.         System.out.println(nomsFournisseurs[i] + " : "
" + (double)nbClient[i]/listeFournisseurs.length*100 + "%");
52.     }
53. }
54.
55. static int rechercheFournisseur(String nom)
56. {
57.     for(int i=0;i<nomsFournisseurs.length;i++)
58.     {
59.         if(nomsFournisseurs[i]!=null &&
nomsFournisseurs[i].equals(nom))
60.         {
61.             return i;
62.         }
63.     }
64.     return -1;
65. }
66. }
```

Correction de l'exercice 2

Pour cet exercice, nous devons utiliser la fonction **random** pour obtenir un nombre aléatoire. Cette fonction retourne un nombre décimal aléatoire compris entre 0 et 1. Comme nous voulons obtenir des nombres entiers, il suffit de multiplier la valeur renournée par la fonction par 1000 et ensuite transformer le résultat en nombre entier. C'est le rôle des lignes 11 à 13. Le compteur de nombre d'essais est ensuite incrémenté puis les trois nombres générés sont ensuite affichés (lignes 14 et 15). Cet ensemble est placé dans une boucle `do ... while` pour recommencer un nouveau tirage tant que nos exigences ne sont pas remplies. Si la condition indiquée dans l'instruction `while` est évaluée comme vraie, alors la boucle effectue une nouvelle itération. Nous souhaitons arrêter les tirages au sort lorsque nous avons le premier nombre pair, le deuxième nombre pair et le troisième nombre impair, il nous faut donc inverser cette condition pour continuer les itérations. Une fois inversée, celle-ci devient donc : on continue les tirages si le premier nombre est impair ou si le deuxième nombre est impair ou si le troisième nombre est pair.

Pour vérifier la parité d'un nombre, on le divise par 2 puis on récupère le reste de la division. Si ce reste est égal à 0, le nombre est pair, sinon il est impair. Cette vérification est effectuée par la ligne 17. La dernière ligne de code affiche simplement le nombre de tirages au sort.

```

1. package exercices.chapitre2.exercice2;
2.
3. public class Principale
4. {
5.     public static void main(String[] args)
6.     {
7.         int compteur=0;
8.         int nb1,nb2,nb3;
9.         do
10.            {
11.                nb1=(int)(Math.random()*1000);
12.                nb2=(int)(Math.random()*1000);
13.                nb3=(int)(Math.random()*1000);
14.                compteur++;
15.                System.out.println("nombre 1:" + nb1 +
" nombre 2:" + nb2 + " nombre 3:" + nb3);
16.            }
17.            while(nb1 % 2==1 || nb2 % 2==1 || nb3 % 2==0);
18.            System.out.println("Résultat obtenu en "
+ compteur + " essai(s)");
19.        }
```

Correction de l'exercice 3

Le code de cet exercice débute, comme c'est très souvent le cas, par les initialisations. Les lignes 9 à 14 initialisent le nombre secret et l'objet Scanner permettant la récupération des saisies de l'utilisateur.

La boucle `do...while` gère ensuite la saisie d'un entier par l'utilisateur puis la comparaison avec le nombre secret et enfin l'affichage du message correspondant au résultat de cette comparaison. Cette boucle est à nouveau exécutée tant que le nombre saisi est différent du nombre secret.

À la sortie de la boucle le score est affiché.

```

1. package exercices.chapitre2.exercice3;
2.
3. import java.util.Scanner;
4.
5. public class Principale
6. {
7.     public static void main(String[] args)
8.     {
9.         int nbEssais=0;
10.        int nombre;
11.        int nbSaisi;
12.        nombre=(int)(Math.random()*1000);
13.        Scanner sc;
14.        sc=new Scanner(System.in);
15.        do
16.        {
17.            nbSaisi=sc.nextInt();
18.            nbEssais++;
19.            if(nbSaisi<nombre)
20.            {
21.                System.out.println("c'est plus");
22.            }
23.            if(nbSaisi>nombre)
24.            {
25.                System.out.println("c'est moins");
26.            }
27.        } while (nombre!=nbSaisi);
28.        System.out.println("Bravo vous avez trouve en "
+ nbEssais + " essai(s)");
29.    }
30. }
```

Correction de l'exercice 4

Pour faire évoluer l'application précédente, nous ajoutons simplement l'initialisation de la variable **debut** avec l'heure courante avant le début de la boucle (ligne 21). À la fin de la boucle (donc du jeu) nous faisons la même chose avec la variable **fin** (ligne 35). La durée de la partie est la différence entre ces deux heures. Elle est calculée grâce à la ligne 36. Il reste juste ensuite à la convertir en heure, minute, seconde et à afficher le résultat (lignes 38 à 43).

```

1. package exercices.chapitre2.exercice4;
2.
3. import java.time.Duration;
4. import java.time.LocalDateTime;
5. import java.time.OffsetTime;
6. import java.util.Scanner;
7.
8. public class Principale
9. {
10.     public static void main(String[] args)
11.     {
12.         int nbEssais=0;
```

```
13.         int nombre;
14.         int nbSaisi;
15.         OffsetTime debut;
16.         OffsetTime fin;
17.         Duration temps;
18.         nombre=(int)(Math.random()*1000);
19.         Scanner sc;
20.         sc=new Scanner(System.in);
21.         debut=OffsetTime.now();
22.         do
23.         {
24.             nbSaisi=sc.nextInt();
25.             nbEssais++;
26.             if(nbSaisi<nombre)
27.             {
28.                 System.out.println("c'est plus");
29.             }
30.             if(nbSaisi>nombre)
31.             {
32.                 System.out.println("c'est moins");
33.             }
34.         } while (nombre!=nbSaisi);
35.         fin=OffsetTime.now();
36.         temps=Duration.between(debut,fin);
37.         LocalTime duree;
38.         duree=LocalTime.ofSecondOfDay(temps.getSeconds());
39.         System.out.println("Bravo vous avez trouve en " + nbEssais +
40.             " essai(s) et " + duree.getHour()+" heure(s) " +
41.             duree.getMinute() + " minute(s) " +
42.             duree.getSecond()+" seconde(s)");
43.     }
44. }
```

Introduction

Avec Java, la notion d'objet est omniprésente et nécessite un minimum d'apprentissage. Nous allons donc voir dans un premier temps les principes de la programmation objet et le vocabulaire associé, puis nous verrons comment mettre cela en application avec Java.

Dans un langage procédural classique, le fonctionnement d'une application est réglé par une succession d'appels aux différentes procédures et fonctions disponibles dans le code. Ces procédures et fonctions sont chargées de manipuler les données de l'application qui sont représentées par les variables de l'application. Il n'y a aucun lien entre les données et le code qui les manipule. Dans un langage objet on va au contraire essayer de regrouper le code. Ce regroupement est appelé une classe. Une application développée avec un langage objet est donc constituée de nombreuses classes représentant les différents éléments manipulés par l'application. Les classes vont décrire les caractéristiques de chacun des éléments. C'est ensuite l'assemblage de ces éléments qui va permettre le fonctionnement de l'application.

Ce principe est largement utilisé dans d'autres domaines que l'informatique. Dans l'industrie automobile, par exemple, il n'existe certainement pas chez aucun constructeur un plan complet décrivant les milliers de pièces constituant un véhicule. Par contre, chaque sous-ensemble d'un véhicule peut être représenté par un plan spécifique (le châssis, la boîte de vitesse, le moteur...). Chaque sous-ensemble est également décomposé jusqu'à la pièce élémentaire (un boulon, un piston, un pignon...). C'est l'assemblage de tous ces éléments qui permet la fabrication d'un véhicule.

En fait ce n'est pas l'assemblage des plans qui permet la construction du véhicule, mais l'assemblage des pièces fabriquées à partir de ces plans. Dans une application informatique c'est l'assemblage des objets créés à partir des classes qui va permettre le fonctionnement de l'application. Les deux termes classe et objet sont souvent confondus mais ils représentent des notions bien distinctes. La classe décrit la structure d'un élément alors que l'objet représente un exemplaire créé sur le modèle de cette structure. Après sa création, un objet est indépendant des autres objets construits à partir de la même classe. Par exemple une portière de voiture pourra après fabrication être peinte d'une couleur différente des autres portières fabriquées selon le même plan.

Par contre si le plan vient à être modifié, toutes les portières fabriquées après la modification du plan bénéficieront des changements apportés au plan (avec le risque de ne plus être compatibles avec les anciennes versions).

Les classes sont constituées de champs et de méthodes. Les champs représentent les caractéristiques des objets. Ils sont représentés par des variables et il est donc possible de lire leur contenu ou de leur affecter une valeur directement. Le robot qui va peindre une portière va modifier le champ couleur de cette portière. Les méthodes représentent les actions qu'un objet peut effectuer. Elles sont mises en œuvre par la création de procédures ou de fonctions dans une classe.

Ceci n'est qu'une facette de la programmation orientée objet. Trois autres concepts sont également fondamentaux :

- L'encapsulation
- L'héritage
- Le polymorphisme

L'encapsulation consiste à cacher les éléments qui ne sont pas nécessaires pour l'utilisation d'un objet. Cette technique permet de garantir que l'objet sera correctement utilisé. C'est un principe qui est aussi largement utilisé dans d'autres domaines que l'informatique. Pour reprendre l'exemple de l'industrie automobile, savez-vous comment fonctionne la boîte de vitesse de votre voiture ?

Pour changer de vitesse allez-vous directement modifier la position des différents engrenages ? Heureusement que non. Les constructeurs ont en fait prévu des solutions plus pratiques pour la manipulation de la boîte de vitesse.

Les éléments d'une classe visibles de l'extérieur de la classe sont appelés l'interface de la classe. Dans le cas de notre voiture, le levier de vitesse constitue l'interface de la boîte de vitesse. C'est par son intermédiaire que l'on peut agir sans risque sur les mécanismes internes de la boîte de vitesse.

L'héritage permet la création d'une nouvelle classe à partir d'une classe existante. La classe servant de

modèle est appelée classe de base. La classe ainsi créée hérite des caractéristiques de sa classe de base. Elle peut aussi être personnalisée en y ajoutant des caractéristiques supplémentaires. Les classes créées à partir d'une classe de base sont appelées classes dérivées. Ce principe est bien sûr aussi utilisé dans le monde industriel. La boîte de vitesse de votre voiture comporte certainement cinq rapports. Les ingénieurs qui ont conçu cette pièce ne sont certainement pas repartis de zéro. Ils ont repris le plan de la génération précédente (quatre rapports) et y ont ajouté des éléments. De même que les ingénieurs qui réfléchissent déjà à la boîte de vitesse à six rapports de votre future voiture vont repartir de la version précédente.

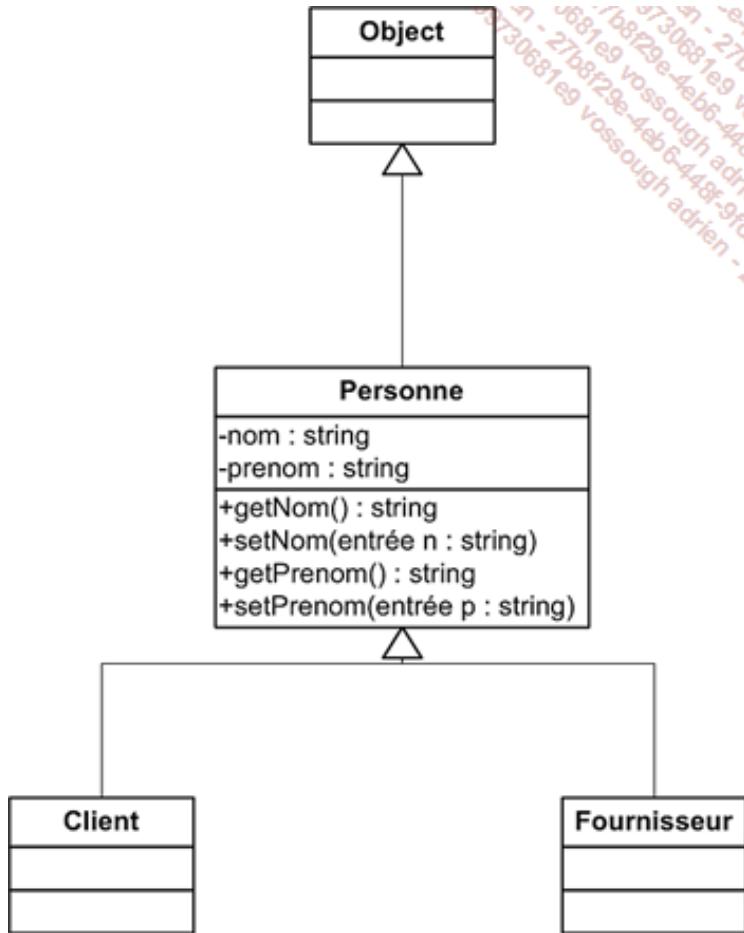
Le polymorphisme est une autre notion importante de la programmation orientée objet. Par son intermédiaire, il est possible d'utiliser plusieurs classes de manière interchangeable même si le fonctionnement interne de ces classes est différent. Si vous savez changer de vitesse sur une voiture Peugeot, vous savez également comment le faire sur une voiture Renault et pourtant les deux types de boîte de vitesse ne sont pas conçus de la même façon.

Deux autres concepts sont également associés au polymorphisme. La surcharge et le masquage de méthodes. La surcharge est utilisée pour concevoir dans une classe des méthodes ayant le même nom mais ayant un nombre de paramètres différent ou des types de paramètres différents. Le masquage est utilisé lorsque dans une classe dérivée, on souhaite modifier le fonctionnement d'une méthode dont on a hérité. Le nombre et le type des paramètres restant identiques à ceux définis dans la classe de base.

Voici mis en place les quelques principes de base de la programmation objet. Le sujet de cet ouvrage n'étant pas la mécanique automobile nous allons tout de suite voir la mise en œuvre de ces principes avec le langage Java.

Mise en œuvre avec Java

Dans le reste de ce chapitre, nous allons travailler sur la classe Personne dont la représentation UML (*Unified Modeling Language*) est disponible ci-dessous.



UML est un langage graphique dédié à la représentation des concepts de programmation orientée objet. Pour plus d'informations sur ce langage, vous pouvez consulter l'ouvrage UML 2 dans la collection Ressources Informatiques des Éditions ENI.

1. Création d'une classe

La création d'une classe passe par la déclaration de la classe elle-même et de tous les éléments la constituant.

a. Déclaration de la classe

La déclaration d'une classe se fait en utilisant le mot clé `class` suivi du nom de la classe puis d'un bloc de code délimité par les caractères `{` et `}`. Dans ce bloc de code, on trouve des déclarations de variables qui seront les champs de la classe et des fonctions qui seront les méthodes de la classe. Plusieurs mots clés peuvent être ajoutés pour modifier les caractéristiques de la classe. La syntaxe générale de déclaration d'une classe est donc la suivante.

```
[liste de modificateurs] class NomDeLaClasse [extends
NomDeLaClasseDeBase]
[implements NomDeInterface1,NomDeInterface2,...]
{
    Code de la classe
}
```

Les signes `[` et `]` sont utilisés ici pour indiquer le caractère optionnel de l'élément. Ils ne doivent pas être

utilisés dans le code de déclaration d'une classe.

Les modificateurs permettent de déterminer la visibilité de la classe et comment vous pouvez l'utiliser. Voici la liste des modificateurs disponibles :

`public` : indique que la classe peut être utilisée par toutes les autres classes. Sans ce modificateur la classe ne sera utilisable que par les autres classes faisant partie du même package.

`abstract` : indique que la classe est abstraite et ne peut donc pas être instanciée. Elle ne peut être utilisée que comme classe de base dans une relation d'héritage. En général, dans ce genre de classe seules les déclarations de méthodes sont définies, et il faudra écrire le contenu des méthodes dans les classes dérivées.

`final` : la classe ne peut pas être utilisée comme classe de base dans une relation d'héritage et peut uniquement être instanciée.

La signification des mots clés `abstract` et `final` étant contradictoire leur utilisation simultanée est bien sûr interdite.

Pour indiquer que votre classe récupère les caractéristiques d'une autre classe par une relation d'héritage, vous devez utiliser le mot clé `extends` suivi du nom de la classe de base. Vous pouvez également implémenter dans votre classe une ou plusieurs interfaces en utilisant le mot `implements` suivi de la liste des interfaces implémentées. Ces deux notions seront vues en détail plus loin dans ce chapitre.

Le début de la déclaration de notre classe `Personne` est donc le suivant :

```
public class Personne
{
}
```

Ce code doit obligatoirement être saisi dans un fichier ayant le même nom que la classe et l'extension `.java`.

b. Création des champs

Intéressons-nous maintenant au contenu de notre classe. Nous devons créer les différents champs de la classe. Pour cela, il suffit de déclarer des variables à l'intérieur du bloc de code de la classe en indiquant la visibilité de la variable, son type et son nom.

```
[private | protected | public] typeDeLaVariable nomDeLaVariable;
```

La visibilité de la variable répond aux règles suivantes :

`private` : la variable n'est accessible que dans la classe où elle est déclarée.

`protected` : la variable est accessible dans la classe où elle est déclarée, dans les autres classes faisant partie du même package et dans les classes héritant de la classe où elle est déclarée.

`public` : la variable est accessible à partir de n'importe où.

Si aucune information n'est fournie concernant la visibilité, la variable est accessible à partir de la classe où elle est déclarée et des autres classes faisant partie du même package. Lorsque que vous choisissez la visibilité d'une variable vous devez autant que possible respecter le principe d'encapsulation et limiter au maximum la visibilité des variables. L'idéal étant de toujours avoir des variables `private` ou `protected` mais jamais `public`.

La variable doit également avoir un type. Il n'y a pas de limitation concernant le type d'une variable et vous pouvez donc utiliser aussi bien les types de base du langage Java tels que `int`, `float`, `char`, ... que des types d'objets.

Le nom de la variable doit quant à lui simplement respecter les règles de nommage (pas d'utilisation de mot clé du langage).

La classe Personne prend donc maintenant la forme suivante :

```
public class Personne
{
    private String nom;
    private String prenom;
    private LocalDate date_nais;
}
```

c. Cration de mthodes

Les mthodes sont simplement des fonctions dfinies  l'intrieur d'une classe. Elles sont en gnral utilises pour manipuler les champs de la classe. La syntaxe gnrale de dclaration d'une mthode est dcrite ci-dessous.

```
[modificateurs] typeDeRetour nomDeLaMethode ([listeDesParamtres])
[throws listeException]
{
```

```
}
```

Les modificateurs suivants sont disponibles :

`private` : indique que la mthode ne peut tre utilise que dans la classe ou elle est dfinie.

`protected` : indique que la mthode peut tre utilise dans la classe ou elle est dfinie, dans les sous-classes de cette classe et dans les classes faisant partie du mme package.

`public` : indique que la mthode peut tre utilise depuis n'importe quelle autre classe.

Si aucun de ces mots cls n'est utilis alors la visibilit sera limite au package dans lequel la classe est dfinie.

`static` : indique que la mthode est une mthode de classe.

`abstract` : indique que la mthode est abstraite et qu'elle ne contient pas de code. La classe ou elle est dfinie doit elle aussi tre abstraite.

`final` : indique que la mthode ne peut pas tre substitue dans une sous-classe.

`native` : indique que le code de la mthode se trouve dans un fichier externe crit dans un autre langage.

`synchronized` : indique que la mthode ne peut tre excute que par un seul thread  la fois.

Le type de retour peut tre n'importe quel type de donnes, type de base du langage ou type objet. Si la mthode n'a pas d'information  renvoyer vous devez utiliser le mot cl `void` en remplacement du type de retour.

La liste des paramtres est identique  une liste de dclaration de variables. Il faut spcifier le type du paramtre et le nom du paramtre. Si plusieurs paramtres sont attendus il faut sparer leur dclaration par une virgule. Mme si aucun paramtre n'est attendu, les parenthses sont tout de mme obligatoires.

Le mot cl `throws` indique la liste des exceptions que cette mthode peut dclencher lors de son excution.

Ajoutons deux mthodes  notre classe Personne.

```
public class Personne
{
    private String nom;
    private String prenom;
    private LocalDate date_nais;

    public long calculAge()
```

```

    {
        return date_nais.until(LocalDate.now(), ChronoUnit.YEARS);
    }

    public void affichage()
    {
        System.out.println("nom : " + nom);
        System.out.println("prénom : " + prenom);
        System.out.println("âge : " + calculAge());
    }
}

```

Dans certains langages de programmation, il n'est pas possible d'avoir plusieurs fonctions portant le même nom. Le langage Java comme beaucoup de langages objet permet de contourner le problème en créant des fonctions surchargées. Une fonction surchargée porte le même nom qu'une autre fonction de la classe mais présente une signature différente. Les informations suivantes sont prises en compte pour déterminer la signature d'une fonction :

- le nom de la fonction
- le nombre de paramètres attendus par la fonction
- le type des paramètres.

Pour pouvoir créer une fonction surchargée il faut qu'au moins un de ses éléments change par rapport à une fonction déjà existante. Le nom de la fonction devant rester le même pour pouvoir réellement parler de surcharge, nous ne pouvons donc agir que sur le nombre de paramètres ou leur type. Nous pouvons par exemple ajouter la fonction suivante à la classe Personne :

```

public void affichage(boolean français)
{
    if (français)
    {
        System.out.println("nom : " + nom);
        System.out.println("prénom : " + prenom);
        System.out.println("âge : " + calculAge());
    }
    else
    {
        System.out.println("name : " + nom);
        System.out.println("first name : " + prenom);
        System.out.println("age : " + calculAge());
    }
}

```

Elle possède effectivement une signature différente de la première fonction affichage que nous avons créée puisqu'elle attend un paramètre de type boolean. Si maintenant nous ajoutons la fonction suivante le compilateur refuse la compilation du code.

```

public void affichage(boolean majuscule)
{
    if (majuscule)
    {
        System.out.println("nom : " + nom.toUpperCase());
        System.out.println("prénom : " + prenom.toUpperCase());
        System.out.println("âge : " + calculAge());
    }
    else
    {
        System.out.println("nom : " + nom.toLowerCase());
        System.out.println("prénom : " + prenom.toLowerCase());
        System.out.println("âge : " + calculAge());
    }
}

```

Il détermine en fait que deux fonctions ont rigoureusement la même signature, même nom, même nombre de paramètres, même type de paramètre. Cet exemple nous montre également que le nom des paramètres n'est pas pris en compte pour déterminer la signature d'une fonction.

Une fonction peut être conçue pour accepter un nombre variable de paramètres. La première solution consiste à utiliser comme paramètre un tableau et à vérifier dans le code de la fonction la taille du tableau pour obtenir les paramètres. Cette solution nécessite cependant la création d'un tableau au moment de l'appel de la fonction. Elle est donc moins évidente que l'utilisation d'une liste de paramètres. Pour simplifier l'appel de ce type de fonction, vous pouvez utiliser la déclaration suivante pour indiquer qu'une fonction attend un nombre quelconque de paramètres.

```
public void affichage(String...couleurs)
{
    if (couleurs==null)
    {
        System.out.println("pas de couleur");
        return;
    }
    switch (couleurs.length)
    {
        case 1:
            System.out.println("une couleur");
            break;
        case 2:
            System.out.println("deux couleurs");
            break;
        case 3:
            System.out.println("trois couleurs");
            break;
        default :
            System.out.println("plus de trois couleurs");
    }
}
```

À l'intérieur de la méthode, le paramètre `couleurs` est considéré comme un tableau (de chaînes de caractères dans notre cas).

Par contre, lors de l'appel de la fonction, vous utilisez une liste de chaînes de caractères séparées par des virgules. Les syntaxes suivantes sont tout à fait valides pour l'appel de cette fonction.

```
p.affichage("rouge");
p.affichage("vert","bleu","rouge");
p.affichage();
```

Il y a juste une petite anomalie au moment de l'exécution car le dernier appel de la fonction `affichage` n'exécute pas la fonction que nous venons de concevoir mais la première version qui n'attend aucun paramètre. En effet le compilateur ne peut pas deviner qu'il s'agit de la version qui attend un nombre variable de paramètres que nous souhaitons exécuter sans lui passer de paramètre. Pour lui montrer notre intention nous devons donc appeler la fonction avec la syntaxe suivante.

```
p.affichage(null);
```

Lors de l'appel d'une fonction les paramètres sont passés par valeur aussi bien pour les types de base du langage (`int`, `float`, `boolean`...) que pour les types objets. Cependant si un objet est passé comme paramètre à une fonction, le code de la fonction a accès aux champs de l'objet et peut donc modifier leurs valeurs. Par contre si le code de la fonction modifie la référence vers l'objet, celle-ci sera rétablie après le retour de la fonction.

Il faut noter que dans les méthodes de la classe nous pouvons utiliser les champs de la classe même s'ils sont déclarés `private`, car nous sommes à l'intérieur de la classe.

d. Les accesseurs

La déclaration des attributs avec une visibilité privée est une bonne pratique pour respecter le principe d'encapsulation. Toutefois cette solution est limitative puisque seul le code de la classe où ils sont déclarés peut y accéder. Pour pallier ce problème vous devez mettre en place des accesseurs. Ce sont des fonctions ordinaires qui ont simplement pour but de rendre visibles les champs à partir de l'extérieur de la classe. Par convention, les fonctions chargées d'affecter une valeur à un champ sont nommées `set` suivi du nom du champ, les fonctions chargées de fournir la valeur d'un champ sont nommées `get` suivi du nom du champ. Si le champ est de type `boolean`, le préfixe `get` est remplacé par le préfixe `is`. Si un champ doit être en lecture seule, l'accesseur `set` ne doit pas être disponible, si un champ doit être en écriture seule alors c'est la fonction `get` qui doit être omise.

Avec cette technique vous pouvez contrôler l'utilisation qui sera faite des champs d'une classe. Nous pouvons donc modifier la classe `Personne` en y ajoutant quelques règles de gestion :

- Le nom doit être en majuscules.
- Le prénom doit être en minuscules.

```
public class Personne
{
    private String nom;
    private String prenom;
    private LocalDate date_nais;

    public String getNom()
    {
        return nom;
    }

    public void setNom(String n)
    {
        nom = n.toUpperCase();
    }

    public String getPrenom()
    {
        return prenom;
    }

    public void setPrenom(String p)
    {
        prenom = p.toLowerCase();
    }
}
```

Les champs de la classe sont maintenant accessibles depuis l'extérieur grâce à ces méthodes.

```
p.setNom("dupont");
p.setPrenom("albert");
System.out.println(p.getNom());
System.out.println(p.getPrenom());
```

e. Constructeurs et destructeurs

Les constructeurs sont des méthodes particulières d'une classe par différents aspects. Le constructeur est une méthode portant toujours le même nom que la classe elle-même. Il ne retourne aucun type, même `void`. Il n'est jamais appelé explicitement dans le code mais de manière implicite, à la création d'une instance de classe. Comme pour une méthode classique, un constructeur peut attendre des paramètres. Le constructeur d'une classe qui n'attend pas de paramètres est désigné comme le constructeur par défaut de la classe. Le rôle du constructeur est principalement l'initialisation des champs d'une instance de classe. Comme les autres méthodes d'une classe, les constructeurs peuvent également être surchargés. La création d'un constructeur par défaut n'est pas obligatoire car le compilateur en fournit un automatiquement. Ce constructeur effectue simplement un appel au constructeur de la super classe qui doit bien sûr exister. Par contre si votre classe contient un constructeur surchargé, elle doit également

posséder un constructeur par défaut. Une bonne habitude à prendre consiste à toujours créer un constructeur par défaut pour chacune de vos classes. Ajoutons à la classe Personne des constructeurs.

```
public Personne()
{
    nom="";
    prenom="";
    date_nais=null;
}
public Personne(String n, String p, LocalDate d)
{
    nom=n;
    prenom=p;
    date_nais=d;
}
```

Les destructeurs sont d'autres méthodes particulières d'une classe. Comme les constructeurs ils sont appelés implicitement mais uniquement lors de la destruction d'une instance de classe. La signature du destructeur est imposée. Cette méthode doit être `protected`, elle ne retourne aucune valeur, se nomme obligatoirement `finalize`, ne prend aucun paramètre et elle est susceptible de déclencher une exception de type `Throwable`.

Du fait de cette signature imposée, il ne peut y avoir qu'un seul destructeur pour une classe, donc pas de surcharge possible pour les destructeurs. La déclaration d'un destructeur est donc la suivante :

```
protected void finalize() throws Throwable
{
}
```

Le code présent dans le destructeur doit permettre la libération des ressources utilisées par la classe. On peut par exemple y trouver du code fermant un fichier ouvert par la classe ou la fermeture d'une connexion vers un serveur de base de données. Nous verrons en détail dans la section Destruction d'une instance les circonstances dans lesquelles est appelé le destructeur.

f. Champs et méthodes statiques

Les membres statiques sont des champs ou méthodes qui sont accessibles par la classe elle-même ou par toutes les instances de la classe. On parle également dans certains langages de membres partagés. Ils sont très utiles lorsque vous avez à gérer dans une classe des informations qui ne sont pas spécifiques à une instance de la classe mais à la classe elle-même. Par opposition aux membres d'instance pour lesquels il existe un exemplaire par instance de la classe, les membres statiques existent en un exemplaire unique. La modification de la valeur d'un membre d'instance ne modifie la valeur que pour cette instance de classe alors que la modification de la valeur d'un membre statique modifie la valeur pour toutes les instances de la classe. Les membres statiques sont assimilables à des variables globales dans une application. Ils sont utilisables dans le code en y faisant référence par le nom de la classe ou grâce à une instance de la classe. Cette deuxième solution n'est pas conseillée car elle ne met pas évidence le fait que l'on travaille avec un membre statique. Le compilateur déclenche un avertissement s'il détecte cette situation.

Les méthodes statiques suivent les mêmes règles et peuvent servir à la création de bibliothèques de fonctions. L'exemple classique est la classe `Math` dans laquelle de nombreuses fonctions statiques sont définies. Les méthodes statiques possèdent cependant une limitation car elles ne peuvent utiliser que des variables locales ou d'autres membres statiques de la classe. Elles ne peuvent pas utiliser des membres d'instance d'une classe car il se peut que la méthode soit utilisée sans qu'il existe une instance de la classe.

Le compilateur détectera cette situation en indiquant : `non-static variable cannot be referenced from a static context`.

Les membres statiques doivent être déclarés avec le mot clé `static`. Vous pouvez également, comme pour n'importe quel autre membre d'une classe, spécifier une visibilité. Par contre une variable locale à une fonction ne peut pas être statique.

Pour illustrer l'utilisation des membres statiques nous allons ajouter à la classe Personne un champ numéro. La valeur de ce champ sera renseignée automatiquement à la création de chaque instance de la classe et sera unique pour chaque instance. Les constructeurs de notre classe sont tout à fait adaptés pour réaliser ce travail. Par contre, il nous faut mémoriser combien d'instances ont été créées pour pouvoir affecter un numéro unique à chaque instance. Une variable statique privée va se charger de cette opération. Voici ci-dessous le code correspondant.

```
public class Personne
{
    private String nom;
    private String prenom;
    private LocalDate date_nais;
    // champ privé représentant le numéro de la Personne
    private int numero;
    // champ statique privé représentant le compteur de Personnes
    private static int nbInstances;

    public String getNom()
    {
        return nom;
    }

    public void setNom(String n)
    {
        nom = n.toUpperCase();
    }

    public String getPrenom()
    {
        return prenom;
    }

    public void setPrenom(String p)
    {
        prenom = p.toLowerCase();
    }
    // méthode d'instance permettant d'obtenir le numéro d'une Personne
    public int getNumero()
    {
        return numero;
    }
    // méthode statique permettant d'obtenir le nombre d'instances créées
    public static int getNbInstances()
    {
        return nbInstances;
    }
    public Personne()
    {
        nom="";
        prenom="";
        date_nais=null;
        // création d'une nouvelle Personne donc
        // incrémentation du compteur
        nbInstances++;
        // affectation à la nouvelle Personne de son numéro
        numero=nbInstances;
    }
}
```

g. Les annotations

Les annotations sont utilisées pour ajouter à un élément des informations supplémentaires. Ces informations n'ont aucun effet sur le code mais peuvent être utilisées par le compilateur, la machine virtuelle qui se chargera de l'exécution de l'application ou certains outils de développement. Elles peuvent s'appliquer à une classe, un champ, une méthode. Elle doit être spécifiée avant l'élément auquel elle se rapporte. Une annotation est précédée du symbole @ et suivie du nom de l'annotation. Le compilateur reconnaît trois types d'annotations qui vont permettre la modification de son comportement au moment

de la compilation du code.

@Deprecated est utilisé pour indiquer qu'une méthode ne doit plus être utilisée. C'est par exemple le cas lorsque vous avez décidé de faire évoluer une méthode et que vous souhaitez que la version précédente ne soit plus utilisée. Cette annotation ne change pas le résultat de la compilation mais ajoute des informations supplémentaires au code compilé. Si une autre classe essaie d'utiliser cette méthode un avertissement est déclenché lors de la compilation du code de cette classe. Si nous ajoutons cette annotation à la méthode `affichage` de la classe `Personne` le code qui utilise la classe `Personne` ne doit plus appeler cette méthode sous peine d'avoir un avertissement au moment de la compilation.

```
public class Personne
{
private String nom;
private String prenom;
private LocalDate date_nais;
private int numéro;
private static int nbInstances;

...
@Deprecated
public void affichage()
{
    System.out.println("nom : " + nom);
    System.out.println("prénom : " + prenom);
    System.out.println("âge : " + calculAge());
}
...
}
```

La compilation d'une classe contenant un appel à la méthode `affiche` de la classe `Personne` déclenche un avertissement sur la ligne contenant l'appel à cette méthode.

```
javac -Xlint:deprecation Principale.java
Principale.java:16: warning: [deprecation] affichage() in
Personne has been deprecated
        p.affichage();
               ^
1 warning
```

► Pour avoir un message détaillé sur les avertissements il faut utiliser l'option `-Xlint:deprecation` lors de l'appel du compilateur.

@Override est utilisé pour indiquer qu'une méthode se substitue à une méthode héritée. Cette annotation n'est pas obligatoire mais elle oblige le compilateur à vérifier que la substitution est correctement réalisée (signature identique des méthodes dans la classe de base et dans la classe actuelle). Si ce n'est pas le cas, une erreur de compilation est déclenchée. L'exemple suivant substitue la méthode `calculAge` dans une classe héritant de la classe `Personne` (la mise en œuvre de l'héritage est vue en détail un peu plus loin dans ce chapitre).

```
public class Client extends Personne
{
    @Override
    public long calculAge()
    {
        ...
        ...
    }
}
```

Cette classe est compilée sans problème, car la méthode `calculAge` possède bien la même signature que

dans la classe Personne. Par contre, si nous essayons de compiler le code suivant :

```
public class Client extends Personne
{
    @Override
    public long calculAge(int unite)
    {
        ...
        ...
    }
}
```

Nous obtenons la réponse suivante du compilateur.

```
Client.java:6: method does not override or implement a method
from a supertype
@Override
^
1 error
```

Il a effectivement raison (il faut bien avouer que c'est pratiquement toujours le cas !) nous lui avons annoncé notre intention de substituer la méthode calculAge et, en fait, nous avons effectué une surcharge, car il n'existe pas dans la classe Personne de méthode avec cette signature. Si nous enlevons le mot clé @Override, le code se compile mais il s'agit dans ce cas d'une surcharge.

`@SuppressWarnings("....")` indique au compilateur de ne pas générer certaines catégories d'avertissemens. Si par exemple vous voulez discrètement utiliser dans une fonction une méthode marquée comme `@Deprecated` vous devez utiliser l'annotation suivante devant la fonction où se trouve l'appel de cette méthode.

```
@SuppressWarnings("deprecation")
public static void main(String[] args)
{
    Personne p;
    p=new Personne();
    p.affichage();
}
```

Ce code se compile correctement sans aucun avertissement, jusqu'au jour où la méthode concernée aura disparu complètement de la classe correspondante. Car il faut avoir à l'esprit que le but de l'annotation `@Deprecated` est de déconseiller l'utilisation d'une méthode, avec certainement l'intention de la faire disparaître dans une version future de la classe.

2. Utilisation d'une classe

L'utilisation d'une classe dans une application passe par trois étapes :

- la déclaration d'une variable permettant l'accès à l'objet ;
- la création de l'objet ;
- l'initialisation d'une instance.

a. Crédation d'une instance

Les variables objet sont des variables de type référence. Elles diffèrent des variables classiques par le fait que la variable ne contient pas directement les données mais une référence sur l'emplacement dans la mémoire où se trouvent les informations. Comme pour toutes les variables elles doivent être déclarées avant leur utilisation. La déclaration se fait de manière identique à celle d'une variable classique (`int` ou autre).

```
Personne p;
```

Après cette étape la variable existe mais ne référence pas d'emplacement valide. Elle contient la valeur `null`. La deuxième étape consiste réellement à créer l'instance de la classe. Le mot clé `new` est utilisé à cet effet. Il attend comme paramètre le nom de la classe dont il est chargé de créer une instance. L'opérateur `new` fait une demande pour obtenir la mémoire nécessaire au stockage de l'instance de la classe, puis initialise la variable avec cette adresse mémoire. Le constructeur de la classe est ensuite appelé pour initialiser la nouvelle instance créée.

```
p=new Personne();
```

Les deux opérations peuvent être combinées en une seule ligne.

```
Personne p=new Personne();
```

Le constructeur par défaut est appelé dans ce cas. Pour utiliser un autre constructeur vous devez spécifier une liste de paramètres et en fonction du nombre et du type des paramètres l'opérateur `new` appelle le constructeur correspondant.

```
Personne pe=new Personne("Dupont","albert",
LocalDate.of(1956,12,13));
```

b. Initialisation d'une instance

Plusieurs possibilités sont disponibles pour initialiser les champs d'une instance de classe. La première consiste à initialiser les variables constituant les champs de la classe au moment de leur déclaration.

```
public class Personne
{
    private String nom="nouveauNom";
    private String prenom="nouveauPrenom";
    private LocalDate date_nais=LocalDate.of(1963,11,29);
    ...
}
```

Cette solution, bien que très simple, est relativement limitée puisqu'il n'est pas possible d'utiliser des structures de contrôle telles qu'une boucle à l'extérieur d'un bloc de code. La solution qui nous vient immédiatement à l'esprit est de placer le code d'initialisation à l'intérieur d'un constructeur. C'est effectivement une très bonne idée et c'est même l'objectif principal du constructeur que d'initialiser les variables d'instance. Par contre, un problème se pose avec les champs statiques, car pour eux il n'est bien sûr pas nécessaire d'avoir une instance de classe pour pouvoir les utiliser. Donc si nous plaçons le code chargé de les initialiser dans un constructeur, rien ne nous garantit que ce constructeur aura été appelé au moins une fois avant l'utilisation des champs statiques.

Pour pallier ce problème, Java propose les blocs d'initialisation statiques. Ce sont simplement des blocs de code précédés du mot clé `static` et délimités par les caractères `{` et `}`. Ils peuvent apparaître n'importe où dans le code de la classe et ils sont exécutés au chargement de la classe par la machine virtuelle dans l'ordre dans lequel ils apparaissent dans le code. Nous pouvons par exemple utiliser le code suivant pour initialiser un champ statique avec une valeur aléatoire supérieure ou égale à 1000.

```
public class Personne
{
    private String nom="nouveauNom";
    private String prenom="nouveauPrenom";
    private LocalDate date_nais=LocalDate.of(1963,11,29);
    private int numéro=0;
    private static int numInstance;

    static
    {
```

```

        while(numInstance<1000)
    {
        numInstance=(int)(10000*Math.random());
    }
}

...
...
}

```

Le même résultat peut être obtenu en créant une fonction privée statique et en l'appelant pour initialiser la variable.

```

public class Personne
{
    private String nom="nouveauNom";
    private String prenom="nouveauPrenom";
    private LocalDate date_nais=LocalDate.of(1963,11,29);
    private int numéro=0;
    private static int numInstance=initCompteur();

    private static int initCompteur()
    {
        int cpt=0;
        while(cpt<1000)
        {
            cpt=(int)(10000*Math.random());
        }
        return cpt;
    }
...
...
}

```

Cette solution présente l'avantage de pouvoir utiliser la fonction dans une autre partie du code de la classe.

Le même principe peut être appliqué pour l'initialisation des champs d'instance. Le bloc de code chargé de l'initialisation ne doit pas, dans ce cas, être précédé du mot clé `static`. Ce bloc de code est implicitement recopié au début de chaque constructeur de la classe pendant la compilation. L'initialisation d'un champ d'instance par un appel de fonction est également possible. Une petite restriction doit cependant être appliquée sur la fonction utilisée car elle ne doit pas pouvoir être substituée dans une sous-classe. Il faut pour cela la déclarer avec le mot clé `final`.

c. Destruction d'une instance

La gestion de la mémoire est parfois un vrai casse-tête dans certains langages de programmation. Le développeur est responsable de la création des instances de classes mais aussi de leur destruction afin de libérer la mémoire. Heureusement Java prend totalement à sa charge cette gestion et nous évite cette tâche fastidieuse.

Il détermine de lui-même qu'un objet n'est plus utilisé dans l'application et le supprime alors de la mémoire. Ce mécanisme est appelé Garbage Collector (ramasse-miettes). Il considère qu'un objet peut être supprimé lorsque l'application n'a plus aucun moyen d'y accéder. Cette situation se produit, par exemple à la sortie d'une fonction, lorsqu'une variable locale est utilisée pour référencer l'objet. Elle peut également être provoquée par l'affectation de la valeur `null` à une variable. Pour qu'un objet soit réellement effacé de la mémoire, il faut que tous les moyens d'y accéder depuis l'application aient disparu. Il ne faut pas oublier que si un objet est stocké dans une collection ou un tableau, la collection ou tableau, conserve une référence vers l'objet.

Regardons un peu plus dans le détail le fonctionnement du Garbage Collector. Il existe plusieurs algorithmes pour mettre en œuvre le mécanisme de gestion de la mémoire. Ces mécanismes sont implémentés par les concepteurs de la machine virtuelle Java. Nous allons rapidement regarder à titre de curiosité quelques mécanismes possibles.

Mark and Sweep

Avec ce mécanisme le Garbage Collector travaille en deux étapes. Il commence par une exploration de la mémoire depuis la racine de l'application, la méthode `main`, et parcourt ainsi tous les objets accessibles depuis cette racine. Chaque objet accessible est marqué pendant cette exploration (Mark). Il effectue ensuite un deuxième passage au cours duquel il supprime tous les objets qui ne sont pas marqués, donc inaccessibles, et retire les marques des objets restant qu'il a placées lors du premier passage. Cette solution rudimentaire présente quelques inconvénients :

- Pendant la première étape l'exécution de l'application est interrompue.
- Sa durée est proportionnelle à la quantité de mémoire utilisée par l'application.
- Après plusieurs passages la mémoire risque d'être fragmentée.

Stop and Copy

Cette solution découpe l'espace mémoire disponible pour l'application en deux parties identiques. Dès que le Garbage Collector entre en action, il effectue, comme pour la solution précédente, une exploration de la mémoire depuis la racine de l'application. Dès qu'il trouve un objet accessible, il le recopie vers la deuxième zone mémoire et modifie bien sûr les variables pour qu'elles réfèrent ce nouvel emplacement. À la fin de l'exploration tous les objets accessibles ont été recopier dans la deuxième zone mémoire. Il est alors possible d'effacer complètement le contenu de la première zone. Le même mécanisme peut alors être répété ultérieurement avec la zone mémoire qui vient d'être libérée. Cette solution présente l'avantage d'éliminer la fragmentation de la mémoire puisque les objets sont recopier les uns à la suite des autres. Par contre, un inconvénient important réside dans le déplacement fréquent des objets ayant une longue durée de vie dans l'application. Une solution intermédiaire consiste à répartir les objets en mémoire en fonction de leur espérance de vie ou de leur âge. La moitié de la mémoire disponible est donc parfois découpée en trois zones :

- Une zone pour les objets ayant une très longue durée de vie et qui n'ont pratiquement aucun risque de disparaître pendant le fonctionnement de l'application.
- Une zone pour les objets créés récemment.
- Une zone pour les objets les plus anciens.

Lorsque le Garbage Collector intervient, il traite tout d'abord la zone réservée aux objets récents. Si après ce premier passage l'application dispose de suffisamment de mémoire, le Garbage Collector arrête son traitement ; à noter que pendant ce premier passage il peut transférer des objets dans la zone réservée aux objets anciens, s'ils existent depuis suffisamment longtemps. Si la mémoire disponible n'est pas suffisante, il refait le traitement sur la zone réservée aux objets les plus anciens.

L'efficacité de cette solution repose sur un constat cruel : les objets Java n'ont pas une grande espérance de vie. Il y a donc plus de chance d'en trouver à éliminer parmi ceux qui ont été créés le plus récemment.

Parmi ces deux solutions il est difficile de dire laquelle est utilisée par une machine virtuelle Java puisque l'implémentation du Garbage Collector est laissée libre au concepteur de la machine virtuelle.

Avant d'éliminer une instance de la mémoire, le Garbage Collector appelle le destructeur de cette instance.

Le dernier point à éclaircir concernant le Garbage Collector concerne son déclenchement. C'est en fait la machine virtuelle Java qui surveille les ressources mémoire disponibles et provoque l'entrée en action du Garbage Collector lorsque celles-ci ont atteint un seuil limite (environ 85 %). Il est cependant possible de forcer l'activation du Garbage Collector en appelant la méthode `gc()` de la classe `System`. Cette utilisation doit être exceptionnelle, car une utilisation trop fréquente affecte les performances de l'application. Elle peut être utilisée juste avant que l'application utilise une quantité de mémoire importante, comme par exemple, la création d'un tableau volumineux.

Le code ci-dessous permet de mettre en évidence l'action du Garbage Collector.

```
import java.util.GregorianCalendar;
public class Personne
{
    private String nom="nouveauNom";
    private String prenom="nouveauPrenom";
```

```
private LocalDate date_nais=LocalDate.of(1963,11,29);
private int numero=0;
private static int numInstance;
public String getNom()
{
    return nom;
}

public void setNom(String n)
{
    nom = n.toUpperCase();
}

public String getPrenom()
{
    return prenom;
}

public void setPrenom(String p)
{
    prenom = p.toLowerCase();
}

@Override
protected void finalize() throws Throwable
{
    System.out.print("\u2020");
    super.finalize();
}
public int getNumero()
{
    return numero;
}

public static int getNbInstances()
{
    return numInstance;
}

public Personne()
{
    nom="";
    prenom="";
    date_nais=null;
    numInstance++;
    numero=numInstance;
}
public Personne(String n,String p,LocalDate d)
{
    nom=n;
    prenom=p;
    date_nais=d;
    numInstance++;
    numero=numInstance;
}
}

/*********************************/
import java.time.LocalDate;

public class GestionMemoire
{

    public static void main(String[] args) throws
InterruptedException
    {
        double total;
        double reste;
        double pourcentage;
```

```

        for (int j=0;j<1000;j++)
    {
        creationTableau();
        total=Runtime.getRuntime().totalMemory();
        reste=Runtime.getRuntime().freeMemory();
        pourcentage=100-(reste/total)*100;
        System.out.println("création du " + j + "ième
tableau mémoire pleine à : " + pourcentage + "%");
        // une petite pause pour pouvoir lire les messages
        Thread.sleep(1000);
    }

}

public static void creationTableau()
{
    // création d'un tableau de 1000 Personnes dans une variable locale
    // à la fin de cette fonction les éléments du tableau ne sont
    // plus accessibles et peuvent être supprimés de la mémoire
    Personne[] tablo;
    tablo=new Personne[1000];
    for (int i=0;i<1000;i++)
    {
        tablo[i]=new
Personne("Dupont","albert",LocalDate.of(1956,12,13));
    }
}
}

```

3. Héritage

L'héritage est une puissante fonctionnalité d'un langage orienté objet mais peut parfois être utilisée mal à propos. Deux types de relations peuvent être envisagés entre deux classes. Nous pouvons avoir la relation « est une sorte de » et la relation « concerne un ». La relation d'héritage doit être utilisée lorsque la relation « est une sorte de » peut être appliquée entre deux classes. Prenons un exemple avec trois classes : Personne, Client, Commande.

Essayons la relation « est une sorte de » pour chacune des classes.

- Une commande est une sorte de client.
- Une commande est une sorte de personne.
- Un client est une sorte de commande.
- Un client est une sorte de personne.
- Une personne est une sorte de client.
- Une personne est une sorte de commande.

Parmi toutes ces tentatives, il n'y en a qu'une seule qui nous semble logique : un client est une sorte de personne. Nous pouvons donc envisager une relation d'héritage entre ces deux classes.

La mise en œuvre est très simple au niveau du code puisque dans la déclaration de la classe il suffit juste de spécifier le mot clé `extends` suivi du nom de la classe dont on souhaite hériter. Java n'acceptant pas l'héritage multiple, vous ne pouvez spécifier qu'un seul nom de classe de base. À l'intérieur de cette nouvelle classe vous pouvez :

- Utiliser les champs hérités de la classe de base (à condition bien sûr que leur visibilité le permette).
- Ajouter des nouveaux champs.
- Masquer un champ hérité en le déclarant avec le même nom que celui utilisé dans la classe de base. Cette technique doit être utilisée avec modération.
- Utiliser une méthode héritée sous réserve que sa visibilité le permette.
- Substituer une méthode héritée en la déclarant à l'identique (même signature).

- Surcharger une méthode héritée en la créant avec une signature différente.
- Ajouter une nouvelle méthode.
- Ajouter un ou plusieurs constructeurs.

Voici par exemple la création de la classe Client qui hérite de la classe Personne et à laquelle est ajouté le champ type et les accesseurs correspondants.

```
public class Client extends Personne
{
    // détermination du type de client
    // P -> particulier
    // E -> entreprise
    // A -> administration
    private char type;
    public char getType()
    {
        return type;
    }
    public void setType(char t)
    {
        type = t;
    }
}
```

La classe peut ensuite être utilisée et propose toutes les fonctionnalités définies dans la classe Client plus celles héritées de la classe Personne.

```
Client c;
c=new Client();
c.setNom("ENI");
c.setPrenom("");
c.setDate_nais(LocalDate.of(1981,05,15));
c.setType('E');
c.affichage();
```

a. this et super

Il est légitime de vouloir ensuite modifier le fonctionnement de certaines méthodes héritées pour les adapter à la classe Client. Par exemple la méthode affichage peut être substituée pour tenir compte du nouveau champ disponible dans la classe.

```
public void affichage()
{
    System.out.println("nom : " + getNom());
    System.out.println("prenom : " + getPrenom());
    System.out.println("age : " + calculAge());
    switch (type)
    {
        case 'P':
            System.out.println("type de client : Particulier");
            break;
        case 'E':
            System.out.println("type de client : Entreprise");
            break;
        case 'A':
            System.out.println("type de client : Administration");
            break;
        default:
            System.out.println("type de client : Inconnu");
            break;
    }
}
```

Ce code fonctionne très bien mais ne respecte pas un des principes de la programmation objet qui veut

que l'on réutilise au maximum ce qui existe déjà. Dans notre cas nous avons déjà une portion de code chargée de l'affichage du nom, du prénom, et de l'âge d'une personne. Pourquoi ne pas la réutiliser dans la méthode affichage de la classe Client puisque l'on en hérite ?

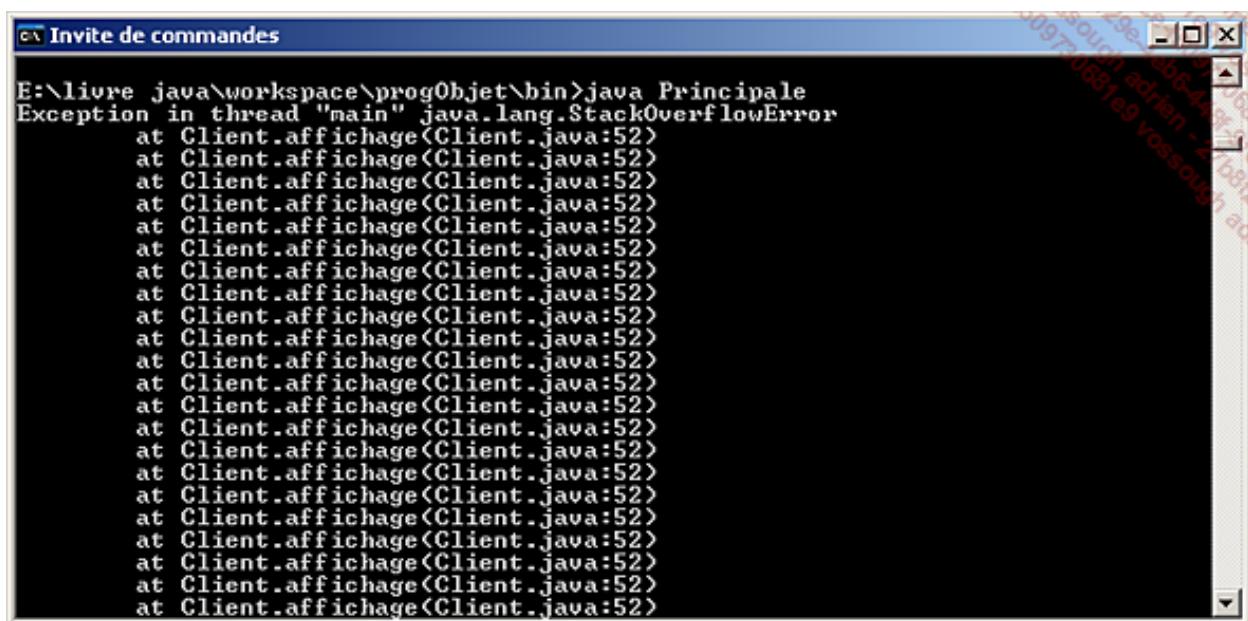
Notre méthode devient donc :

```
public void affichage()
{
    affichage();
    switch (type)
    {
        case 'P':
            System.out.println("type de client : Particulier");
            break;
        case 'E':
            System.out.println("type de client : Entreprise");
            break;
        case 'A':
            System.out.println("type de client : Administration");
            break;
        default:
            System.out.println("type de client : Inconnu");
            break;
    }
}
```

Essayons de l'utiliser :

```
Client c;
c=new Client();
c.setNom("ENI");
c.setPrenom("");
c.setDate_naissance(LocalDate.of(1981,05,15));
c.setType('E');
c.affichage();
```

Hélas le résultat n'est pas à la hauteur de nos espérances !



```
E:\livre\java\workspace\progObjet\bin>java Principale
Exception in thread "main" java.lang.StackOverflowError
    at Client.affichage(Client.java:52)
    at Client.affichage(Client.java:52)
```

Que s'est-il passé lors de l'exécution ?

Lors de l'appel de la méthode affichage la première ligne de code a consisté à appeler la méthode affichage de la classe de base. En fait la machine virtuelle Java recherche la première méthode affichage qu'elle trouve. Pour cela, elle commence la recherche dans la classe à partir de laquelle est créé l'objet, dans notre cas la classe Client. Elle appelle donc ainsi en boucle la

méthode affichage de la classe Client d'où l'erreur de débordement de pile que nous obtenons. Pour éviter ce genre de problème, nous devons lui préciser que la méthode affichage à appeler se trouve dans la classe de base. Pour cela, nous devons utiliser le mot clé super pour qualifier la méthode affichage appelée.

```
public void affichage()
{
    super.affichage();
    switch (type)
    {
        case 'P':
            System.out.println("type de client : Particulier");
            break;
        case 'E':
            System.out.println("type de client : Entreprise");
            break;
        case 'A':
            System.out.println("type de client : Administration");
            break;
        default:
            System.out.println("type de client : Inconnu");
            break;
    }
}
```

Après cette modification tout rentre dans l'ordre et notre code affiche :

Le même mot clé peut être utilisé pour appeler le constructeur de la classe de base.

L'appel au constructeur de la classe de base doit être la première ligne d'un constructeur.

Nous pouvons donc créer un constructeur pour la classe Client qui réutilise le constructeur de la classe Personne.

➤ Après la création d'un constructeur surchargé, le constructeur par défaut généré automatiquement par le compilateur n'est plus disponible. C'est donc une bonne habitude de toujours créer un constructeur par défaut. Celui-ci devra simplement faire un appel au constructeur par défaut de la classe de base. Nous devons donc ajouter le code suivant à la classe Client.

```
public Client()
{
    // appel au constructeur par défaut de la super classe
```

```

        super();
    }
public Client(String nom, String prenom, LocalDate date_nais, char
type)
{
    // appel au constructeur surchargé de la super classe
    super(nom, prenom, date_nais);
    // initialisation du type de client
    type=type;
}

```

Vérifions que le nouveau constructeur fonctionne.

```
c=new Client("ENI","",LocalDate.of(1981,05,15),'E');
c.affichage();
```

Nous affiche :

```

nom : ENI
prenom :
age : 27
type de client : Inconnu

```

Les informations ont bien été prises en compte sauf le type de client qui n'a pas été initialisé. Regardons de plus près le code du constructeur. Nous découvrons qu'un paramètre du constructeur porte le même nom qu'un champ de la classe. Lorsque nous écrivons la ligne `type=type`, le compilateur considère que nous souhaitons affecter au paramètre `type` la valeur contenue dans le paramètre `type`. Rien d'illégal, mais ce n'est absolument pas ce que nous souhaitons faire. Nous devons indiquer que l'affectation doit se faire au champ de la classe. Pour cela nous devons préfixer son nom avec le mot clé `this`.

Le constructeur devient donc :

```

public Client(String nom, String prenom, LocalDate date_nais,
char type)
{
    // appel au constructeur surchargé de la super classe
    super(nom, prenom, date_nais);
    this.type=type;
}

```

Notre code de test affiche alors les bonnes informations :

```

nom : ENI
prenom :
age : 27
type de client : Entreprise

```

b. Classes abstraites

Les classes abstraites sont des classes qui peuvent uniquement être utilisées comme classe de base dans une relation d'héritage. Il est impossible de créer une instance d'une classe abstraite. Elles servent essentiellement de modèle pour la création de classe devant toutes avoir un minimum de caractéristiques identiques. Elles peuvent contenir des champs, des propriétés et des méthodes comme une classe ordinaire. Pour qu'une classe soit une classe abstraite vous devez utiliser le mot clé `abstract` lors de sa déclaration.

Cette technique facilite l'évolution de l'application car si une nouvelle fonctionnalité doit être disponible dans les classes dérivées, il suffit d'ajouter cette fonctionnalité dans la classe de base. Il est également possible de ne pas fournir d'implémentation pour certaines méthodes d'une classe abstraite et ainsi laisser à l'utilisateur de la classe le soin de créer l'implémentation dans la classe dérivée. Ces méthodes doivent également être déclarées avec le mot clé `abstract`. Il n'y a dans ce cas pas de bloc de code correspondant à cette méthode et sa déclaration doit se terminer avec un point-virgule.

Une classe abstraite n'a pas obligatoirement de méthodes abstraites, par contre une classe contenant une méthode abstraite doit obligatoirement être elle aussi abstraite. La classe qui héritera de la classe abstraite devra implémenter les méthodes déclarées abstraites dans sa classe de base ou être elle aussi abstraite. Si elle implémente une méthode abstraite, elle ne peut pas réduire la visibilité déclarée dans la classe abstraite. Voici ci-dessous un exemple de classe abstraite :

```
public abstract class EtreVivant
{
    private double taille;
    private double poids;
    public double getTaille()
    {
        return taille;
    }
    public void setTaille(double taille)
    {
        this.taille = taille;
    }
    public double getPoids()
    {
        return poids;
    }
    public void setPoids(double poids)
    {
        this.poids = poids;
    }
    // cette méthode devra être implémentée dans les classes dérivées
    protected abstract void seDeplacer();
}
```

c. Classes finales

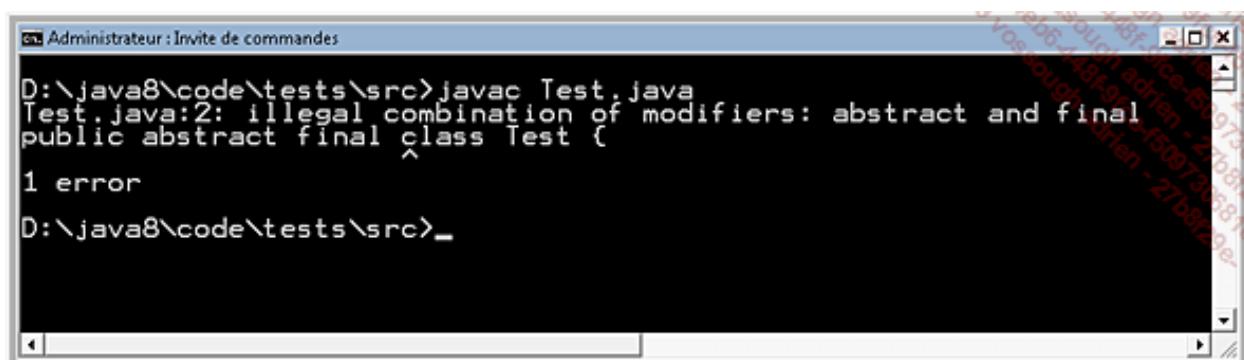
Les classes sont des classes ordinaires qui peuvent être instanciées mais ne sont pas utilisables comme classe de base dans une relation d'héritage. Elles doivent être déclarées avec le mot clé `final`. C'est le cas de plusieurs classes du langage Java, comme par exemple la classe `String` dont voici la déclaration extraite de la documentation Java.

```
public final class String
extends Object
implements Serializable, Comparable, CharSequence

The String class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class.
```

Une méthode peut également être déclarée avec le mot clé `final` pour interdire sa redéfinition dans une sous-classe.

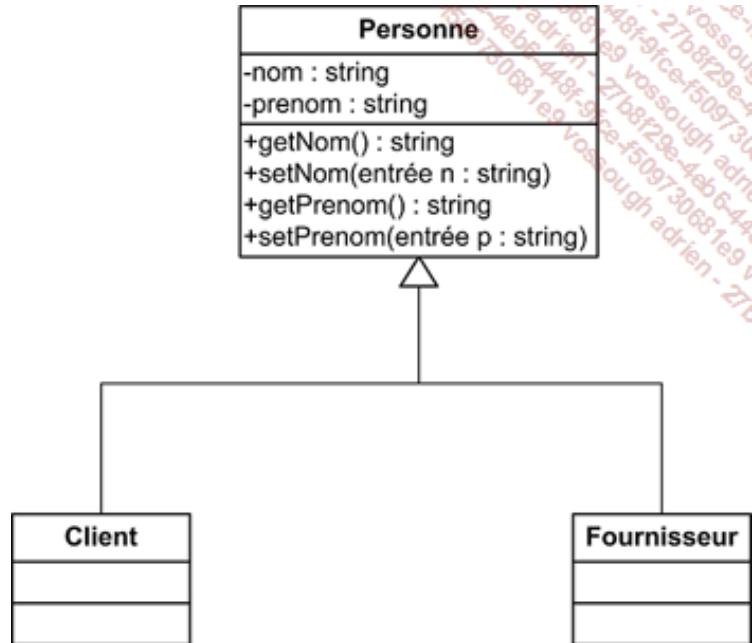
La signification des mots clés `abstract` et `final` étant contradictoire, l'utilisation simultanée de ces deux mots clés est bien sûr interdite. Le compilateur détecte cette situation et génère une erreur.



d. Conversion de type

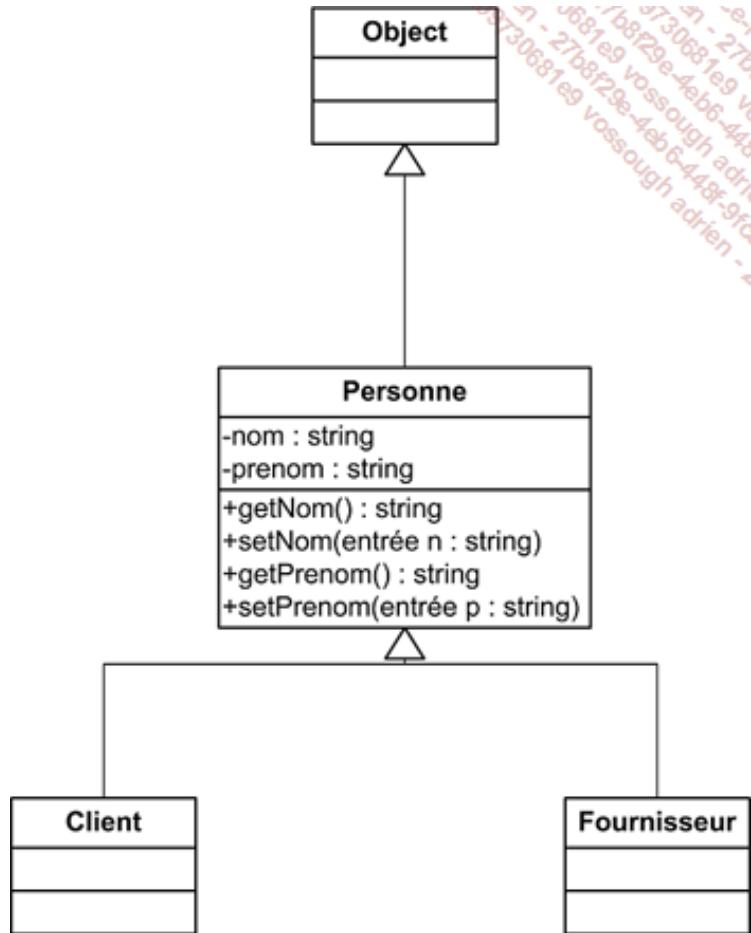
Pour bien comprendre les opérations de conversion de type il faut toujours garder présent à l'esprit que une fois créé en mémoire un objet ne changera jamais de type jusqu'à la fin de sa vie. Ainsi si vous créez une instance de la classe Personne, cette instance restera toujours une instance de la classe Personne. Avec ce préambule vous devez vous poser des questions sur la signification réelle du terme "conversion". En fait la conversion n'intervient pas sur l'objet lui-même mais sur la façon de le manipuler. Elle va agir sur le type de la variable utilisée pour accéder à l'objet.

La relation d'héritage entre classes est l'élément fondamental permettant l'utilisation des conversions de type. Nous avons vu qu'une classe pouvait récupérer les caractéristiques d'une autre classe par cet intermédiaire. C'est-à-dire qu'elle récupère automatiquement les caractéristiques de sa classe de base. Nous pouvons par exemple avoir la hiérarchie de classe suivante.



La classe Client est une évolution de la classe Personne au même titre que la classe Fournisseur est une évolution de la classe Personne. Toutes les caractéristiques disponibles avec une instance de la classe Personne seront également disponibles avec une instance de la classe Client ou une instance de la classe Fournisseur.

En fait notre schéma n'est pas complet, il devrait plutôt avoir la forme suivante :



Il ne faut jamais oublier qu'une classe qui n'a pas de classe de base explicite hérite implicitement de la classe `Object`. Nous pouvons donc dire que n'importe quelle instance de classe aura au minimum les caractéristiques d'une instance de la classe `Object`.

Maintenant que nous avons établi un bilan, voyons sous forme d'un petit jeu ce que nous pouvons faire et ne pas faire avec les conversions.

En partant du code suivant :

```

Object o;
Personne p;
Client c;
Fournisseur f;

```

Vous devez indiquer parmi les lignes suivantes quelles sont celles qui posent un problème.

Pour vous aider, posez-vous toujours la question : est-ce que ce que je peux faire avec une variable de type X est disponible dans l'instance de classe que je veux lui faire référencer ?

<code>f=new Fournisseur();</code>	<input type="checkbox"/> fonctionne	<input type="checkbox"/> ne fonctionne pas
<code>o=f;</code>	<input type="checkbox"/> fonctionne	<input type="checkbox"/> ne fonctionne pas
<code>p=f;</code>	<input type="checkbox"/> fonctionne	<input type="checkbox"/> ne fonctionne pas
<code>c=f;</code>	<input type="checkbox"/> fonctionne	<input type="checkbox"/> ne fonctionne pas
<code>c=new Client();</code>	<input type="checkbox"/> fonctionne	<input type="checkbox"/> ne fonctionne pas
<code>o=c;</code>	<input type="checkbox"/> fonctionne	<input type="checkbox"/> ne fonctionne pas
<code>p=c;</code>	<input type="checkbox"/> fonctionne	<input type="checkbox"/> ne fonctionne pas
<code>f=c;</code>	<input type="checkbox"/> fonctionne	<input type="checkbox"/> ne fonctionne pas
<code>p=new Personne();</code>	<input type="checkbox"/> fonctionne	<input type="checkbox"/> ne fonctionne pas
<code>o=p;</code>	<input type="checkbox"/> fonctionne	<input type="checkbox"/> ne fonctionne pas
<code>c=p;</code>	<input type="checkbox"/> fonctionne	<input type="checkbox"/> ne fonctionne pas
<code>f=p;</code>	<input type="checkbox"/> fonctionne	<input type="checkbox"/> ne fonctionne pas

<code>o=new Object();</code>	<input type="checkbox"/>	fonctionne	<input type="checkbox"/>	ne fonctionne pas
<code>p=o;</code>	<input type="checkbox"/>	fonctionne	<input type="checkbox"/>	ne fonctionne pas
<code>f=o;</code>	<input type="checkbox"/>	fonctionne	<input type="checkbox"/>	ne fonctionne pas
<code>c=o;</code>	<input type="checkbox"/>	fonctionne	<input type="checkbox"/>	ne fonctionne pas

Voici la solution :

<code>f=new Fournisseur();</code>	<input checked="" type="checkbox"/>	fonctionne	<input type="checkbox"/>	ne fonctionne pas
<code>o=f;</code>	<input checked="" type="checkbox"/>	fonctionne	<input type="checkbox"/>	ne fonctionne pas
<code>p=f;</code>	<input checked="" type="checkbox"/>	fonctionne	<input type="checkbox"/>	ne fonctionne pas
<code>c=f;</code>	<input type="checkbox"/>	fonctionne	<input checked="" type="checkbox"/>	ne fonctionne pas
<code>c=new Client();</code>	<input checked="" type="checkbox"/>	fonctionne	<input type="checkbox"/>	ne fonctionne pas
<code>o=c;</code>	<input checked="" type="checkbox"/>	fonctionne	<input type="checkbox"/>	ne fonctionne pas
<code>p=c;</code>	<input checked="" type="checkbox"/>	fonctionne	<input type="checkbox"/>	ne fonctionne pas
<code>f=c;</code>	<input type="checkbox"/>	fonctionne	<input checked="" type="checkbox"/>	ne fonctionne pas
<code>p=new Personne();</code>	<input checked="" type="checkbox"/>	fonctionne	<input type="checkbox"/>	ne fonctionne pas
<code>o=p;</code>	<input checked="" type="checkbox"/>	fonctionne	<input type="checkbox"/>	ne fonctionne pas
<code>c=p;</code>	<input type="checkbox"/>	fonctionne	<input checked="" type="checkbox"/>	ne fonctionne pas
<code>f=p;</code>	<input type="checkbox"/>	fonctionne	<input checked="" type="checkbox"/>	ne fonctionne pas
<code>o=new Object();</code>	<input type="checkbox"/>	fonctionne	<input checked="" type="checkbox"/>	ne fonctionne pas
<code>p=o;</code>	<input type="checkbox"/>	fonctionne	<input checked="" type="checkbox"/>	ne fonctionne pas
<code>f=o;</code>	<input type="checkbox"/>	fonctionne	<input checked="" type="checkbox"/>	ne fonctionne pas
<code>c=o;</code>	<input type="checkbox"/>	fonctionne	<input checked="" type="checkbox"/>	ne fonctionne pas

Nous pouvons déduire de ce résultat la règle suivante : avec une variable du type X nous pouvons bien sûr référencer une instance de classe de type X mais aussi une instance de n'importe quelle sous-classe de la classe X. Ainsi une variable de type `Object` peut être utilisée pour référencer une instance de n'importe quelle classe.

Pour poursuivre nos expérimentations sur les conversions voici un deuxième test.

```

...
o=new Object();
p=new Personne();
c=new Client();
f=new Fournisseur();

essai(o);       fonctionne       ne fonctionne pas
essai(p);       fonctionne       ne fonctionne pas
essai(f);       fonctionne       ne fonctionne pas
essai(c);       fonctionne       ne fonctionne pas
...

public void essai(Object obj)
{
    Object o;
    Personne p;
    Client c;
    Fournisseur f;

    o=obj;           fonctionne       ne fonctionne pas
    p=obj;           fonctionne       ne fonctionne pas
    c=obj;           fonctionne       ne fonctionne pas
    f=obj;           fonctionne       ne fonctionne pas
}

```

Voici la solution de ce deuxième exercice :

```

...
o=new Object();
p=new Personne();
c=new Client();
f=new Fournisseur();

```

```

        essai(o);      ■ fonctionne     □ ne fonctionne pas
        essai(p);      ■ fonctionne     □ ne fonctionne pas
        essai(f);      ■ fonctionne     □ ne fonctionne pas
        essai(c);      ■ fonctionne     □ ne fonctionne pas
        ...
public void essai(Object obj)
{
    Object o;
    Personne p;
    Client c;
    Fournisseur f;

    o=obj;          ■ fonctionne     □ ne fonctionne pas
    p=obj;          □ fonctionne     ■ ne fonctionne pas
    c=obj;          □ fonctionne     ■ ne fonctionne pas
    f=obj;          □ fonctionne     ■ ne fonctionne pas
}

```

Les règles à suivre pour résoudre ce problème sont les mêmes que pour l'exercice précédent. Lors de l'appel de la fonction `essai` le passage du paramètre est équivalent à une affectation à une variable de type `Object`. Nous pouvons donc appeler cette fonction en lui fournissant une instance de n'importe quelle classe.

À l'intérieur de la fonction `essai` le problème est inversé. Le paramètre `obj` référence une instance de classe mais le compilateur ne peut pas savoir de quelle classe il s'agit. C'est pour cette raison qu'il n'accepte que l'affectation `o=obj`; qui est sans risque puisque les deux variables sont de même type. Il est toutefois possible de contourner ce problème en effectuant une opération de transtypage. Cette opération est réalisée simplement en faisant précéder la variable du nom de la classe vers laquelle nous voulons réaliser le transtypage. Le nom de la classe doit être placé entre les caractères (et).

```

public static void essai(Object obj)
{
    Object o;
    Personne p;
    Client c;
    Fournisseur f;

    o=obj;
    p=(Personne)obj;
    c=(Client)obj;
    f=(Fournisseur)obj;
}

```

Nous n'avons plus d'erreurs au moment de la compilation. Par contre au moment de l'exécution nous obtenons une exception de type `java.lang.ClassCastException`. Cette exception est déclenchée par la machine virtuelle lorsqu'elle découvre que nous lui avons menti en essayant de lui faire croire que la variable `obj` référence une instance d'une certaine classe alors qu'il n'en est rien. Nous devons donc être plus prudents avec nos opérations de transtypage et vérifier ce que référence réellement la variable `obj`. L'opérateur `instanceof` permet de faire cette vérification. La fonction `essai` doit donc s'écrire de la façon suivante :

```

public static void essai(Object obj)
{
    Object o;
    Personne p;
    Client c;
    Fournisseur f;

    o=obj;
    if (obj instanceof Personne)
    {
        p=(Personne)obj;
    }
    if (obj instanceof Client)
    {

```

```

        c=(Client)obj;
    }
    if (obj instanceof Fournisseur)
    {
        f=(Fournisseur)obj;
    }
}

```

Cet opérateur doit pratiquement toujours être utilisé pour vérifier la faisabilité d'une opération de transtypage.

e. La classe Object

La classe `Object` est directement ou indirectement présente dans la hiérarchie de toute classe d'une application. Les méthodes définies dans la classe `Object` sont donc disponibles pour n'importe quelle classe. L'inconvénient de ces méthodes est qu'elles ne font pas grand-chose d'utile ou leur fonctionnement n'est pas adapté aux différentes classes de l'application. Elles ont donc souvent besoin d'être substituées dans les classes de l'application pour pouvoir être utilisées efficacement. Il est donc important de bien connaître leur utilité pour adapter leur fonctionnement. Nous allons donc étudier en détail les plus utilisées d'entre elles.

`hashCode`

Cette méthode permet d'obtenir l'adresse mémoire où est stockée une instance de classe. En elle-même elle n'est pas très utile, mis à part pour quelques expérimentations comme nous le ferons dans le paragraphe suivant, par contre elle est utilisée en interne par d'autres méthodes et principalement par la méthode `equals`.

`clone`

La méthode `clone` peut revendiquer le titre de "photocopieur" d'objets. Par son intermédiaire nous pouvons obtenir une copie conforme d'un objet présent en mémoire. Pour que cette méthode soit disponible, il est indispensable que la classe à partir de laquelle l'objet à copier a été créé, implémente l'interface `Cloneable`. Cette interface exige la création d'une méthode `clone` dans la classe. Cette méthode est souvent très simple puisqu'il suffit qu'elle appelle la méthode `clone` de la classe `Object`. Cette version par défaut se contente d'effectuer une copie de la zone mémoire correspondant à l'instance de classe à dupliquer. Si l'instance à dupliquer contient des références vers d'autres objets, ces objets ne seront pas dupliqués et seront partagés par l'original et la copie. Si ce fonctionnement n'est pas adapté à l'application, il faut concevoir la méthode `clone` pour qu'elle effectue également une copie des objets référencés.

Pour illustrer ce mécanisme, nous allons travailler avec la classe `Client` à laquelle nous allons associer une classe `Commande` elle-même associée à une classe `LignesDeCommandes`.



Si nous effectuons une copie d'une commande nous conservons la référence vers le client par contre les lignes de commande doivent être dupliquées. Nous devons donc concevoir la méthode `clone` de la classe `Commande` pour qu'elle duplique également l'instance de la classe `LignesDeCommande` référencée. Voici ci-dessous un extrait du code de ces classes.

```

public class Commande implements Cloneable
{
    Client leClient;
    LignesDeCommande lesLignes;

    public Commande()
    {
        super();
        lesLignes=new LignesDeCommande();
    }
}

```

```

}
public Object clone() throws CloneNotSupportedException
{
    Commande cmd;
    // création d'une copie de la commande
    cmd=(Commande)super.clone();
    // duplication des lignes de la commande
    cmd.lesLignes=(LignesDeCommande)lesLignes.clone();
    return cmd;
}

public Client getLeClient()
{
    return leClient;
}
public void setLeClient(Client leClient)
{
    this.leClient = leClient;
}
public LignesDeCommande getLesLignes()
{
    return lesLignes;
}
public void setLesLignes(LignesDeCommande lesLignes)
{
    this.lesLignes = lesLignes;
}
}

/*****public class LignesDeCommande implements Cloneable
{
    public Object clone() throws CloneNotSupportedException
    {
        return super.clone();
    }
}

```

Nous pouvons maintenant créer des instances de nos classes et vérifier le bon fonctionnement.

```

Client c;
c=new Client("ENI","",new GregorianCalendar(1981,05,15),'E');
Commande cmd1,cmd2;
// création et initialisation d'une commande
cmd1=new Commande();
cmd1.setLeClient(c);
System.out.println("hashCode de la commande : " +cmd1.hashCode());
System.out.println("hashCode du Client : "
+cmd1.getLeClient().hashCode());
System.out.println("hashCode des lignes : "
+cmd1.getLesLignes().hashCode());
cmd2=(Commande)cmd1.clone();
System.out.println("hashCode de la copie : " +cmd2.hashCode());
System.out.println("hashCode du Client de la copie: "
+cmd2.getLeClient().hashCode());
System.out.println("hashCode des lignes de la copie:"
+cmd2.getLesLignes().hashCode());

```

Nous obtenons le résultat suivant à l'exécution de ce code :

```

hashCode de la commande : 6413875
hashCode du Client : 21174459
hashCode des lignes : 827574
hashCode de la copie : 17510567
hashCode du Client de la copie: 21174459
hashCode des lignes de la copie:27744459

```

Nous avons bien deux commandes distinctes qui référencent le même client.

equals

La méthode `equals` est utilisée pour comparer deux instances de classe. Le code suivant permet de vérifier l'égalité de deux clients.

```
Client c1,c2;
c1=new Client("ENI","",LocalDate.of(1981,05,15),'E');
c2=new Client("ENI","",LocalDate.of(1981,05,15),'E');
if (c1.equals(c2))
{
    System.out.println("les deux clients sont identiques");
}
else
{
    System.out.println("les deux clients sont différents");
}
```

Malgré les apparences, l'exécution de ce code nous affiche le résultat suivant :

```
les deux clients sont différents
```

L'implémentation par défaut de cette méthode effectue une comparaison des références pour déterminer s'il y a égalité des deux objets. Ce sont en fait les `hashCode` des deux objets qui sont comparés. Si nous voulons avoir un critère de comparaison différent, nous devons substituer la méthode `equals` dans la classe `Client` en utilisant nos propres critères de comparaison.

```
public boolean equals(Object obj)
{
    Client c;
    // vérification si obj est null ou référence une instance
    // d'une autre classe
    if (obj ==null || obj.getClass()!=this.getClass())
    {
        return false;
    }
    else

    {
        c=(Client)obj;
        // vérification des critères d'égalité sur
        // - le nom
        // - le prénom
        // - la date de naissance
        // - le type de client
        if (c.getNom().equals(getNom())&
            c.getPrenom().equals(getPrenom()) &
            c.getDate_nais().equals(getDate_nais()) &
            c.getType()== getType() )
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}
```

Pour conserver une cohérence entre la méthode `equals` et la méthode `hashCode`, il est important de redéfinir cette dernière pour qu'elle calcule le `hashcode` à partir de la valeur des champs utilisés dans les critères de comparaison. La valeur renvoyée par cette fonction a peu d'importance, seule compte réellement la garantie d'obtenir toujours le même résultat pour toute instance de classe ayant les mêmes valeurs de champs.

Voici un exemple possible d'implémentation de la méthode hashCode :

```
public int hashCode()
{
    return this.getNom().hashCode()+
           this.getPrenom().hashCode() +
           this.getDate_nais().hashCode()+
           (int)this.getType();
}
```

getClass

Cette méthode fournit une instance de la classe `Class` contenant les caractéristiques de la classe à partir de laquelle l'objet a été créé. Nous pouvons par exemple obtenir le nom de la classe, les méthodes disponibles, les champs, etc. Par mesure de sécurité, cette méthode ne peut pas être substituée.

Voici une fonction affichant quelques informations sur la classe de l'objet qui lui est passé en paramètre.

```
public static void infoClasse(Object o)
{
    Class c;
    c=o.getClass();
    System.out.println("nom de la classe : " + c.getName());
    System.out.println("elle est dans le package : " +
c.getPackage().getName());
    System.out.println("elle hérite de la classe : " +
c.getSuperclass().getName());
    System.out.println("elle possède les champs : ");
    for (int i=0;i<c.getFields().length;i++)
    {
        System.out.print("\t" + c.getFields()[i].getName());
        System.out.println(" de type :" +
c.getFields()[i].getType().getName());
    }
    System.out.println("elle possède les méthodes : ");
    for (int i=0;i<c.getMethods().length;i++)
    {
        System.out.print("\t" + c.getMethods()[i].getName());
        System.out.print(" qui attend comme paramètre (");
        for (int
j=0;j<c.getMethods()[i].getParameterTypes().length;j++)
        {
            System.out.print(c.getMethods()[i].getParameterTypes()[j]+ " ");
        }
        System.out.println(")");
    }
}
```

toString

À l'inverse de la précédente, cette méthode devrait pratiquement toujours être surchargée. Elle permet d'obtenir la représentation d'un objet sous forme d'une chaîne de caractères. Par défaut l'implémentation de cette méthode dans la classe `Object` retourne le nom de la classe suivi du `hashCode` de l'instance. Une représentation plus parlante est bien sûr conseillée. Elle doit être construite à partir des valeurs contenues dans les champs de l'objet. Une version possible de la méthode `toString` pour la classe `Personne` pourrait par exemple être la suivante.

```
public String toString()
{
    String chaine;
    chaine="nom : " + getNom()+"\r\n";
    chaine=chaine + "prénom : " + getPrenom();
    return chaine;
```

```
}
```

L'appel de la méthode `toString` est parfois implicite lorsque l'on passe un objet comme paramètre à une fonction. Les deux syntaxes suivantes sont donc équivalentes.

```
Client c;  
c=new Client("ENI","",LocalDate.of(1981,05,15),'E');  
System.out.println(c);
```

ou

```
Client c;  
c=new Client("ENI","",LocalDate.of(1981,05,15),'E');  
System.out.println(c.toString());
```

4. Interfaces

Nous avons vu que l'on pouvait obliger une classe à implémenter une méthode en déclarant celle-ci avec le mot clé `abstract`. Si plusieurs classes doivent implémenter la même méthode, il est plus pratique d'utiliser les interfaces. Comme les classes, les interfaces permettent de définir un ensemble de constantes et méthodes. Généralement, une interface ne contient que des signatures de méthodes, elles sont dans ce cas similaires aux méthodes abstraites définies dans une classe abstraite. L'interface constitue un contrat que la classe signe. En déclarant que la classe implémente une interface, elle s'engage à fournir tout ce qui est défini dans l'interface. Il faut être prudent si vous utilisez les interfaces et ne jamais modifier une interface qui est déjà utilisée sinon vous courez le risque de devoir reprendre le code de toutes les classes qui implémentent cette interface.

À partir de la version 8 du langage Java, cette contrainte peut être contournée en créant des méthodes par défaut dans l'interface (cf. section Méthodes par défaut).

a. Crédation d'une interface

Pour pouvoir utiliser une interface, il faut la définir au préalable. La déclaration est semblable à la déclaration d'une classe mais on utilise le mot clé `interface` en lieu et place du mot clé `class`.

Vous pouvez éventuellement utiliser le mot clé `extends` pour introduire une relation d'héritage dans l'interface. Contrairement aux classes les interfaces autorisent l'héritage multiple. Dans ce cas, les noms des interfaces héritées sont séparés par des virgules après le mot clé `extends`. Il faut être raisonnable avec cette possibilité car les classes qui implémenteront cette interface devront fournir toutes les méthodes définies dans la hiérarchie de l'interface.

Créons donc notre première interface. Celle-ci va imposer la présence d'une fonction `compare` attendant comme paramètre un objet. Lors de la définition d'une interface, il est recommandé de fournir, sous forme de commentaires, une description du travail que devra accomplir chaque méthode et les résultats qu'elle devra fournir.

```
// cette interface devra être implémentée par les classes  
// pour lesquelles un classement des instances est envisagé  
public interface Classable  
{  
    // cette méthode pourra être appelée pour comparer l'instance  
    // courante avec celle reçue en paramètre  
    // la méthode retourne un entier dont la valeur dépend  
    // des règles suivantes  
    // 1 si l'instance courante est supérieure à celle reçue  
    // en paramètre  
    // 0 si les deux instances sont égales  
    // -1 si l'instance courante est inférieure à celle reçue  
    // en paramètre  
    // -99 si la comparaison est impossible  
  
    int compare(Object o);
```

```

    public static final int INFERIEUR=-1;
    public static final int EGAL=0;
    public static final int SUPERIEUR=1;
    public static final int ERREUR=-99;

}

```

b. Utilisation d'une interface

Mais quels critères allons-nous utiliser pour dire qu'un objet est supérieur à un autre ?

Dans la description de notre interface ce n'est pas notre souci ! Nous laissons le soin à l'utilisateur qui va définir une classe implémentant l'interface de définir quels sont les critères de comparaison. Par exemple dans notre classe Personne, nous pourrions implémenter l'interface Classable de la manière suivante en choisissant de comparer deux instances de la classe Personne sur le nom :

```

public class Personne
implements Classable
{

    public int compare(Object o)
    {
        Personne p;
        if (o instanceof Personne)
        {
            p=(Personne)o;
        }
        else
        {
            return Classable.ERREUR;
        }
        if (getNom().compareTo(p.getNom())<0)
        {
            return Classable.INFERIEUR;
        }
        if (getNom().compareTo(p.getNom())>0)
        {
            return Classable.SUPERIEUR;
        }

        return Classable.EGAL;
    }

...
...
}

```

Deux modifications sont visibles dans la classe :

- Le fait qu'elle implémente l'interface Classable.
- L'implémentation réelle de la fonction compare.

Dans cette fonction la comparaison se fera sur le nom des clients. Très bien, mais à quoi sert-il ?

Il arrive fréquemment que l'on ait besoin de trier des éléments dans une application. Deux solutions :

- Créer une fonction de tri spécifique pour chaque type d'élément que l'on veut trier.
- Créer une routine de tri générique et faire en sorte que les éléments que l'on utilise soient triables par cette routine.

Les interfaces vont nous aider à mettre en œuvre cette deuxième solution. Pour pouvoir trier des éléments, et quelle que soit la méthode utilisée pour le tri, nous aurons besoin de comparer deux éléments. Pour être certain que notre routine de tri fonctionnera sans problème, il faut s'assurer que les éléments qu'elle devra trier auront la possibilité d'être comparés les uns aux autres.

Nous ne pouvons garantir cela que si tous nos éléments implémentent l'interface Classable. Nous allons donc l'exiger dans la déclaration de notre routine de tri.

```
public static void tri(Classable[] tablo)
{
}
```

Définie ainsi, notre fonction sera capable de trier toutes sortes de tableaux pourvu que leurs éléments implémentent l'interface Classable. Elle retournera le tableau trié. Nous pouvons donc écrire le code suivant et utiliser la méthode compare sans risque.

```
public static Classable[] tri(Classable[] tablo)
{
    int i,j;
    Classable c;
    for (i=0;i< tablo.length;i++)
    {
        for( j = i + 1; j<tablo.length;j++)
        {
            if (tablo[j].compare(tablo[i])==Classable.INFERIEUR)
            {
                c = tablo[j];
                tablo[j] = tablo[i];
                tablo[i] = c;
            }
            else if (tablo[j].compare(tablo[i])==Classable.ERREUR)
            {
                return null;
            }
        }
    }
    return tablo;
}
```

Puis pour tester notre procédure, créons quelques clients et essayons de les trier, puis d'afficher leurs noms.

```
Personne[] tab;
tab=new Personne[5];
tab[0] = new Personne("toto2", "prenom2",new
GregorianCalendar(1922,2,15));
tab[1] = new Personne("toto1", "prenom1 ",new
GregorianCalendar(1911,1,15));
tab[2] = new Personne("toto5", "prenom5 ",new
GregorianCalendar(1955,05,15));
tab[3] = new Personne("toto3", "prenom3 ",new
GregorianCalendar(1933,03,15));
tab[4] = new Personne("toto4", "prenom4 ",new
GregorianCalendar(1944,04,15));
Personne[] tabTrie;
tabTrie=(Personne[])tri(tab);
for (int i=0;i<tabTrie.length;i++)
{
    System.out.println(tabTrie[i]);
}
```

Nous obtenons le résultat suivant :

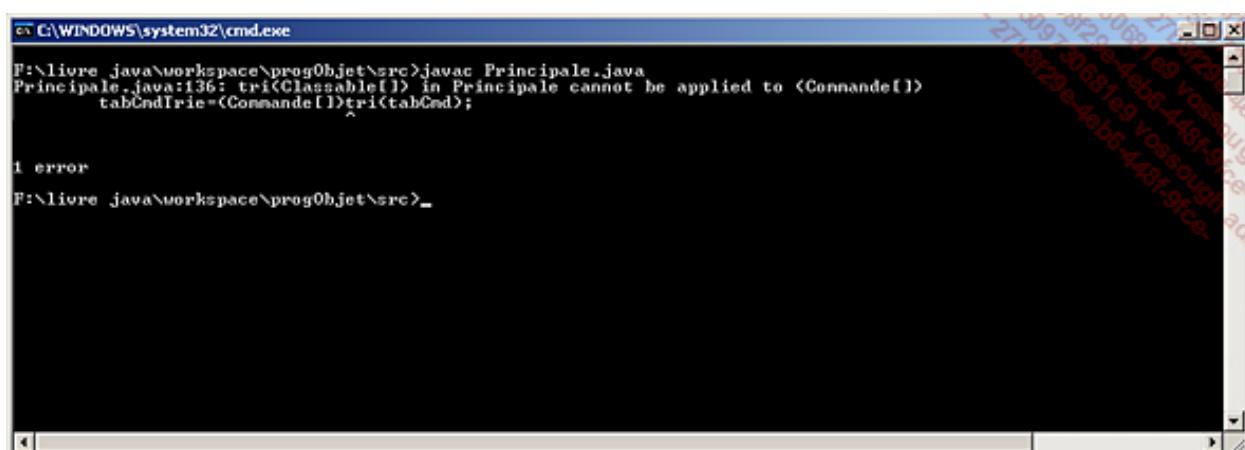
```
Mr toto1 prenom1 né le 01/01/0001 00:00:00 code Client : 1
Mr toto2 prenom2 né le 01/01/0001 00:00:00 code Client : 2
Mr toto3 prenom3 né le 01/01/0001 00:00:00 code Client : 3
Mr toto4 prenom4 né le 01/01/0001 00:00:00 code Client : 4
Mr toto5 prenom5 né le 01/01/0001 00:00:00 code Client : 5
```

Nous avons bien la liste de nos clients triée par ordre alphabétique sur le nom.

Essayons d'utiliser notre procédure de tri avec un tableau d'objets qui n'implémentent pas l'interface `Classable`.

```
Commande[] tabCmd;
tabCmd=new Commande[5];
tabCmd[0] = new Commande();
tabCmd[1] = new Commande();
tabCmd[2] = new Commande();
tabCmd[3] = new Commande();
tabCmd[4] = new Commande();
Commande[] tabCmdTrie;
tabCmdTrie=(Commande[])tri(tabCmd);
for (int i=0;i<tabCmdTrie.length;i++)
{
    System.out.println(tabCmdTrie[i]);
}
```

À la compilation les choses se compliquent.



The screenshot shows a command-line interface window titled 'cmd.exe'. The command entered is 'java \livre\workspace\progObjet\src>javac Principale.java'. The output shows an error message: 'Principale.java:136: tri<Classable[]> in Principale cannot be applied to <Commande[]> tabCmdTrie=(Commande[])tri(tabCmd);'. Below the error message, it says '1 error' and then 'F:\livre\workspace\progObjet\src>_'. The window has a standard Windows title bar and scroll bars.

Cette erreur intervient au moment de l'appel de la procédure de tri. Les éléments du tableau que nous avons passé comme paramètre n'implémentent pas l'interface `Classable` et nous ne sommes pas certains qu'ils contiennent une fonction `compare`. À noter que, même s'il existe une fonction `compareCorrecte` dans la classe `Commande`, il faut obligatoirement spécifier que cette classe implémente l'interface `Classable`, pour que le code puisse fonctionner.

c. Méthodes par défaut

Maintenant que notre code fonctionne correctement, nous pouvons envisager l'optimisation de l'algorithme de tri utilisé. Pour cette amélioration, nous avons besoin que les objets que nous souhaitons trier fournissent deux méthodes supplémentaires. Nous pouvons donc ajouter à l'interface `Classable` la signature de ces deux méthodes.

```
// cette interface devra être implémentée par les classes
// pour lesquelles un classement des instances est envisagé
public interface Classable
{
    // cette méthode pourra être appelée pour comparer l'instance
    // courante avec celle reçue en paramètre
    // la méthode retourne un entier dont la valeur dépend des règles
    // suivantes
    // 1 si l'instance courante est supérieure à celle reçue en
    // paramètre
    // 0 si les deux instances sont égales
    // -1 si l'instance courante est inférieure à celle reçue en
    // paramètre
    // -99 si la comparaison est impossible
}
```

```

int compare(Object o);
boolean isInferieur(Object o);
boolean isSuperieur(Object o);

public static final int INFERIEUR=-1;
public static final int EGAL=0;
public static final int SUPERIEUR=1;
public static final int ERREUR=-99;

}

```

L'ajout de ces deux méthodes provoque maintenant un problème lors de la compilation de la classe Personne, et plus généralement celle de toutes les classes qui ont implémenté l'ancienne version de l'interface.

```

Administrator : C:\Windows\system32\cmd.exe
C:\java8\code\tests\src>javac Personne.java
Personne.java:3: error: Personne is not abstract and does not override abstract
method isSuperieur<Object> in Classable
public class Personne implements Classable
1 error
C:\java8\code\tests\src>

```

Si de nombreuses classes ont implémenté l'ancienne version de l'interface, il va falloir fournir un important travail de modification pour que celles-ci respectent la nouvelle version de l'interface.

Pour éviter ce long et fastidieux travail, nous pouvons définir les deux méthodes ajoutées à l'interface avec le mot clé `default`. Il faut également fournir une implémentation par défaut pour ces deux méthodes. Celles-ci doivent donc comporter un bloc de code délimité par des accolades. Mais que peut-on placer dans ce bloc de code puisque nous ne savons pas sur quelle classe l'interface va être appliquée ? Peu importe, ce bloc de code est important par sa présence et non par son contenu. Il permet simplement aux classes ayant implémenté l'ancienne version de l'interface d'être compatibles avec la nouvelle version.

```

// cette interface devra être implémentée par les classes
// pour lesquelles un classement des instances est envisagé
public interface Classable
{
    // cette méthode pourra être appelée pour comparer l'instance
    // courante avec celle reçue en paramètre
    // la méthode retourne un entier dont la valeur dépend des règles
    // suivantes
    // 1 si l'instance courante est supérieure à celle reçue en
    // paramètre
    // 0 si les deux instances sont égales
    // -1 si l'instance courante est inférieure à celle reçue en
    // paramètre
    // -99 si la comparaison est impossible

    int compare(Object o);
    default boolean isInferieur(Object o)
}

```

```

    {
        return false;
    }
    default boolean isSuperieur(Object o)
    {
        return false;
    }

    public static final int INFERIEUR=-1;
    public static final int EGAL=0;
    public static final int SUPERIEUR=1;
    public static final int ERREUR=-99;
}

```

Si certaines classes n'implémentent pas toutes les méthodes de l'interface, elles héritent des méthodes définies par défaut dans l'interface. Elles ont bien sûr la possibilité de fournir leur propre implémentation de ces méthodes. Avec cette nouvelle définition de l'interface, la classe Personne n'a pas besoin d'être modifiée. Nous pouvons cependant créer d'autres classes qui implémentent complètement l'interface en fournissant toutes les méthodes de l'interface.

```

public class Voiture implements Classable
{
    private String immatriculation;
    private String marque;
    private String modele;
    private int puissance;
    public Voiture()
    {
        super();
    }

    public Voiture(String immatriculation, String marque, String
modele, int puissance)
    {
        this.immatriculation=immatriculation;
        this.marque=marque;
        this.modele=modele;
        this.puissance=puissance;
    }

    public String getImmatriculation()
    {
        return immatriculation;
    }

    public void setImmatriculation(String immatriculation)
    {
        this.immatriculation = immatriculation;
    }
    public String getMarque()
    {
        return marque;
    }

    public void setMarque(String marque)
    {
        this.marque = marque;
    }
    public String getModele()
    {
        return modele;
    }

    public void setModele(String modele)
    {
        this.modele = modele;
    }
}

```

```
public int getPuissance()
{
    return puissance;
}

public void setPuissance(int puissance)
{
    this.puissance = puissance;
}

@Override
public int compare(Object o)
{
    Voiture v;
    if (o instanceof Voiture)
    {
        v=(Voiture)o;
    }
    else
    {
        return Classable.ERREUR;
    }
    if (getPuissance()<v.getPuissance())
    {
        return Classable.INFERIEUR;
    }
    if (getPuissance()>v.getPuissance())
    {
        return Classable.SUPERIEUR;
    }

    return Classable.EGAL;
}

@Override
public boolean isInferieur(Object o)
{
    Voiture v;
    if (o instanceof Voiture)
    {
        v=(Voiture)o;
    }
    else
    {
        return false;
    }
    if (getPuissance()<v.getPuissance())
    {
        return true;
    }
    else
    {
        return false;
    }
}

@Override
public boolean isSuperieur(Object o)
{
    Voiture v;
    if (o instanceof Voiture)
    {
        v=(Voiture)o;
    }
    else
    {
        return false;
    }
    if (getPuissance()>v.getPuissance())
    {
```

```

        return true;
    }
else
{
    return false;
}
}

```

5. Classes imbriquées

La majorité des classes que vous utiliserez dans une application seront définies dans leur propre fichier source. Cependant Java fournit la possibilité de déclarer une classe à l'intérieur d'une autre classe, voire à l'intérieur d'une méthode. Cette technique permet de définir une classe uniquement dans le contexte où elle est réellement utile. On désigne également les classes imbriquées par le terme classes assistantes. Suivant l'emplacement de leur déclaration elles ont accès, soit aux autres membres de la classe dans laquelle elles sont déclarées (y compris les membres privés) soit uniquement aux variables locales des méthodes.

a. Classes imbriquées statiques

Comme n'importe quel élément déclaré dans une classe (champ ou méthode) une classe imbriquée peut être déclarée avec le mot clé `static`. Elle est dans ce cas soumise aux mêmes règles imposées par ce mot clé que les autres éléments d'une classe.

- Elle ne peut utiliser que les champs et méthodes statiques de sa classe container.
- Elle peut être utilisée (instanciée) sans qu'il existe une instance de sa classe container.
- Elle peut utiliser les membres d'instance de sa classe container uniquement par l'intermédiaire d'une instance de celle-ci.

Voici ci-dessous un exemple simple de classe imbriquée statique.

```

public class Externe
{
    static class Interne
    {
        public double calculTTC(double prix)
        {
            return prix*taux;
        }
    }
    static double taux=1.196;
}

```

Elle peut être utilisée de façon tout à fait semblable à n'importe quelle autre classe. La seule contrainte réside dans le nom utilisé pour y faire référence dans le code qui doit être précédé du nom de la classe container.

```

Externe.Interne ci;
ci=new Externe.Interne();
System.out.println(ci.calculTTC(100));

```

b. Classes internes

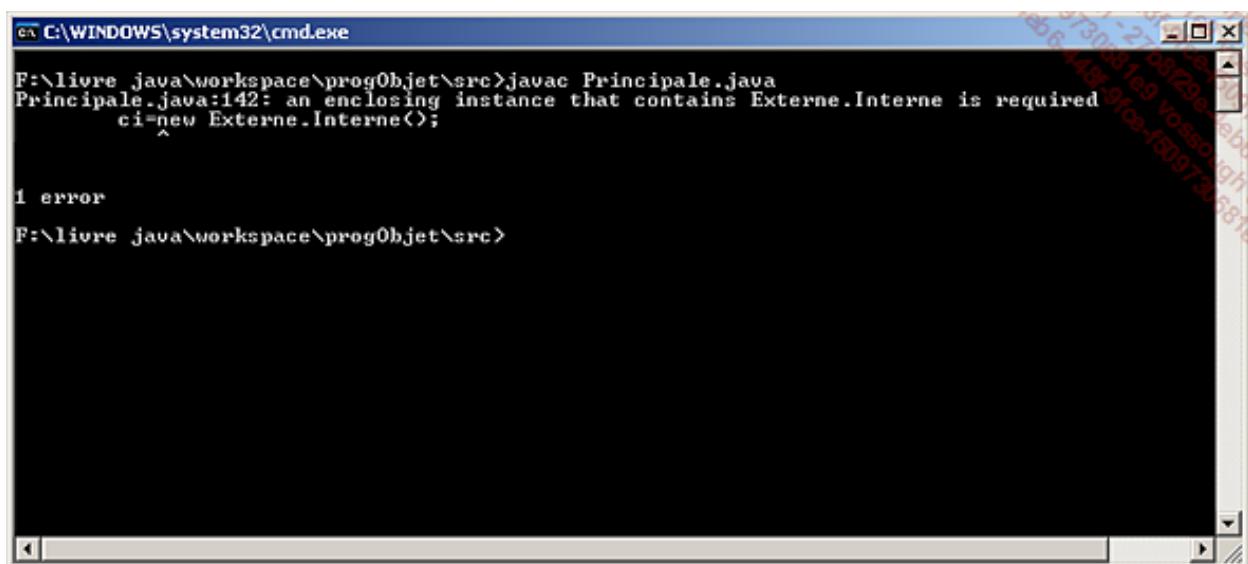
Les classes imbriquées sont également connues sous la désignation de classes internes. Elles sont soumises aux mêmes règles que n'importe quel élément déclaré dans une classe.

- Elles peuvent avoir accès à tous les membres de la classe dans laquelle elles sont déclarées y compris les membres privés.
- Elles ne peuvent être utilisées que si une instance de la classe container est disponible.

La déclaration est très simple et tout à fait similaire à la précédente hormis le mot clé static qui disparaît.

```
public class Externe
{
    class Interne
    {
        public double calculTTC(double prix)
        {
            return prix*taux;
        }
    }
    double taux=1.196;
}
```

Par contre, si nous tentons de l'utiliser avec le même code que la version static de la classe, nous rencontrons quelques soucis au moment de la compilation.



Effectivement nous devons obligatoirement avoir une instance de la classe container pour que la classe interne soit disponible. Nous devons donc au préalable instancier la classe puis lui demander de nous fournir une instance de la classe interne. La bonne syntaxe est donc :

```
Externe e;
e=new Externe();
Externe.Interne ci;
ci=e.new Interne();
System.out.println(ci.calculTTC(100));
```

c. Classes anonymes

Une classe anonyme est une classe interne pour laquelle aucun nom n'est fourni. Les classes anonymes sont tout à fait adaptées pour réaliser une opération nécessitant un objet mais qui ne justifie pas la création d'une classe normale. C'est le cas par exemple si la classe est très simple ou si elle n'est utilisée que dans une seule méthode. Ce mécanisme est très utilisé pour la gestion des actions de l'utilisateur dans une application en mode graphique.

Pour illustrer l'utilisation des classes internes anonymes, nous allons reprendre la fonction de tri de tableau créée précédemment et la rendre encore plus universelle. Dans sa version actuelle, elle exige que les éléments du tableau implémentent l'interface `Classifiable` pour qu'elle puisse les comparer deux à deux. Nous modifions légèrement la fonction pour que, lors de l'appel, on puisse lui fournir en plus du tableau à trier, la fonction qu'elle devra utiliser pour effectuer les comparaisons. Une solution pour transmettre une fonction à une autre fonction est de lui fournir une instance de classe qui contient la fonction à transmettre. Par mesure de sécurité, nous pouvons exiger que cette instance soit créée à partir d'une classe implémentant une interface pour garantir l'existence de la fonction. Voici le code de cette

nouvelle interface :

```
// cette interface devra être implémentée par les classes
// pour lesquelles une comparaison des instances est envisagée
public interface Comparateur
{
    // cette méthode pourra être appelée pour comparer les deux
    // objets reçus en paramètre
    // la méthode retourne un entier dont la valeur dépend
    // des règles suivantes
    // 1 si l'instance o1 est supérieure à o2
    // 0 si les deux instances sont égales
    // -1 si l'instance o1 est inférieure à o2
    // -99 si la comparaison est impossible
    int compare(Object o1, Object o2);

    public static final int INFERIEUR=-1;
    public static final int EGAL=0;
    public static final int SUPERIEUR=1;
    public static final int ERREUR=-99;
}
```

Une telle interface qui ne contient que la définition d'une seule et unique méthode est appelée interface fonctionnelle.

Nous pouvons maintenant revoir légèrement notre fonction de tri en prenant en compte nos améliorations.

```
public static Object[] tri(Object[] tablo, Comparateur trieur)
{
    int i, j;
    Object c;
    Object[] tabloTri;
    tabloTri = Arrays.copyOf(tablo, tablo.length);
    for (i=0; i < tabloTri.length; i++)
    {
        for (j = i + 1; j < tabloTri.length; j++)
        {
            // utilise la fonction compare de l'objet reçu en paramètre
            // pour comparer le contenu de deux cases du tableau
            if
(trieur.compare(tabloTri[j], tabloTri[i]) == Comparateur.INFERIEUR)
            {
                c = tabloTri[j];
                tabloTri[j] = tabloTri[i];
                tabloTri[i] = c;
            }
            else if
(trieur.compare(tabloTri[j], tabloTri[i]) == Comparateur.ERREUR)
            {
                return null;
            }
        }
    }
    return tabloTri;
}
```

Pour utiliser cette nouvelle fonction de tri nous devons maintenant lui fournir deux paramètres :

- le tableau à trier.
- une instance de classe ayant implémenté l'interface comparateur.

Pour fournir cette instance de classe nous avons plusieurs solutions :

- créer une instance d'une classe "normale" qui implémente l'interface.

- créer une instance classe interne qui implémente l'interface.
- créer une instance d'une classe interne anonyme qui implémente l'interface.

C'est cette dernière solution que nous allons utiliser. La syntaxe de création d'une instance d'une classe interne anonyme est au premier abord assez bizarre.

```
new "nom de la classe de base ou de l'interface implementée" ()
{
    // déclaration des champs
    int i;
    float j;
    // déclaration des méthodes
    public int methode1(...,...)
    {
        } // accolade de fermeture du bloc de code de la méthode

} // accolade de fermeture du bloc de code de la classe
```

Comme pour créer une instance de classe normale, nous utilisons l'opérateur new suivi du nom de la classe. En fait nous n'indiquons pas directement le nom de la classe dont nous souhaitons obtenir une instance, mais le nom de la classe de base ou de l'interface implémentée par cette classe (la classe n'a pas de nom puisqu'elle est anonyme !). Nous fournissons ensuite un bloc de code délimité par des accolades correspondant au contenu de la classe, comme pour la définition d'une classe normale. À noter que si la classe anonyme implémente une interface, ce bloc de code doit obligatoirement fournir toutes les méthodes exigées par l'interface.

Voici la mise en application de ces principes pour l'appel de notre fonction de tri avec comme critère de tri le nom de la personne.

```
Personne[] tab;
tab=new Personne[5];
tab[0] = new Personne("toto2", "prenom2",new
GregorianCalendar(1922,2,15));
tab[1] = new Personne("toto1", "prenom1 ",new
GregorianCalendar(1911,1,15));
tab[2] = new Personne("toto5", "prenom5 ",new
GregorianCalendar(1955,05,15));
tab[3] = new Personne("toto3", "prenom3 ",new
GregorianCalendar(1933,03,15));
tab[4] = new Personne("toto4", "prenom4 ",new
GregorianCalendar(1944,04,15));

tabTrie=(Personne[])tri(tab,
// création d'une instance de classe implementant
// l'interface Comparateur
new Comparateur()
// voici le code de la classe
{
    // comme l'exige l'interface voici la méthode compare
    public int compare(Object o1, Object o2)
    {
        Personne p1,p2;
        if (o1 instanceof Personne & o2 instanceof Personne)
        {
            p1=(Personne)o1;
            p2=(Personne)o2;
        }
        else
        {
            return Classable.ERREUR;
        }
        if (p1.getNom().compareTo(p2.getNom())<0)
        {
            return Classable.INFERIEUR;
        }
        if (p1.getNom().compareTo(p2.getNom())>0)
```

```

        {
            return Classable.SUPERIEUR;
        }

        return Classable.EGAL;
    } // accolade de fermeture de la méthode compare

} // accolade de fermeture de la classe
); // fin de l'appel de la fonction de tri

// affichage du tableau trié

for (int i=0;i<tabTrie.length;i++)
{
    System.out.println(tabTrie[i]);
}

```

Si nous voulons effectuer un autre tri avec un critère différent, nous devons simplement appeler la fonction `tri` avec une nouvelle instance d'une classe interne anonyme qui implémente différemment la fonction `Compare`. Nous pouvons par exemple trier les personnes sur leur âge.

```

tabTrie=(Personne[])tri(tab,
// création d'une instance de classe implémentant
// l'interface Comparateur
    new Comparateur()
    // voici le code de la classe
    {
        // comme l'exige l'interface voici la méthode compare
        public int compare(Object o1, Object o2)
        {
            Personne p1,p2;
            if (o1 instanceof Personne & o2 instanceof Personne)
            {
                p1=(Personne)o1;
                p2=(Personne)o2;
            }
            else
            {
                return Classable.ERREUR;
            }
            if (p1.calculAge()<p2.calculAge())
            {
                return Classable.INFERIEUR;
            }
            if (p1.calculAge()>p2.calculAge())
            {
                return Classable.SUPERIEUR;
            }
            return Classable.EGAL;
        } // accolade de fermeture de la méthode

    } // accolade de fermeture de la classe
); // fin de l'appel de la fonction de tri

for (int i=0;i<tabTrie.length;i++)
{
    System.out.println(tabTrie[i]);
}

```

Lors de la compilation de ce code, des fichiers .class sont générés pour chaque classe utilisée dans le code. Le compilateur génère automatiquement un nom pour le fichier de chaque classe anonyme. Il utilise pour cela le nom de classe container auquel il ajoute le suffixe \$ suivi d'une valeur numérique. La compilation de ce code va donc générer les fichiers suivants :

`Principale.class` : le fichier correspondant à la classe `Principale`.

Principale\$1.class : le fichier correspondant à la première classe interne anonyme.

Principale\$2.class : le fichier correspondant à la deuxième classe interne anonyme.

Le seul inconvénient des classes internes anonymes réside dans le fait qu'elles sont à usage unique. Si vous souhaitez la réutiliser plusieurs fois, il faut dans ce cas créer une classe nommée. Le nom attribué par le compilateur à la classe interne anonyme n'étant bien sûr pas accessible à partir du code (puisque elle est anonyme !).

6. Expression lambda

Dans la section précédente nous avons utilisé une classe interne anonyme pour transmettre à la fonction de tri la méthode avec laquelle nous souhaitons effectuer le classement des éléments du tableau. L'instance de la classe interne anonyme que nous avons transmise à la fonction de tri doit respecter l'interface Comparateur. Cette interface étant une interface fonctionnelle (elle ne contient qu'une seule définition de méthode), l'instance de la classe transmise ne contient elle aussi qu'une seule méthode. Cette instance de classe interne anonyme va donc pouvoir être remplacée par une expression lambda. Nous allons donc transmettre à la fonction de tri non pas une instance de classe qui contient la fonction à utiliser pour effectuer le tri, mais directement la fonction elle-même.

Une expression lambda ressemble à une fonction mis à part qu'elle ne porte pas de nom. Elle est constituée d'un couple de parenthèses contenant les éventuels paramètres suivis des caractères -> et du bloc de code de l'expression délimité par des accolades.

L'appel à la fonction de tri peut donc se faire de la manière suivante, en utilisant une expression lambda à la place de l'instance de classe implémentant l'interface Comparateur.

```
tabTrie=(Personne[])tri(tab,(Object o1, Object o2)->
{
    Personne p1,p2;
    if (o1 instanceof Personne & o2 instanceof Personne)
    {
        p1=(Personne)o1;
        p2=(Personne)o2;
    }
    else
    {
        return Classable.ERREUR;
    }
    if (p1.getNom().compareTo(p2.getNom())<0)
    {
        return Classable.INFERIEUR;
    }
    if (p1.getNom().compareTo(p2.getNom())>0)
    {
        return Classable.SUPERIEUR;
    }

    return Classable.EGAL;
});
```

Ce code ressemble énormément à celui utilisé pour appeler la fonction de tri avec comme paramètre une instance de classe anonyme.

```
tabTrie=(Personne[])tri(tab, new Comparateur()
{
    public int compare(Object o1, Object o2)
    {
        Personne p1,p2;
        if (o1 instanceof Personne & o2 instanceof Personne)
        {
            p1=(Personne)o1;
            p2=(Personne)o2;
        }
    }
});
```

```

        else
        {
            return Classable.ERREUR;
        }
        if (p1.getNom().compareTo(p2.getNom())<0)
        {
            return Classable.INFERIEUR;
        }
        if (p1.getNom().compareTo(p2.getNom())>0)
        {
            return Classable.SUPERIEUR;
        }

        return Classable.EGAL;
    }
);

```

La syntaxe reste cependant relativement complexe. Regardons maintenant comment simplifier un peu cela.

Dans une application, nous devons gérer un ensemble de personnes. Pour cette gestion nous regroupons nos personnes dans un tableau.

```

Personne[] tab;
tab=new Personne[5];
tab[0] = new Personne("Wayne", "John",LocalDate.of(1907,5,26));
tab[1] = new Personne("McQueen", "Steeve ",LocalDate.of(1930,3,24));
tab[2] = new Personne("Lennon", "John ",LocalDate.of(1940,10,9));
tab[3] = new Personne("Gibson", "Mel ",LocalDate.of(1956,1,3));
tab[4] = new Personne("Willis", "Bruce ",LocalDate.of(1955,3,19));

```

Notre application doit proposer différentes fonctionnalités de recherche d'une personne dans le tableau.

- la recherche par nom
- la recherche par prénom
- la recherche par nom et prénom
- la recherche par âge

Nous devons donc créer quatre fonctions permettant d'effectuer ces recherches.

```

public static Personne rechercheParNom(Personne[] tablo,String nom)
{
    for(Personne p:tablo)
    {
        if (p.getNom().equals(nom))
        {
            return p;
        }
    }
    return null;
}

public static Personne rechercheParPrenom(Personne[] tablo,String prenom)
{
    for(Personne p:tablo)
    {
        if (p.getPrenom().equals(prenom))
        {
            return p;
        }
    }
    return null;
}

public static Personne rechercheParAge(Personne[] tablo,int age)

```

```

{
    for(Personne p:tablo)
    {
        if (p.calculAge()==age)
        {
            return p;
        }
    }
    return null;
}

public static Personne rechercheParNomPrenom(Personne[]
tablo,String nom,String prenom)
{
    for(Personne p:tablo)
    {
        if (p.getNom().equals(nom)&& p.getPrenom().equals(prenom))
        {
            return p;
        }
    }
    return null;
}

```

Ces quatre fonctions ont encore une fois énormément de ressemblances. Seule la ligne de code effectuant la comparaison change d'une version à l'autre. Pour nous permettre de « factoriser » notre code, il peut être intéressant d'extraire de la fonction le code de comparaison.

Commençons par définir une interface décrivant la signature que devra respecter la fonction chargée de vérifier l'égalité de deux personnes.

```

public interface ComparateurPersonne
{
    boolean isIdentique(Personne p);
}

```

Il nous faut maintenant concevoir la nouvelle version de la fonction de recherche d'une personne qui utilise l'interface définie précédemment.

```

public static Personne recherchePersonne(Personne[] tablo,
ComparateurPersonne cp)
{
    for(Personne p:tablo)
    {
        if (cp.isIdentique(p))
        {
            return p;
        }
    }
    return null;
}

```

Il n'y a plus aucune trace de critère de comparaison à l'intérieur de cette fonction. Celui-ci sera défini lors de l'appel de la fonction.

 Le package `java.util.function` propose de nombreuses interfaces contenant des définitions de fonctions dont l'utilisation revient de manière récurrente dans une application. Pour que ces interfaces soient facilement utilisables, elles sont généralement génériques. (voir la section suivante). En utilisant l'une de ces interfaces prédéfinies, notre fonction de recherche peut s'écrire :

```

public static Personne recherchePersonnePrd(Personne[]

```

```

tablo,Predicate<Personne>pr)
{
    for(Personne p:tablo)
    {
        if (pr.test(p))
        {
            return p;
        }
    }
    return null;
}

```

Appliquons maintenant notre première expérience d'écriture d'une expression lambda à cette nouvelle fonction.

```

System.out.println(recherchePersonne(tab, (Personne pe)->
{
    if(pe.getPrenom().equals("Bruce"))
        return true;
    else
        return false;
});

```

Cette syntaxe fonctionne correctement mais reste relativement verbeuse. Regardons maintenant comment simplifier cette expression.

La première simplification porte sur les paramètres de l'expression lambda. Nous ne sommes pas obligés de spécifier le type des paramètres et, en complément, si l'expression n'attend qu'un seul paramètre les parenthèses sont elles aussi facultatives.

```

System.out.println(recherchePersonne(tab, pe->
{
    if(pe.getPrenom().equals("Bruce"))
        return true;
    else
        return false;
});

```

La deuxième simplification possible porte sur le corps de l'expression lambda. Si celle-ci ne contient qu'une seule expression, les accolades sont facultatives ainsi que l'utilisation du mot clé `return`. Dans ce cas, l'expression est évaluée lors de l'exécution et la valeur générée est retournée au code appelant.

```

System.out.println(recherchePersonne(tab, pe->
pe.getPrenom().equals("Bruce")));

```

Si vous utilisez le mot clé `return` dans votre expression, vous devez obligatoirement placer celle-ci dans un bloc de code délimité par des accolades, même si elle ne contient qu'une seule expression.

L'expression lambda peut utiliser les variables disponibles dans le contexte où elle est définie.

```

BufferedReader br;
br=new BufferedReader(new InputStreamReader(System.in));
String prenom;
prenom=br.readLine();

System.out.println(recherchePersonne(tab, pe->
{
    if(pe.getPrenom().equals(prenom))
        return true;
});

```

```

        else
            return false;
    }
});

```

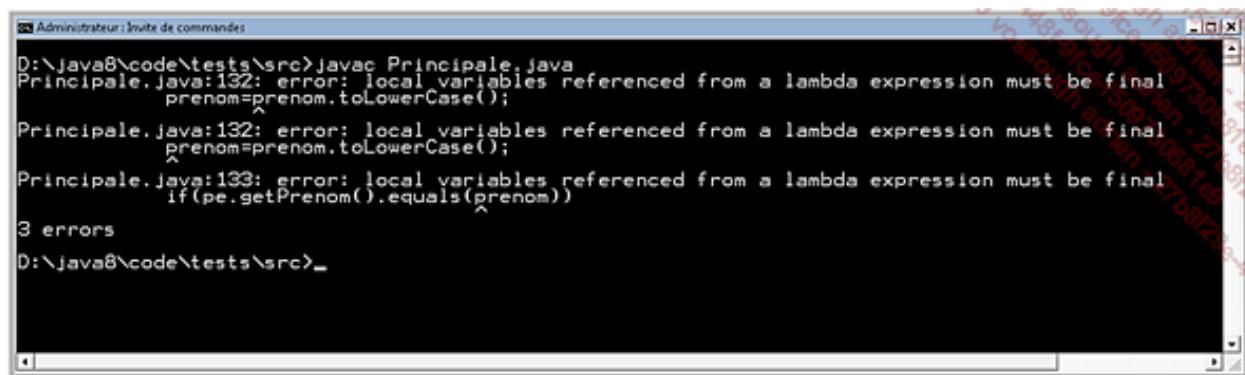
Toutefois, l'expression lambda ne peut pas modifier le contenu de la variable mais uniquement l'utiliser. La modification suivante génère une erreur lors de la compilation.

```

BufferedReader br;
br=new BufferedReader(new InputStreamReader(System.in));
String prenom;
prenom=br.readLine();

System.out.println(recherchePersonne(tab, pe->
{
    prenom=prenom.toLowerCase();
    if(pe.getPrenom().equals(prenom))
        return true;
    else
        return false;
}
));

```



7. Référence de méthode

Lorsqu'une expression lambda devient trop volumineuse ou qu'elle doit être réutilisée à plusieurs emplacements dans l'application, il est préférable de déplacer son contenu dans une fonction et de simplement appeler cette fonction dans le corps de l'expression lambda.

Par exemple, l'expression lambda suivante :

```

tabTrie=(Personne[])tri(tab,(Object o1,Object o2)->
{
    Personne p1,p2;
    if (o1 instanceof Personne & o2 instanceof Personne)
    {
        p1=(Personne)o1;
        p2=(Personne)o2;
    }
    else
    {
        return Classable.ERREUR;
    }
    if (p1.getNom().compareTo(p2.getNom())<0)
    {
        return Classable.INFERIEUR;
    }
    if (p1.getNom().compareTo(p2.getNom())>0)

```

```

    {
        return Classable.SUPERIEUR;
    }

    return Classable.EGAL;
});
```

peut facilement être externalisée dans une fonction.

```

public static int comparePersonne(Object o1, Object o2)
{
    Personne p1, p2;
    if (o1 instanceof Personne & o2 instanceof Personne)
    {
        p1 = (Personne) o1;
        p2 = (Personne) o2;
    }
    else
    {
        return Classable.ERREUR;
    }
    if (p1.getNom().compareTo(p2.getNom()) < 0)
    {
        return Classable.INFERIEUR;
    }
    if (p1.getNom().compareTo(p2.getNom()) > 0)
    {
        return Classable.SUPERIEUR;
    }

    return Classable.EGAL;
}
```

Cette fonction est maintenant utilisable dans plusieurs expressions lambda. Le code de l'expression est d'ailleurs énormément simplifié par cette modification.

```
tabTrie=(Personne[])tri(tab,(Object o1, Object o2)->
comparePersonne(o1, o2));
```

La simplification peut être encore plus poussée en utilisant une référence de méthode. Cette solution consiste simplement à identifier la méthode à utiliser par son nom. L'opérateur `::` doit être utilisé dans cette situation. Il permet d'obtenir une référence vers la fonction à utiliser mais ne provoque pas l'appel de la fonction comme le ferait l'opérateur..

```
tabTrie=(Personne[])tri(tab,Principale::comparePersonne);
```

Cette syntaxe est possible car la fonction est déclarée avec le mot clé `static`. Il n'y a donc pas besoin d'avoir une instance de la classe dans laquelle elle est définie pour pouvoir l'utiliser.

Si la fonction n'est pas déclarée `static`, la règle est la même que pour tous les autres éléments disponibles dans une classe : il faut obligatoirement avoir une instance de la classe pour qu'elle soit utilisable.

Dans ce cas, nous devons utiliser le nom de la variable contenant l'instance de la classe pour pouvoir faire référence à la méthode.

```
TrieusePersonne tp;
tp=new TrieusePersonne();
tabTrie=(Personne[])tri(tab,tp::comparePersonne);
```

8. Les génériques

Les types génériques sont des éléments d'un programme qui s'adaptent automatiquement pour réaliser la

même fonctionnalité sur différents types de données. Lorsque vous créez un élément générique, vous n'avez pas besoin de concevoir une version différente pour chaque type de donnée avec lequel vous souhaitez réaliser une fonctionnalité. Pour faire une analogie avec un objet courant, nous allons prendre l'exemple d'un tournevis. En fonction du type de vis que vous avez à utiliser, vous pouvez prendre un tournevis spécifique pour ce type de vis (plat, cruciforme, torx...). Une technique fréquemment utilisée par un bricoleur averti consiste à acquérir un tournevis universel avec de multiples embouts. En fonction du type de vis, il choisit l'embout adapté. Le résultat final est le même que s'il disposait d'une multitude de tournevis différents : il peut visser et dévisser.

Lorsque vous utilisez un type générique, vous le paramétrez avec un type de données. Ceci permet au code de s'adapter automatiquement et de réaliser la même action indépendamment du type de données. Une alternative pourrait être l'utilisation du type universel `Object`. L'utilisation des types génériques présente plusieurs avantages par rapport à cette solution :

- Elle impose la vérification des types de données au moment de la compilation et évite les inévitables vérifications qui doivent être faites manuellement avec l'utilisation du type `Object`.
- Elle évite les opérations de conversion du type `Object` vers un type plus spécifique et inversement.
- L'écriture du code est facilitée dans certains environnements de développement avec l'affichage automatique de tous les membres disponibles pour un type de données particulier.
- Elle favorise l'écriture d'algorithmes qui sont indépendants des types de données.

Les types génériques peuvent cependant imposer certaines restrictions concernant le type de données utilisé. Ils peuvent par exemple imposer que le type utilisé implémente une ou plusieurs interfaces, soit un type référence ou possède un constructeur par défaut. Il est important de bien comprendre quelques termes utilisés avec les génériques.

- Le type générique : c'est la définition d'une classe, interface ou fonction pour laquelle vous spécifiez au moins un type de données au moment de sa déclaration.
- Le type de paramètre : c'est l'emplacement réservé pour le type de paramètre dans la déclaration du type générique.
- Le type argument : c'est le type de données qui remplace le type de paramètre lors de la construction d'un type à partir d'un type générique.
- Les contraintes : ce sont les conditions que vous imposez qui limitent le type argument que vous pouvez fournir.
- Le type construit : c'est la classe, interface, ou fonction construite à partir d'un type générique pour lequel vous avez spécifié des types argument.

a. Classes génériques

Une classe qui attend un type de paramètre est appelée classe générique. Vous pouvez générer une classe construite en fournissant à la classe générique un type argument pour chacun de ces types paramètre.

Définition d'une classe générique

Vous pouvez définir une classe générique qui fournit les mêmes fonctionnalités sur différents types de données. Pour cela, vous devez fournir un ou plusieurs types de paramètre dans la définition de la classe. Prenons l'exemple d'une classe capable de gérer une liste d'éléments avec les fonctionnalités suivantes.

- Ajouter un élément.
- Supprimer un élément.
- Se déplacer sur le premier élément.
- Se déplacer sur le dernier élément.
- Se déplacer sur l'élément suivant.

- Se déplacer sur l'élément précédent.
- Obtenir le nombre d'éléments.

Nous devons tout d'abord définir la classe comme une classe ordinaire.

```
public class ListeGenerique
{
}
```

La transformation de cette classe en classe générique se fait en ajoutant un type de paramètre immédiatement après le nom de la classe.

```
public class ListeGenerique <T>
{
}
```

Si plusieurs types paramètres sont nécessaires, ils doivent être séparés par des virgules dans la déclaration. Par convention les types paramètres sont représentés par un caractère majuscule unique.

Si le code de la classe doit réaliser d'autres opérations que des affectations, vous devez ajouter des contraintes sur le type paramètre. Pour cela, ajoutez le mot clé `extends` suivi de la contrainte. La contrainte peut être constituée par une classe spécifique dont le type argument devra hériter ou par une ou plusieurs interfaces qu'il devra implémenter. Si plusieurs contraintes doivent s'appliquer, elles sont séparées par le caractère `&`. Il faut dans ce cas spécifier en début de liste la contrainte liée à une classe puis celles liées aux interfaces. Ci-dessous quelques exemples pour illustrer cela.

Classe générique exigeant que le type argument hérite de la classe `Personne` :

```
public class ListeGenerique <T extends Personne>
{
}
```

Classe générique exigeant que le type argument implémente l'interface `Classable` :

```
public class ListeGenerique <T extends Classable>
{
}
```

 C'est le même mot clé `extends` qui est utilisé pour une contrainte liée à une classe ou à une interface.

Classe générique exigeant que le type argument hérite de la classe `Personne` et implémente les interfaces `Classable` et `Cloneable` :

```
public class ListeGenerique <T extends Personne & Classable & Cloneable>
{
}
```

S'il n'y a pas de contrainte spécifiée, les seules opérations autorisées seront celles supportées par le type `Object`.

Dans le code de la classe, chaque membre qui doit être du type argument doit être défini avec le type paramètre, `T` dans notre cas. Voyons maintenant le code complet de la classe.

```
import java.util.ArrayList;
public class ListeGenerique <T>
{
    // pour stocker les éléments de la liste
    private ArrayList<T> liste;
```

```
// pointeur de position dans la liste
private int position;
// nombre d'éléments de la liste
private int nbElements;
// constructeur avec un paramètre permettant de dimensionner
// la liste
public ListeGenerique(int taille)
{
    liste=new ArrayList<T>(taille);
}
public void ajout(T element)
{
    liste.add(element);
    nbElements = nbElements + 1;
}
public void insert(T element,int index)
{
    // on vérifie si l'index n'est pas supérieur au nombre
    // d'éléments ou si l'index n'est pas inférieur à 0
    if (index >= nbElements || index < 0)
    {
        return;
    }
    liste.add(index,element);
    // on met à jour le nombre d'éléments
    nbElements = nbElements + 1;
}

public void remplace(T element,int index)
{
    // on vérifie si l'index n'est pas supérieur au nombre
    // d'éléments ou si l'index n'est pas inférieur à 0
    if (index >= nbElements || index < 0)
    {
        return;
    }
    liste.set(index,element);
}
public void supprime(int index)
{
    int i;
    // on vérifie si l'index n'est pas supérieur au nombre
    // d'éléments ou si l'index n'est pas inférieur à 0
    if (index >= nbElements || index < 0)
    {
        return;
    }
    liste.remove(index);
    // on met à jour le nombre d'éléments
    nbElements = nbElements - 1;
}
public T getElement(int j)
{
    return liste.get(j);
}
public int getNbElements()
{
    return nbElements;
}
public T premier() throws Exception
{
    if (nbElements == 0)
    {
        throw new Exception("liste vide");
    }
    // on déplace le pointeur sur le premier élément
    position = 0;
    return liste.get(0);
}
public T dernier() throws Exception
```

```

{
    if (nbElements == 0)
    {
        throw new Exception("liste vide");
    }
    // on déplace le pointeur sur le dernier élément
    position = nbElements - 1;
    return liste.get(position);
}
public T suivant() throws Exception
{
    if (nbElements == 0)
    {
        throw new Exception("liste vide");
    }
    // on vérifie si on n'est pas à la fin de la liste
    if (position == nbElements - 1)
    {
        throw new Exception("pas d'element suivant");
    }
    // on déplace le pointeur sur l'élément suivant
    position = position + 1;
    return liste.get(position);
}
public T precedent() throws Exception
{
    if (nbElements == 0)
    {
        throw new Exception("liste vide");
    }
    // on vérifie si on n'est pas sur le premier élément
    if (position == 0)
    {
        throw new Exception("pas d'élément précédent");
    }
    // on se déplace sur l'élément précédent
    position = position - 1;
    return liste.get(position);
}
}

```

Utilisation d'une classe générique

Pour pouvoir utiliser une classe générique, vous devez tout d'abord générer une classe construite en fournissant un type argument pour chacun de ces types paramètre. Vous pouvez alors instancier la classe construite par un des constructeurs disponibles. Nous allons utiliser la classe conçue précédemment pour travailler avec une liste de chaînes de caractères.

```
ListeGenerique<String> liste = new ListeGenerique<String>(5);
```

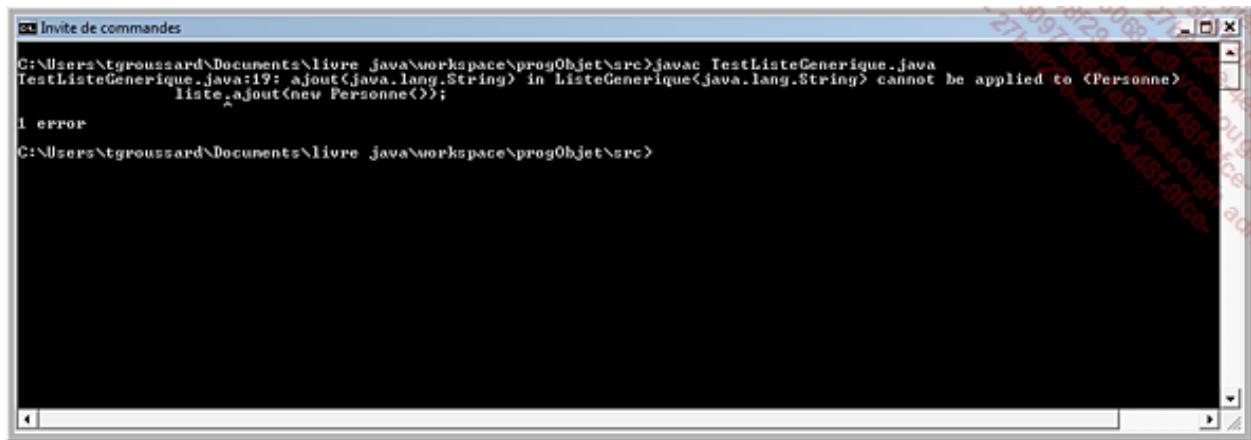
Cette syntaxe peut être simplifiée en laissant le compilateur déterminer lui-même le type argument à utiliser lors de l'appel du constructeur. Il suffit simplement d'omettre le type argument entre les caractères < et >.

```
ListeGenerique<String> liste=new ListeGenerique<>(5);
```

Cette déclaration permet d'instancier une liste de cinq chaînes. Les méthodes de la classe sont alors disponibles.

```
liste.ajout("premier") ;
liste.ajout("deuxieme");
```

Le compilateur vérifie bien sûr que nous utilisons notre classe correctement notamment en vérifiant les types de données que nous lui confions.



Voici le code d'une petite application permettant de tester le bon fonctionnement de notre classe générique :

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
public class TestListeGenerique
{
    static ListeGenerique<String> liste = new
ListeGenerique<String>(5);
    public static void main(String[] args)
    {
        liste.ajout("premier");
        liste.ajout("deuxième");
        liste.ajout("troisième");
        liste.ajout("quatrième");
        liste.ajout("cinquième");
        menu();
    }
    public static void menu()
    {
        char choix='\0';
        System.out.println("p (premier) < (précédent) >(suivant) d
(dernier) f (fin)");
        while (choix != 'f')
        {
            try
            {
                BufferedReader br;
                br=new BufferedReader(new
InputStreamReader(System.in));
                choix=br.readLine().charAt(0);
                switch (choix)
                {
                    case 'p':
                        System.out.println("le premier " +
liste.premier());
                        break;
                    case '<':
                        System.out.println("le précédent " +
liste.precedent());
                        break;
                    case '>':
                        System.out.println("le suivant " + liste.suivant());
                        break;
                    case 'd':
                        System.out.println("le dernier " +
liste.dernier());
                        break;
                }
            }
            catch (Exception e)
            {

```

```

        System.out.println(e.getMessage());
    }
    System.out.println("p (premier) < (précédent) >(suivant) d
(dernier) f (fin)");
}
}
}

```

Nous pouvons également vérifier que notre classe fonctionne sans problème si nous lui demandons de travailler avec des personnes.

```

liste.ajout(new Personne("toto2","prenom2",LocalDate.of(1922,2,15)));
liste.ajout(new Personne("toto1","prenom1",LocalDate.of(1911,1,15)));
liste.ajout(new Personne("toto5","prenom5",LocalDate.of(1955,5,15)));
liste.ajout(new Personne("toto3","prenom3",LocalDate.of(1933,3,15)));
liste.ajout(new Personne("toto4","prenom4",LocalDate.of(1944,4,15)));

```

b. Méthodes génériques

Une fonction générique est une méthode définie avec au moins un type paramètre. Ceci permet au code appelant de spécifier le type de données dont il a besoin à chaque appel de la fonction. Une telle méthode peut cependant être utilisée sans indiquer d'informations pour le type argument. Dans ce cas le compilateur essaie de déterminer le type en fonction des arguments qui sont passés à la méthode. Pour illustrer la création de fonctions génériques, nous allons transformer la fonction de tri en version générique.

```

public static <T extends Classable> void
tri(ListeGenerique<T> liste) throws Exception
{
    int i,j;
    T c;
    for (i=0;i< liste.getNbElements()-1;i++)
    {
        for( j = i + 1; j<liste.getNbElements();j++)
        {
            if
(liste.getElement(j).compare(liste.getElement(i))==Classable.
INFERIEUR)
            {
                c = liste.getElement(j);
                liste.reemplace(liste.getElement(i), j);
                liste.reemplace(c,i);
            }
            else if
(liste.getElement(j).compare(liste.getElement(i))==Classable.
ERREUR)
            {
                throw new Exception("erreur pendant le tri");
            }
        }
    }
}

```

Comme pour la création d'une classe générique le type paramètre est spécifié entre les caractères < et >. Si des contraintes doivent s'appliquer aux types arguments, vous devez les spécifier avec les mêmes règles que pour les classes génériques.

L'appel d'une fonction générique est semblable à l'appel d'une fonction normale hormis le fait que nous pouvons spécifier des types arguments pour chacun des types paramètre. Ceci n'est pas une obligation car le compilateur est capable d'inférer les types arguments au moment de la compilation.

L'appel de la fonction peut donc se faire avec les deux syntaxes suivantes :

```
TestListeGenerique.<Personne>tri(liste);
```

ou

```
TestListeGenerique.tri(liste);
```

c. Les génériques et l'héritage

L'utilisation combinée des génériques et de l'héritage peut parfois provoquer quelques soucis au développeur d'une application. Nous avons déterminé dans le paragraphe consacré à l'héritage qu'avec une variable d'un certain type l'on pouvait référencer une instance de classe de ce type mais aussi une instance de classe de n'importe lequel de ses sous-types. Le code suivant est donc parfaitement légal et fonctionne correctement.

```
Personne p;
Client c;
c=new Client();
p=c;
p.setNom("Dupont");
p.setPrenom("paul");
```

Si nous tentons la même expérience avec les génériques en testant le code suivant.

```
ListeGenerique<Personne> listePersonnes;
ListeGenerique<Client> listeClient;
listeClient=new ListeGenerique<Client>(10);
listePersonnes=listeClient;
```

Il est effectivement légitime de penser que, puisque l'on peut affecter à une variable de type Personne une référence vers une instance d'une de ses sous-classes (Client), la même opération est certainement réalisable avec une `ListeGenerique<Personne>` et une `ListeGenerique<Client>`. Pourtant le compilateur ne voit pas les choses de la même façon.

```
E:\livre\java\workspace\progObjet\src>javac TestListeGenerique2.java
TestListeGenerique2.java:19: incompatible types
found : ListeGenerique<Personne>
required: ListeGenerique<Client>
    listeClient=listePersonnes;

TestListeGenerique2.java:20: incompatible types
found : ListeGenerique<Client>
required: ListeGenerique<Personne>
    listePersonnes=listeClient;

2 errors
E:\livre\java\workspace\progObjet\src>_
```

Cette limitation est en fait liée au mécanisme d'effacement de type utilisé par le compilateur. Son principal but est de rendre compatible avec les versions antérieures des machines virtuelles Java le code compilé.

En fait lors de la compilation d'une classe générique, le compilateur remplace le type paramètre par le type Object ou par un type correspondant à la contrainte placée sur le type paramètre. Le code de la classe `ListeGenerique` sera traité de la façon suivante par le compilateur.

```
import java.util.ArrayList;
public class ListeGenerique <T>
```

```

{
    // pour stocker les éléments de la liste
    private ArrayList<T Object> liste;
    // pointeur de position dans la liste
    private int position;
    //nombre d'éléments de la liste
    private int nbElements;
    // constructeur avec un paramètre permettant
    // de dimensionner la liste
    public ListeGenerique(int taille)
    {
        liste=new ArrayList<T Object>(taille);
    }
    public void ajout(T Object element)
    {
        liste.add(element);
        nbElements = nbElements + 1;
    }
    public void insert(T Object element,int index)
    {
        // on vérifie si l'index n'est pas supérieur au nombre
        // d'éléments ou si l'index n'est pas inférieur à 0
        if (index >= nbElements || index < 0)
        {
            return;
        }
        liste.add(index,element);
        // on met à jour le nombre d'éléments
        nbElements = nbElements + 1;
    }
    public T Object premier() throws Exception
    {
        if (nbElements == 0)
        {
            throw new Exception("liste vide");
        }
        // on déplace le pointeur sur le premier élément
        position = 0;
        return liste.get(0);
    }
    public T Object dernier() throws Exception
    {
        if (nbElements == 0)
        {
            throw new Exception("liste vide");
        }
        // on déplace le pointeur sur le dernier élément
        position = nbElements - 1;
        return liste.get(position);
    }
...
...
}

```

Pour en être convaincu, essayons le code suivant dans la classe `ListeGenerique`.

```

public void ajout(T element)
{
    liste.add(element);
    nbElements = nbElements + 1;
}
public void ajout(Object element)
{
    liste.add(element);
    nbElements = nbElements + 1;
}

```

À la compilation, nous avons bien un message d'erreur nous indiquant que la méthode `ajout (Objet)` est

définie deux fois dans la classe.

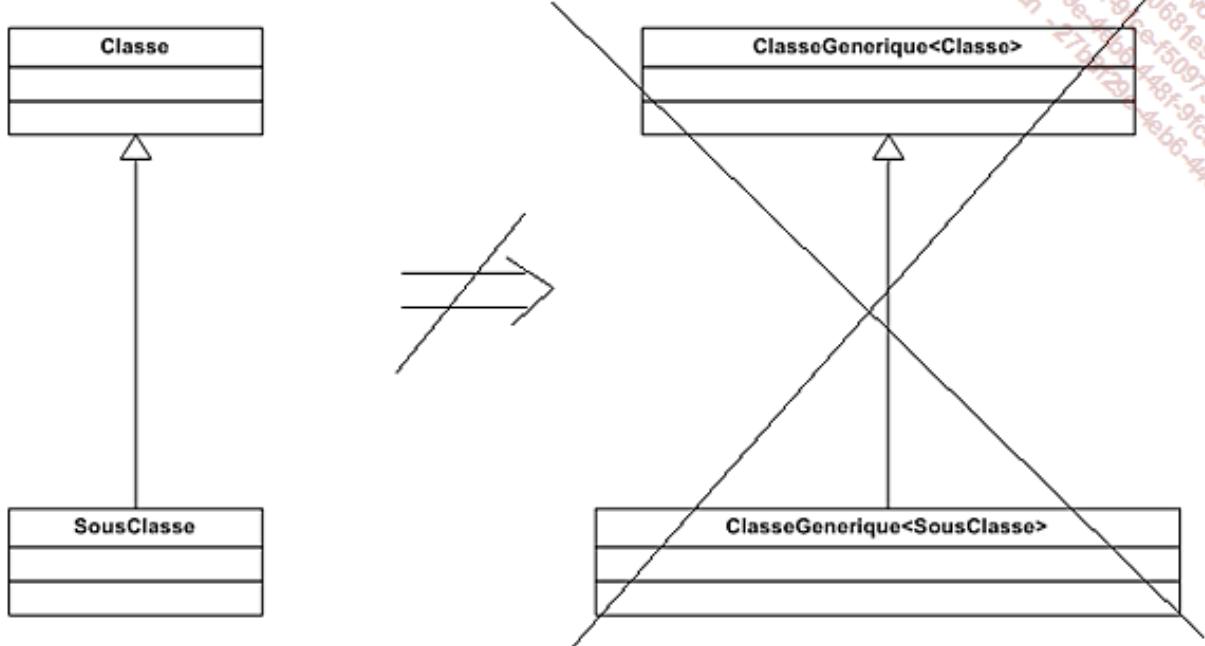
Une autre opération réalisée par le compilateur consiste à effectuer une conversion pour les valeurs renvoyées par les fonctions de la classe générique. Le code suivant :

```
static ListeGenerique<Personne> liste = new
ListeGenerique<Personne>(5);
public static void main(String[] args) throws Exception
{
    liste.ajout(new Personne("toto2","prenom2",
LocalDate.of(1922,2,15)));
    liste.ajout(new Personne("toto1","prenom1",
LocalDate.of(1911,1,15)));
    liste.ajout(new Personne("toto5","prenom5",
LocalDate.of(1955,5,15)));
    liste.ajout(new Personne("toto3","prenom3",
LocalDate.of(1933,3,15)));
    liste.ajout(new Personne("toto4","prenom4",
LocalDate.of(1944,4,15)));
    Personne p;
    p=liste.getElement(0);
}
```

sera en fait implicitement compilé sous cette forme :

```
static ListeGenerique<Personne> liste = new
ListeGenerique<Personne>(5);
public static void main(String[] args) throws Exception
{
    liste.ajout(new Personne("toto2","prenom2",
LocalDate.of(1922,2,15)));
    liste.ajout(new Personne("toto1","prenom1",
LocalDate.of(1911,1,15)));
    liste.ajout(new Personne("toto5","prenom5",
LocalDate.of(1955,5,15)));
    liste.ajout(new Personne("toto3","prenom3",
LocalDate.of(1933,3,15)));
    liste.ajout(new Personne("toto4","prenom4",
LocalDate.of(1944,4,15)));
    Personne p;
    p=(Personne)liste.getElement(0);
}
```

Il faut donc retenir de ces expériences qu'une classe générique construite avec comme type paramètre une classe quelconque n'est pas la sous-classe d'une classe générique construite avec un type qui est le super-type de cette classe quelconque.



Cette limitation peut parfois être gênante. Si nous souhaitons créer une fonction qui accepte comme paramètre une `ListeGenerique` construite à partir de n'importe quel type de données, notre première intuition est d'écrire le code suivant :

```

public static void affichage(ListeGenerique<Object> liste)
{
}

```

Hélas, notre première intuition n'est pas bonne car la fonction `affichage` n'accepte, comme paramètre, avec cette syntaxe, que des instances de `ListeGenerique d'Object`.

Pour représenter le fait que la fonction `affichage` accepte comme paramètre une instance de `ListeGenerique` construite à partir de n'importe quel type, nous devons utiliser le caractère joker "?" dans la définition de la fonction `affichage`. Elle prend donc la forme suivante :

```

public static void affichage(ListeGenerique<?> liste)
{
}

```

Avec cette syntaxe nous pouvons utiliser la fonction `affichage` avec comme paramètre, une instance de `ListeGenerique` construite à partir de n'importe quelle classe. Par contre, dans le code de la fonction nous ne pourrons utiliser que les méthodes de la classe `Object` sur les éléments présents dans la liste. Nous pouvons ajouter des restrictions sur le type argument de la fonction en faisant suivre le caractère "?" par une contrainte définie avec exactement la même syntaxe que pour une classe générique. Pour que la fonction `affichage` accepte comme paramètre une `ListeGenerique dePersonne` et de n'importe laquelle de ses sous-classes, nous devons utiliser la syntaxe suivante :

```

public static void affichage(ListeGenerique<? extends Personne> liste)
{
    liste.premier();
    for (int i=0;i<liste.getNbElements();i++)
    {
        System.out.println(liste.getElement(i).getNom());
        System.out.println(liste.getElement(i).getPrenom());
        System.out.println("-----");
    }
}

```

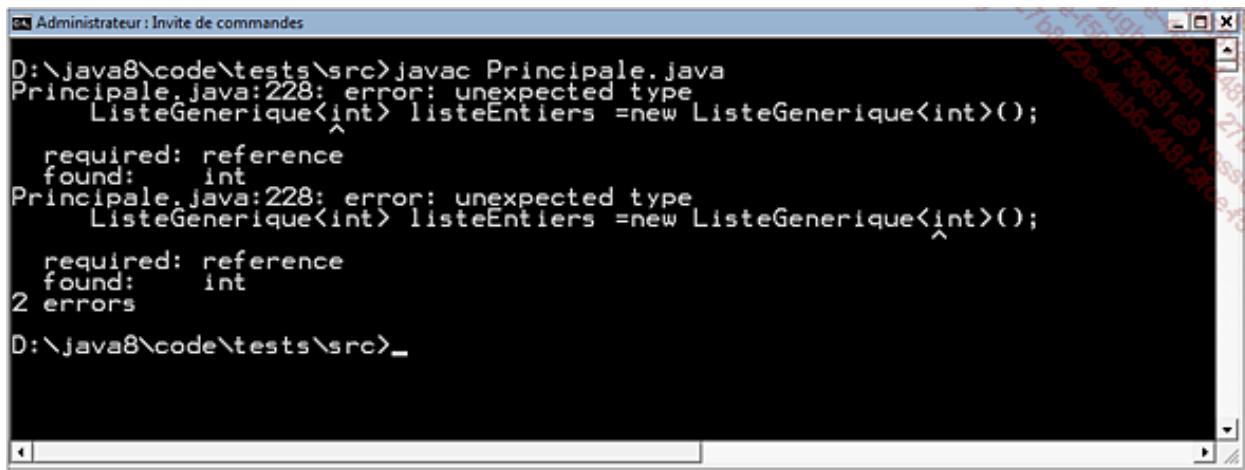
À noter que dans ce cas les instances présentes dans la liste seront au moins des instances de la classe Personne et pourront être utilisées comme telles en toute sécurité à l'intérieur de la fonction.

d. Limitation des génériques

L'utilisation des types génériques impose quelques contraintes. Ces limitations sont liées au type argument utilisé et à l'usage qui en est fait.

a - Le type argument est obligatoirement un type par référence. Par exemple, il nous est impossible de créer une instance de ListeGenerique de type int.

```
ListeGenerique<int> listeEntiers =new ListeGenerique<int>();
```



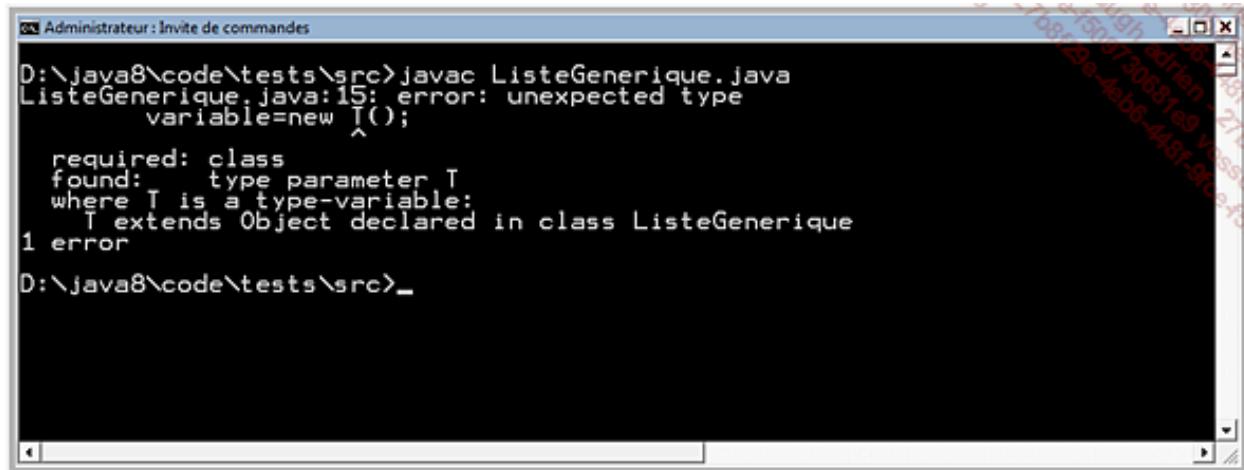
The screenshot shows a Windows Command Prompt window titled "Administrateur : Invite de commandes". The command entered is "D:\java8\code\tests\src>javac Principale.java". The output shows two errors:

```
D:\java8\code\tests\src>javac Principale.java
Principale.java:228: error: unexpected type
    ListeGenerique<int> listeEntiers =new ListeGenerique<int>();
                           ^
required: reference
found:    int
Principale.java:228: error: unexpected type
    ListeGenerique<int> listeEntiers =new ListeGenerique<int>();
                           ^
required: reference
found:    int
2 errors
```

Cette limitation peut être contournée en utilisant à la place les classes wrapper correspondant aux types simples. Dans notre cas, l'utilisation de la classe Integer résout le problème. Les mécanismes d'autoboxing et d'unboxing rendent transparente l'utilisation des types wrapper en lieu et place des types simples.

```
ListeGenerique<Integer> listeEntiers =
new ListeGenerique<Integer>(5);
listeEntiers.ajout(10);
listeEntiers.ajout(7);
listeEntiers.ajout(19);
listeEntiers.ajout(45);
listeEntiers.ajout(8);
```

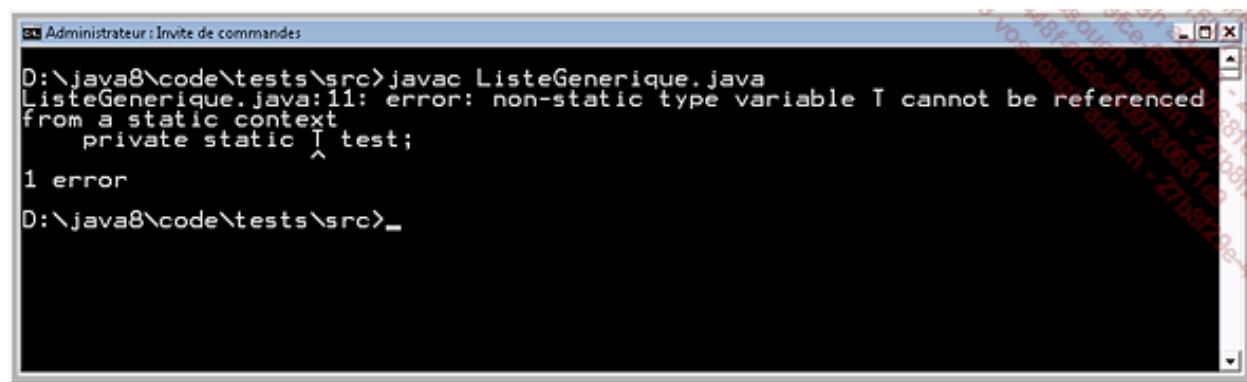
b - Le type paramètre ne peut pas être utilisé pour la création d'une instance de classe à l'intérieur d'un type générique. Le code suivant est par exemple interdit à l'intérieur de la classe ListeGenerique.



The screenshot shows a Windows Command Prompt window titled "Administrateur : Invite de commandes". The command entered is "D:\java8\code\tests\src>javac ListeGenerique.java". The output shows one error:

```
D:\java8\code\tests\src>javac ListeGenerique.java
ListeGenerique.java:15: error: unexpected type
    variable=new T();
                   ^
required: class
found:   type parameter T
where T is a type-variable:
    T extends Object declared in class ListeGenerique
1 error
```

c - Le type paramètre ne peut pas non plus être utilisé pour déclarer des variables de classe (static) à l'intérieur d'un type générique.



```
D:\java8\code\tests\src>javac ListeGenerique.java
ListeGenerique.java:11: error: non-static type variable T cannot be referenced
from a static context
    private static T test;
           ^
1 error
D:\java8\code\tests\src>_
```

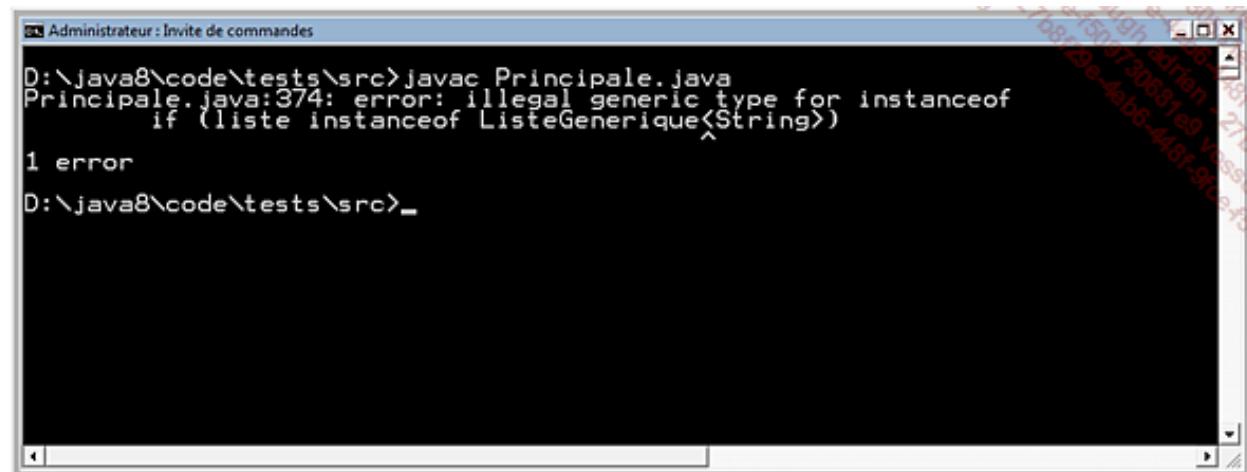
La variable `test` étant unique pour toutes les instances de la classe `ListeGenerique`, son type ne peut pas changer en fonction du type argument utilisé pour la création de chacune des instances. Par exemple avec la ligne de code suivante la variable `test` serait de type `String`.

```
ListeGenerique<String> listeChaine=new ListeGenerique<String>(5);
```

Si une autre instance de la classe `ListeGenerique` est ensuite créée avec un type argument différent, il faudrait que la variable de classe `test` change de type, ce qui est bien sûr impossible.

d - L'opérateur `instanceOf` ne peut pas être utilisé avec des types génériques. Le mécanisme d'effacement de type faisant disparaître toute trace du type argument, après la compilation, le test suivant ne pourrait pas fonctionner.

```
public static <T extends Classable> void tri(ListeGenerique<T>
liste) throws Exception
{
    int i,j;
    T c;
    if (liste instanceof ListeGenerique<String>)
    {
        ...
        ...
    }
```



```
D:\java8\code\tests\src>javac Principale.java
Principale.java:374: error: illegal generic type for instanceof
    if (liste instanceof ListeGenerique<String>)
                           ^
1 error
D:\java8\code\tests\src>_
```

9. Les packages

Le but principal des packages est l'organisation et le rangement des classes et interfaces d'une application. Le principe est le même que dans la vie courante. Si vous avez un grand placard dans votre maison, il est évident que l'installation d'étagères dans ce placard va faciliter l'organisation et la recherche des objets qui y

sont rangés par rapport à un stockage en "vrac". L'organisation des classes en package apporte les avantages suivants :

- Facilité pour retrouver et réutiliser un élément.
- Limitation des risques de conflits de noms.
- Création d'une nouvelle visibilité en plus des visibilités standards (`private`, `protected`, `public`).

a. Crédation d'un package

La première chose à faire lorsque l'on souhaite créer un package est de lui trouver un nom. Ce nom doit être choisi avec précaution pour permettre au package de remplir pleinement son rôle. Il doit être unique et représentatif des éléments stockés à l'intérieur.

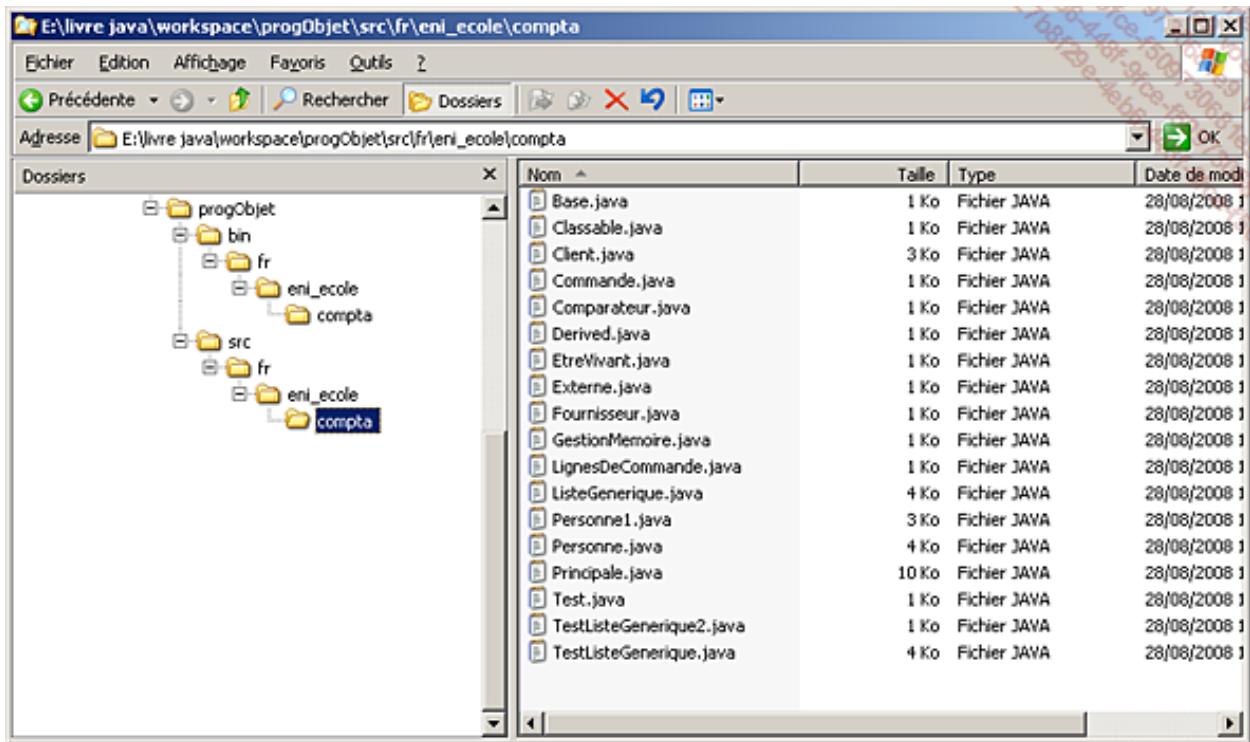
Pour assurer l'unicité du nom d'un package, on utilise par convention son nom de domaine en inversant l'ordre des éléments comme première partie pour le nom du package.

Par exemple, si le nom de domaine est eni.fr, la première partie du nom du package sera fr.eni. Si le nom de domaine comporte des caractères interdits pour les noms de packages, ils sont remplacés par le caractère _. Ainsi pour le nom de domaine eni-ecole.fr, la première partie du nom de package sera fr.eni_ecole. Cette première partie du nom permet d'assurer l'unicité des éléments par rapport à l'extérieur. La suite du nom de package va assurer l'unicité des éléments à l'intérieur de l'application. Il faut bien sûr choisir des noms clairs et représentatifs des éléments du package. Nous pouvons par exemple avoir dans une application deux classes Client l'une représentant une personne ayant passé des commandes à notre entreprise, l'autre représentant un client au sens informatique du terme (client - serveur). Les noms de package suivant peuvent être utilisés pour héberger ces deux classes `fr.eni_ecole.compta` pour la classe représentant un client physique et `fr.eni_ecole.reseau` pour la classe représentant un client logiciel. Par convention les noms de packages sont entièrement en lettres minuscules. Le nom choisi doit être placé dans le fichier source précédé du mot clé `package`. La déclaration du package doit être obligatoirement la première ligne du fichier. Tous les éléments placés dans ce fichier source font désormais partie de ce package. Si aucune information n'est indiquée concernant le package dans le fichier source alors les éléments définis dans ce fichier seront considérés comme faisant partie du package par défaut qui ne possède pas de nom. Il convient d'être prudent avec le package par défaut car les classes qui y sont définies ne sont pas accessibles à partir d'un package nommé.

C'est entre autres pour cette raison que les concepteurs de Java recommandent de toujours placer une classe dans un package nommé.

L'utilisation des packages nous impose également une organisation spécifique pour l'enregistrement sur disque des fichiers sources et des fichiers compilés. Les répertoires où sont enregistrés les fichiers doivent respecter les noms utilisés pour les packages. Chaque composant du nom de package doit correspondre à un sous-répertoire. Les fichiers sources et les fichiers compilés peuvent être stockés dans des branches d'arborescence différentes.

Nous pouvons par exemple avoir l'organisation suivante :



Le respect de cette arborescence est impératif. Si le fichier d'une classe appartenant à un package n'est pas placé dans le bon répertoire, la machine virtuelle chargée d'exécuter l'application sera incapable de localiser cette classe. Pour localiser une classe, la machine virtuelle utilise la variable d'environnement `CLASSPATH` pour construire le chemin d'accès au fichier de cette classe. Elle concatène le chemin contenu dans cette variable avec le chemin correspondant au package dans lequel se trouve la classe. Si plusieurs répertoires sont indiqués dans la variable `CLASSPATH`, ils doivent être séparés par des points-virgules pour un environnement Windows et par des virgules pour un environnement Unix. Le répertoire courant fait partie par défaut du chemin de recherche.

b. Utilisation et importation d'un package

Lorsqu'une classe faisant partie d'un package est utilisée dans une application, le nom complet de la classe (nom du package + nom de la classe) doit être utilisé. Ceci provoque l'écriture de lignes de code relativement longues et difficiles à relire.

```
fr.eni_ecole.compta.Client c=new fr.eni_ecole.compta.Client
("dupont","paul",LocalDate.of(1953,11,8),'E');
```

Java propose l'instruction `import` pour indiquer au compilateur que certaines classes peuvent être utilisées directement par leur nom sans utilisation du nom de package. Cette instruction doit être placée en début de fichier aussitôt après une éventuelle déclaration de package mais avant la définition de la classe. La ligne de code précédente peut être considérablement abrégée avec la syntaxe suivante :

```
import fr.eni_ecole.compta.Client;
...
Client c=new Client("dupont","paul", LocalDate.of(1953,11,8),'E');
```

Par contre, si beaucoup de classes du même package sont utilisées, la liste des importations va devenir volumineuse. Il est plus rationnel dans ce cas d'importer le package complet avec le caractère joker `*`.

Le code devient donc :

```
import fr.eni_ecole.compta.*;
```

```
...  
...
```

```
Client c=new Client("dupont","paul", LocalDate.of(1953,11,8), 'E');
```

Avec la possibilité d'utiliser toutes les classes du package `fr.eni_ecole.compta` simplement en utilisant le nom de la classe. Cette solution peut parfois poser un problème si deux packages contenant des classes ayant des noms identiques sont importés. Il faudra dans ce cas, pour ces classes, utiliser leurs noms complets. Cette solution ne permet pas non plus l'importation de plusieurs packages simultanément. Les apparences étant parfois trompeuses, la structure de packages n'est pas hiérarchique et, malgré ce que pourrait laisser penser la structure des fichiers sur le disque, le package `fr.eni_ecole.compta` n'est pas inclus dans le package `fr.eni_ecole`. L'importation `import fr.eni_ecole.*;` permet uniquement l'accès aux classes de ce package et absolument pas aux classes du package `fr.eni_ecole.compta`. Si c'était le cas, on peut imaginer facilement les problèmes posés par les importations suivantes :

```
import com.*;  
import fr.*;
```

L'importation peut également simplifier l'écriture du code lors de l'utilisation de classes contenant de nombreuses méthodes `static`. Vous devez pour chaque appel d'une de ces méthodes la préfixer par le nom de la classe. Ceci a pour conséquence de réduire la lisibilité du code.

```
public class Distance  
{  
  
    public static double dist(double lat1,double lon1,double lat2,double lon2)  
    {  
        double deltaLat;  
        double abscurv;  
        deltaLat =lon2-lon1;  
        abscurv=Math.acos((Math.sin(lat1)*Math.sin(lat2))+(Math.cos(lat1)  
*Math.cos(lat2)*Math.cos(deltaLat)));  
        return abscurv * 6371598;  
    }  
}
```

Il est possible de simplifier ce code en effectuant une importation `static` de la classe `Math` permettant l'utilisation de ces membres `static` sans les préfixer par le nom de la classe.

```
import static java.lang.Math.*;  
public class Distance  
{  
    public static double dist(double lat1,double lon1,double lat2,double lon2)  
    {  
        double deltaLat;  
        double abscurv;  
        deltaLat =lon2-lon1;  
        abscurv=acos((sin(lat1)*sin(lat2))+(cos(lat1)*cos(lat2)*cos(  
deltaLat)));  
        return abscurv; /*6371598;  
    }  
}
```

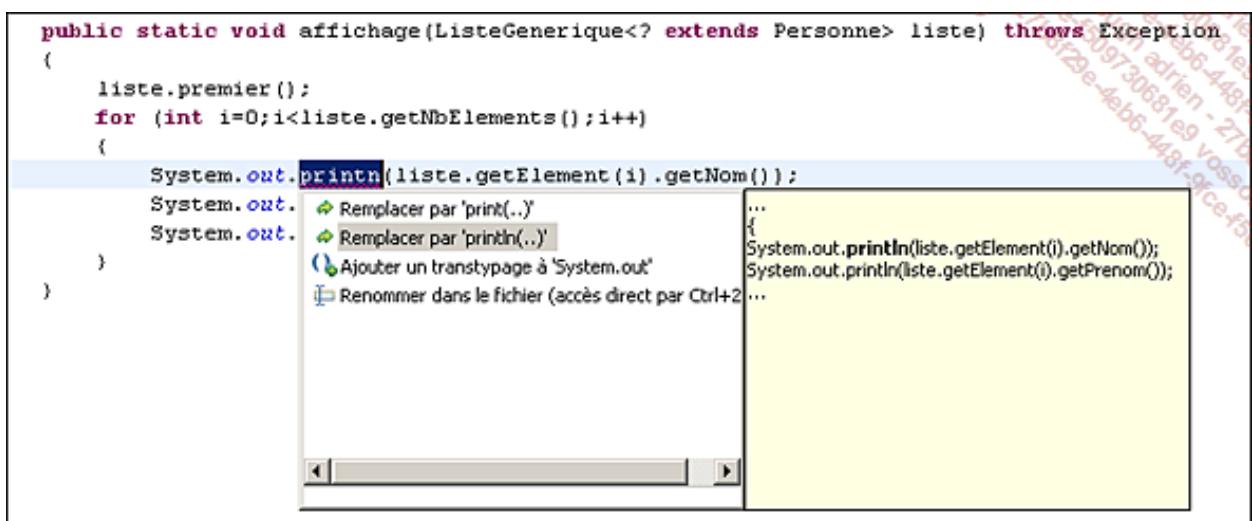
Gestion des exceptions

Dans la vie d'un développeur, tout n'est pas rose ! Les erreurs sont une des principales sources de stress. En fait, lorsque l'on y regarde de plus près, nous pouvons classer ces erreurs qui nous gâchent la vie en trois catégories. Regardons chacune d'entre elles et les solutions disponibles pour les traiter.

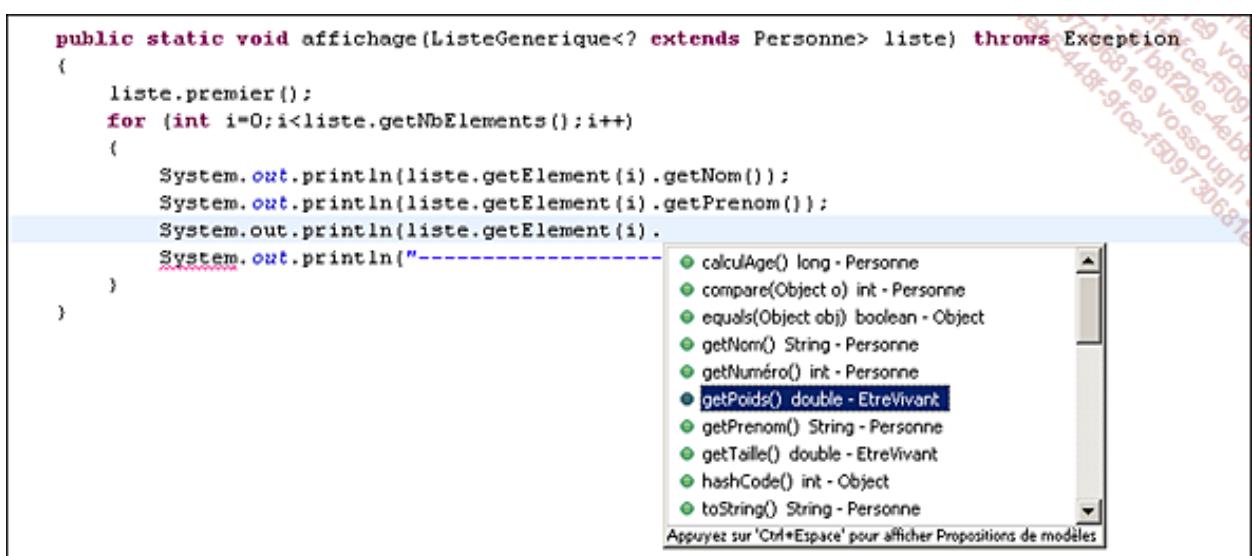
1. Les erreurs de syntaxe

Ce type d'erreur se produit au moment de la compilation lorsqu'un mot clé du langage est mal orthographié. Très fréquentes avec les outils de développement où l'éditeur de code et le compilateur sont deux entités séparées, elles deviennent de plus en plus rares avec les environnements de développement intégré (Eclipse, NetBeans, Jbuilder...). La plupart de ces environnements proposent une analyse syntaxique au fur et à mesure de la saisie du code. Les exemples suivants sont obtenus à partir de l'environnement Eclipse.

Si une erreur de syntaxe est détectée, alors l'environnement propose des solutions possibles pour corriger cette erreur.



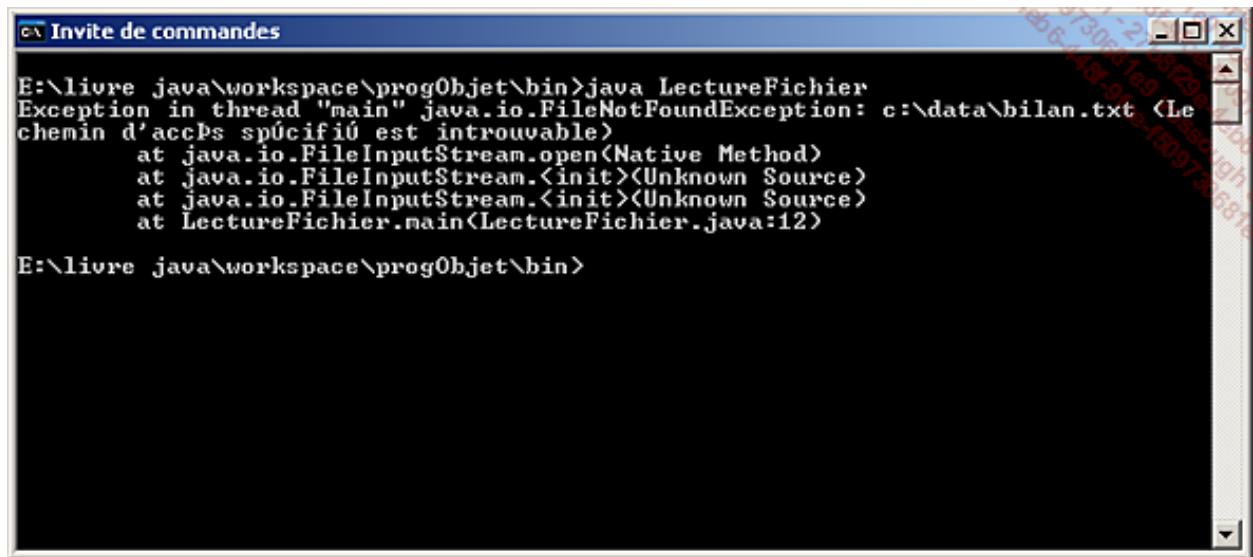
D'autre part, les "fautes d'orthographe" dans les noms de champs ou de méthodes sont facilement éliminées grâce aux fonctionnalités disponibles dans ces environnements.



2. Les erreurs d'exécution

Ces erreurs apparaissent après la compilation lorsque vous lancez l'exécution de votre application. La

La syntaxe du code est correcte mais l'environnement de votre application ne permet pas l'exécution d'une instruction utilisée dans votre application. C'est par exemple le cas si vous essayez d'ouvrir un fichier qui n'existe pas sur le disque de votre machine. Vous obtiendrez sûrement un message de ce type.



```
E:\livre java\workspace\progObjet\bin>java LectureFichier
Exception in thread "main" java.io.FileNotFoundException: c:\data\bilan.txt <Le
chemin d'accès spécifié est introuvable>
        at java.io.FileInputStream.open(Native Method)
        at java.io.FileInputStream.<init>()
        at java.io.FileInputStream.<init>()
        at LectureFichier.main(LectureFichier.java:12)

E:\livre java\workspace\progObjet\bin>
```

Ce type de message n'est pas très sympathique pour l'utilisateur !

Heureusement Java permet la récupération de ce type d'erreur et évite ainsi l'affichage de cet inquiétant message. Nous détaillerons cela un peu plus loin dans ce chapitre.

3. Les erreurs de logique

Les pires ennemis des développeurs. Tout se compile sans problème, tout s'exécute sans erreurs et pourtant "ça ne marche pas comme prévu" !!!

Il faut dans ce cas revoir la logique de fonctionnement de l'application. Les outils de débogage nous permettent de suivre le déroulement de l'application, de placer des points d'arrêt, de visualiser le contenu des variables, etc.

Ces outils ne remplacent cependant pas une bonne dose de réflexion (et parfois quelques comprimés d'aspirine).

a. Les exceptions

Lorsqu'une erreur se produit au cours de l'exécution d'une méthode, un objet `Exception` est créé pour représenter l'erreur qui vient de ce produire. Cet objet contient de nombreuses informations concernant l'erreur qui est survenue dans l'application ainsi que l'état de l'application au moment de l'apparition de l'erreur. Cet objet est ensuite transmis à la machine virtuelle. C'est le déclenchement de l'exception. La machine virtuelle doit alors rechercher une solution pour la gérer. Pour cela, elle explore les différentes méthodes ayant été appelées pour atteindre l'emplacement où l'erreur s'est produite. Dans ces différentes méthodes, elle recherche un gestionnaire d'exceptions capable de traiter le problème. La recherche débute par la méthode où l'exception a été déclenchée puis remonte jusqu'à la méthode `main` de l'application si besoin. Lorsqu'un gestionnaire d'exception adapté est trouvé, l'objet `Exception` lui est transmis pour qu'il en assure le traitement. Si la recherche est infructueuse, l'application s'arrête.

Les exceptions sont en général classées en trois catégories.

- Les exceptions vérifiées correspondent à une situation anormale au cours du fonctionnement de l'application. Cette situation est en général liée à un élément extérieur à l'application, comme par exemple une connexion vers une base de données ou une lecture de fichier. Ces exceptions sont représentées par des instances de la classe `Exception` ou de l'une de ses sous-classes. Elles doivent obligatoirement être traitées par un bloc `try catch` ou propagées au code appelant avec le mot clé `throws` placé dans la signature de la fonction.

- Les erreurs correspondent à des conditions exceptionnelles extérieures à l'application que celle-ci ne peut généralement pas prévoir. Ces exceptions sont représentées par une instance de la classe `Error` ou de l'une de ses sous-classes. Ces exceptions ne sont pas obligatoirement traitées. Il faut d'ailleurs préciser que lorsqu'elles surviennent, le fonctionnement de l'application est généralement très fortement compromis. Si elles sont gérées, la plupart du temps le seul traitement consiste à afficher un message un peu moins effrayant pour l'utilisateur que celui proposé par défaut par la machine virtuelle Java. Dans pratiquement tous les cas l'arrêt de l'application est inéluctable.
- Les erreurs liées à une mauvaise utilisation d'une fonctionnalité du langage, ou à une erreur de logique dans la conception de l'application. L'erreur la plus fréquente que vous rencontrerez dans vos débuts avec Java sera certainement l'exception `NullPointerException` déclenchée lors de l'utilisation d'une variable non initialisée. Ces exceptions sont représentées par une instance de la classe `RuntimeException`. Bien que ces exceptions puissent être traitées par des blocs `try catch`, il est fortement déconseillé de le faire. Il est préférable d'analyser le code et de le modifier pour éviter qu'elles n'apparaissent.

b. Récupération d'exceptions

La gestion des exceptions donne la possibilité de protéger un bloc de code contre les exceptions qui pourraient s'y produire. Le code "dangereux" doit être placé dans un bloc `try`. Si une exception est déclenchée dans ce bloc de code, le ou les blocs de code `catch` sont examinés. S'il en existe un capable de traiter l'exception, le code correspondant est exécuté, sinon la même exception est déclenchée pour éventuellement être récupérée par un bloc `try` de plus haut niveau. Une instruction `finally` permet de marquer un groupe d'instructions qui seront exécutées à la sortie du bloc `try` si aucune exception ne s'est produite ou à la sortie d'un bloc `catch` si une exception a été déclenchée. La syntaxe générale est donc la suivante :

```
try
{
...
Instructions dangereuses
...
}
catch (exception1 e1)
{
...
code exécuté si une exception de type Exception1 se produit
...
}
catch (exception2 e2)
{
...
code exécuté si une exception de type Exception1 se produit
...
}
finally
{
...
code exécuté dans tous les cas avant la sortie du bloc try ou
d'un bloc catch
...
}
```

Cette structure a un fonctionnement très semblable au `switch case` déjà étudié.

Vous devez indiquer pour chaque bloc `catch` le type d'exception qu'il doit gérer.

```
public void lireFichier(String nom)
{
    FileInputStream fichier=null;
    BufferedReader br=null;
    String ligne=null;
    try
```

```

{
    fichier=new FileInputStream("c:\\data\\bilan.txt");
}
catch (FileNotFoundException e)
{
    e.printStackTrace();
}
br=new BufferedReader(new InputStreamReader(fichier));
try
{
    ligne=br.readLine();
}
catch (IOException e)
{
    e.printStackTrace();
}
while (ligne!=null)
{
    System.out.println(ligne);
    try
    {
        ligne=br.readLine();
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
}
}

```

Dans l'exemple précédent, chaque instruction susceptible de déclencher une exception est protégée par son propre bloc `try`. Cette solution présente l'avantage d'être extrêmement précise pour la gestion des exceptions au détriment de la lisibilité du code. Une solution plus simple consiste à regrouper plusieurs instructions dans un même bloc `try`. Notre exemple peut également être codé de la façon suivante :

```

public void lireFichier(String nom)
{
    FileInputStream fichier=null;
    BufferedReader br=null;
    String ligne=null;
    try
    {
        fichier=new FileInputStream(nom);
        br=new BufferedReader(new InputStreamReader(fichier));
        ligne=br.readLine();
        while (ligne!=null)
        {
            System.out.println(ligne);
            ligne=br.readLine();
        }
    }
    catch (FileNotFoundException e)
    {
        e.printStackTrace();
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
}

```

Le code est beaucoup plus lisible, par contre nous perdons en précision car il devient difficile de déterminer quelle instruction a déclenché l'exception.

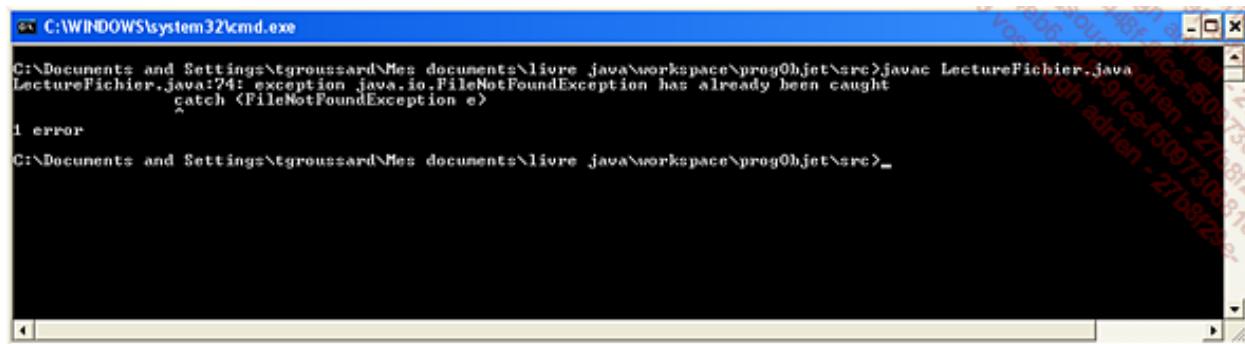
Il faut également être vigilant sur l'ordre des blocs `catch` et toujours les organiser du plus précis au plus général. Les exceptions étant des classes, elles peuvent avoir des relations d'héritage. Si un bloc `catch` est

prévu pour gérer un type particulier d'exception, il est également capable de gérer toutes les types d'exceptions qui héritent de celle-ci. C'est le cas dans notre exemple puisque la classe `FileNotFoundException` hérite de la classe `IOException`. Le compilateur détecte une telle situation et génère une erreur. Si nous modifions notre code de la façon suivante :

```
public void lireFichier(String nom)
{
    FileInputStream fichier=null;
    BufferedReader br=null;
    String ligne=null;
    try
    {
        fichier=new FileInputStream(nom);
        br=new BufferedReader(new InputStreamReader(fichier));
        ligne=br.readLine();
        while (ligne!=null)
        {
            System.out.println(ligne);
            ligne=br.readLine();
        }
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
    catch (FileNotFoundException e)
    {
        e.printStackTrace();
    }

}
```

nous obtenons cette erreur au moment de la compilation.



The screenshot shows a command-line interface window titled 'C:\WINDOWS\system32\cmd.exe'. The command entered is 'javac LectureFichier.java'. The output shows an error message: 'LectureFichier.java:74: exception java.io.FileNotFoundException has already been caught'. A red diagonal watermark with the text 'Tutoriel Java - Adrien - 27/08/2013' is visible across the window.

Le code peut être encore plus concis en indiquant qu'un même bloc `catch` doit gérer plusieurs types d'exceptions. Les différents types d'exceptions qu'un bloc `catch` peut traiter doivent être indiqués dans la déclaration en les séparant par le caractère |.

```
public static void lireFichier(String nom)
{
    FileInputStream fichier=null;
    BufferedReader br=null;
    String ligne=null;
    double somme=0;

    try
    {
        fichier=new FileInputStream(nom);
        br=new BufferedReader(new InputStreamReader(fichier));
        ligne=br.readLine();
        while (ligne!=null)
        {
```

```

        System.out.println(ligne);
        ligne=br.readLine();
        somme=somme+Double.parseDouble(ligne);
    }
    System.out.println("total : " + somme);
}

catch (IOException | NumberFormatException e)
{
    e.printStackTrace();
}
}

```

c. Exceptions associées à des ressources

De nombreuses applications ont fréquemment besoin d'accéder à des ressources externes. Les fichiers et les bases de données sont certainement les exemples les plus courants. L'utilisation de ces ressources débute par une opération d'ouverture, l'exploitation de la ressource, puis la fermeture de la ressource. Fréquemment, les méthodes permettant d'exploiter ces ressources sont susceptibles de déclencher de nombreuses exceptions et de ce fait sont placées dans une structure try-catch. Celle-ci peut aussi se voir confier la fermeture de la ressource à la fin de l'exécution des instructions qu'elle contient. La ou les ressources doivent être déclarées et instanciées entre parenthèses après le mot clé `try`. Si le bloc `try` contient plusieurs déclarations, celles-ci doivent être séparées par un point-virgule.

À la fin de l'exécution du bloc `try` la méthode `close` est appelée sur toutes les ressources déclarées au niveau du mot clé `try`. Cet appel est toujours effectué avant l'exécution d'un bloc `catch` ou du bloc `finally`. Pour que ce mécanisme fonctionne, les classes correspondant aux ressources utilisées doivent implémenter l'interface `Closeable` ou `AutoCloseable`. Ces deux interfaces exigent l'existence de la méthode `close` dans la classe de la ressource utilisée dans le bloc `try`.

Dans l'exemple ci-dessous, l'objet `BufferedReader` est automatiquement fermé après l'exécution du bloc `try`.

```

String reponse="";
try (BufferedReader br=new BufferedReader(new
InputStreamReader(System.in)))
{
    while (!reponse.equals("fin"))
    {
        ...
        ...
        reponse=br.readLine();
    }
}
catch(IOException e )
{
    e.printStackTrace();
}

```

Le code de chaque bloc `catch` peut obtenir plus d'informations sur l'exception qu'il doit traiter en utilisant les méthodes disponibles dans la classe correspondant à l'exception. Les méthodes suivantes sont les plus utiles pour obtenir des informations complémentaires sur l'exception.

- `getMessage` : permet d'obtenir le message d'erreur associé à l'exception.
- `getCause` : permet d'obtenir l'exception initiale dans le cas où le chaînage d'exception est utilisé.
- `getStackTrace` : permet d'obtenir un tableau de `StackTraceElement` dont chaque élément représente une méthode appelée jusqu'à celle où est traitée l'exception. Pour chacune d'elle nous pouvons obtenir les informations suivantes :
 - Le nom de la classe où se trouve la méthode : `getClassName`.
 - Le nom du fichier où se trouve cette classe : `getFilename`.
 - Le numéro de la ligne où l'exception a été déclenchée : `getLineNumber`.

- Le nom de la méthode : `getMethodName`.

Ces informations peuvent être utilisées pour générer des fichiers historiques du fonctionnement de l'application. Voici un exemple d'enregistrement de ces informations dans un fichier texte.

```

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.GregorianCalendar;

public class LectureFichier
{
    public static void main(String args[])
    {
        try {
            lireFichier("bilan.txt");
        }
        catch (CaMarchePasException e)
        {
            FileWriter log;
            BufferedWriter br;
            try
            {
                log=new FileWriter("historique.txt",true);
                br=new BufferedWriter(log);
                br.write("----->"+ new GregorianCalendar().
getTime()+" <-----\r\n");
                br.write("erreur : " + e.getMessage()+"\r\n");
                for (int i=0;i<e.getStackTrace().length;i++)
                {
                    br.write("dans le fichier " +
e.getStackTrace()[i].getFileName());
                    br.write(" à la ligne " +
e.getStackTrace()[i].getLineNumber());
                    br.write(" dans la méthode " +
e.getStackTrace()[i].getMethodName());
                    br.write(" de la classe " +
e.getStackTrace()[i].getClassName()+"\r\n");
                }
                br.close();
                log.close();
            }
            catch (IOException ex)
            {
                System.out.println("erreur dans l'application");
            }
        }
    }

    public static void lireFichier(String nom) throws
CaMarchePasException
    {
        FileInputStream fichier=null;
        BufferedReader br=null;
        String ligne=null;
        try
        {
            fichier=new FileInputStream(nom);
            br=new BufferedReader(new InputStreamReader(fichier));
            ligne=br.readLine();
            while (ligne!=null)
            {
                System.out.println(ligne);
            }
        }
        catch (IOException ex)
        {
            System.out.println("erreur dans l'application");
        }
        finally
        {
            if (fichier!=null)
                try
                {
                    fichier.close();
                }
                catch (IOException ex)
                {
                }
        }
    }
}

```

```

        ligne=br.readLine();
    }
}
catch (FileNotFoundException e)
{
    throw new CaMarchePasException("le fichier n'existe
pas",e);
}
catch (IOException e)
{
    throw new CaMarchePasException("erreur de lecture du
fichier",e);
}
}
}
}

```

d. Création et déclenchement d'exceptions

Les exceptions sont avant tout des classes, il est donc possible de créer nos propres exceptions en héritant d'une des nombreuses classes d'exception déjà disponibles. Pour respecter les conventions, il est conseillé de terminer le nom de la classe par le terme Exception. Nous pouvons par exemple écrire le code suivant :

```

public class CaMarchePasException extends Exception
{
    public CaMarchePasException()
    {
        super();
    }
    public CaMarchePasException(String message)
    {
        super(message);
    }
    public CaMarchePasException(String message, Throwable cause)
    {
        super(message,cause);
    }
    public CaMarchePasException(Throwable cause)
    {
        super(cause);
    }
}

```

- La surcharge des constructeurs de la classe de base est fortement conseillée pour conserver la cohérence entre les classes d'exception.

Cette classe peut ensuite être utilisée pour le déclenchement d'une exception personnalisée. Pour déclencher une exception, il faut au préalable créer une instance de la classe correspondante puis déclencher l'exception avec le mot clé `throw`. Le déclenchement d'une exception dans une fonction avec le mot clé `throw` provoque la sortie immédiate de la fonction. Le code suivant déclenche une exception personnalisée dans les blocs `catch`.

```

public static void lireFichier2(String nom) throws CaMarchePasException
{
    FileInputStream fichier=null;
    BufferedReader br=null;
    String ligne=null;
    try
    {
        fichier=new FileInputStream(nom);
        br=new BufferedReader(new InputStreamReader(fichier));
        ligne=br.readLine();
        while (ligne!=null)
        {
            System.out.println(ligne);
            ligne=br.readLine();
        }
    }
    catch (CaMarchePasException e)
    {
        throw e;
    }
}

```

```
        }
    } catch (FileNotFoundException e)
    {
        throw new CaMarchePasException("le fichier n'existe pas",e);
    } catch (IOException e)
    {
        throw new CaMarchePasException("erreur de lecture
du fichier",e);
    }
}
```

Lorsqu'une fonction est susceptible de déclencher une exception, vous devez le signaler dans la signature de cette fonction avec le mot clé `throws` suivi de la liste des exceptions qu'elle peut déclencher. Lorsque cette fonction sera ensuite utilisée dans une autre fonction, vous devrez obligatoirement tenir compte de cette, ou de ces éventuelles exceptions. Vous devrez donc soit gérer l'exception avec un bloc `try ... catch` soit la propager en ajoutant le mot clé `throws` à la déclaration de la fonction. Il faut cependant être prudent et ne pas propager les exceptions au-delà de la méthode `main` car dans ce cas, c'est la machine virtuelle Java qui les récupère et arrête brutalement l'application.

Les collections

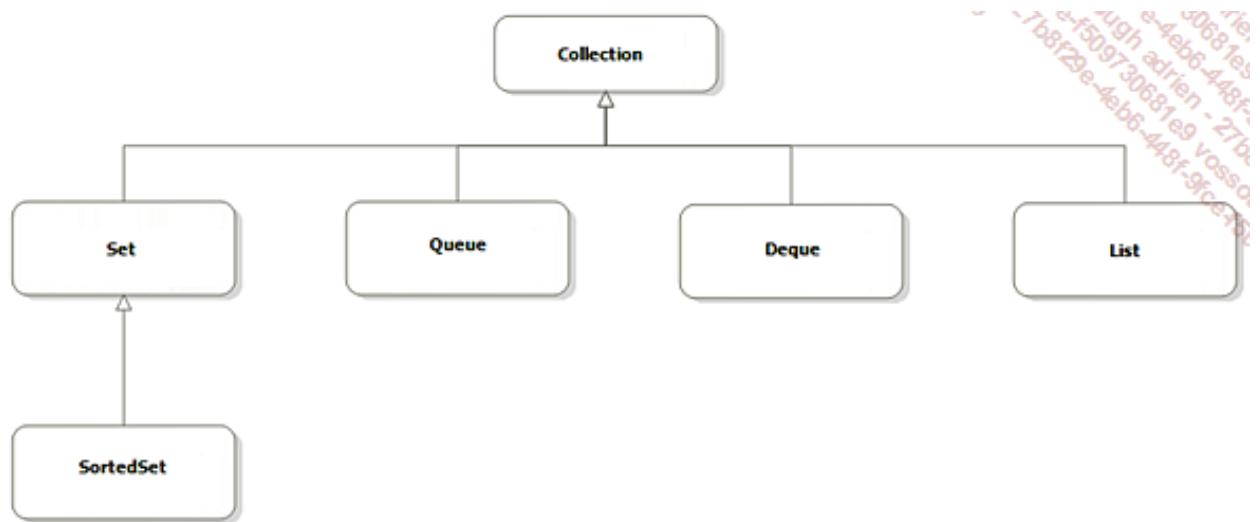
Les applications ont très fréquemment besoin de manipuler de grandes quantités d'informations. De nombreuses structures sont disponibles pour faciliter la gestion de ces informations. Elles sont regroupées sous le terme collection. Comme dans la vie courante, il y a différents types de collections et de collectionneurs. Il peut y avoir des personnes qui récupèrent toutes sortes de choses mais qui n'ont pas d'organisation particulière pour les ranger, d'autres qui sont spécialisées dans la collection de certains types d'objets, les maniaques qui prennent toutes les précautions possibles pour pouvoir retrouver à coup sûr un objet...

Le langage Java propose différentes classes permettant de mettre en place plusieurs modes de gestion.

La première solution pour gérer un ensemble d'éléments est l'utilisation de tableaux. Cette solution a été décrite dans la section Les tableaux du chapitre Bases du langage. Bien que simple à mettre en œuvre, cette solution n'est pas très souple. Son principal défaut tient au caractère fixe de la taille d'un tableau. S'il n'y a plus de place pour stocker des éléments supplémentaires, il faut créer un nouveau tableau plus grand et y transférer le contenu du tableau précédent. Cette solution, lourde à mettre en œuvre, est consommatrice de ressources.

Le langage Java propose une vaste palette d'interfaces et de classes pour gérer facilement des ensembles d'éléments. Les interfaces décrivent les fonctionnalités disponibles alors que les classes implémentent et fournissent réellement ces fonctionnalités. Suivant le mode de gestion des éléments souhaité, on utilisera bien sûr la classe la plus adaptée. Cependant, les mêmes fonctionnalités de base doivent être accessibles quel que soit le mode de gestion. Pour assurer la présence de ces fonctionnalités indispensables dans toutes les classes, celles-ci ont été regroupées dans l'interface `Collection` qui est par la suite utilisée comme interface de base.

La hiérarchie des interfaces est représentée sur le diagramme ci-dessous.



Toutes ces interfaces sont génériques afin de gérer des ensembles composés de n'importe quels types d'éléments tout en évitant les fastidieuses opérations de transtypage.

Pour chacune de ces interfaces, une ou plusieurs classes correspondantes sont disponibles. Ce sont bien sûr celles-ci qui vont nous intéresser.

1. La classe `ArrayList`

Cette classe est une implémentation de l'interface `List`. Elle permet la gestion des éléments que l'on y place de manière quasi similaire à celle disponible avec un tableau, avec en plus l'aspect dynamique. Après la création de l'instance de la classe `ArrayList`, les éléments peuvent y être ajoutés grâce aux méthodes `add` et `addAll`. Par défaut, ces deux méthodes ajoutent les éléments à la suite de ceux déjà présents. Une version surchargée de ces deux méthodes permet de choisir l'emplacement où les éléments sont placés. Pour cette version, il faut spécifier la position où doit avoir lieu l'ajout.

La première position est à l'index 0. Si des éléments sont déjà présents, ils sont simplement décalés.

L'accès aux éléments de la liste se fait via la méthode `get` à laquelle on indique simplement l'index de l'élément. À l'inverse, la méthode `set` permet de placer un élément à la position indiquée. Il n'y a pas dans ce cas d'insertion, mais remplacement de l'élément se trouvant déjà à cette position.

La suppression d'un élément est effectuée par la méthode `remove` à laquelle on indique soit l'index, soit l'élément que l'on souhaite supprimer. Dans ces deux cas, les éléments suivants sont décalés d'un cran (il n'y a jamais de "trou" dans une `ArrayList`). Les trois fonctions `indexOf`, `lastIndexOf` et `contains` effectuent la recherche d'un élément dans l'`ArrayList`. Les deux premières retournent l'index où l'élément a été trouvé ou -1 si aucun élément n'est trouvé. La fonction `indexOf` commence la recherche par le premier élément de la liste, la fonction `lastIndexOf` débute la recherche par la fin de la liste. La fonction `contains` indique simplement si l'élément est présent dans la liste, sans indication sur sa position s'il existe.

Le parcours des éléments de la liste est possible avec une boucle `for` ou en utilisant l'objet `Iterator` fourni par la méthode `iterator` de la classe `ArrayList`. Avec ces deux solutions, le parcours se fait uniquement du premier vers le dernier élément de la liste, de plus le contenu de la liste ne doit pas être modifié pendant le parcours. Un objet `ListIterator`, retourné par la méthode `listIterator`, permet plus de souplesse puisqu'il autorise les déplacements dans la liste dans les deux sens et accepte également la modification de la liste pendant le parcours.

L'exemple de code ci-dessous illustre ces différentes fonctionnalités.

```
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.ListIterator;
public class testArrayList
{
    public static void main(String[] args)
    {
        ArrayList<Personne> liste1;
        ArrayList<Personne> liste2;
        // création des deux instances
        liste1=new ArrayList<Personne>();
        liste2=new ArrayList<Personne>();

        // création des personnes pour remplir la liste
        Personne p1,p2,p3,p4,p5;
        p1 = new Personne("Wayne", "John", LocalDate.of(1907,5,26));
        p2 = new Personne("McQueen", "Steeve",LocalDate.of(1930,3,24));
        p3 = new Personne("Lennon", "John",LocalDate.of(1940,10,9));
        p4 = new Personne("Gibson", "Mel",LocalDate.of(1956,1,3));
        p5 = new Personne("Willis", "Bruce",LocalDate.of(1955,3,19));

        // ajout de quatre personnes à la liste
        liste1.add(p1);
        liste1.add(p3);
        liste1.add(p4);
        liste1.add(p5);

        // insertion d'une personne entre p1 et p3
        // donc à la position 1 de la liste
        liste1.add(1, p2);

        // ajout du contenu d'une liste à une autre liste
        // les deux listes contiennent maintenant les mêmes objets.
        // !!!! ne pas confondre avec liste2=liste1; !!!
        liste2.addAll(liste1);

        // affichage du nombre d'éléments de la liste
        System.out.println("il y a " + liste1.size() + " personne(s) dans la liste");

        // parcours de la première liste du début vers la fin
```

```

Iterator<Personne> it;
it=liste1.iterator();
Personne p;
// tant qu'il reste des éléments

while (it.hasNext())
{
    // récupération de l'élément courant
    p=it.next();
    System.out.println(p.getNom());
}

// parcours de la première liste de la fin vers le début
// récupération d'un ListIterator positionné après
// le dernier élément (le nombre d'éléments de la liste)

ListIterator<Personne> lit;
lit=liste1.listIterator(liste1.size());
// tant qu'il reste des éléments
while (lit.hasPrevious())
{
    // récupération de l'élément courant
    // en remontant dans la liste
    p=lit.previous();
    System.out.println(p.getNom());
}

// remplacement d'un élément de liste
liste1.set(2,new Personne("Grant", "Cary",LocalDate.of(1904,1,18)));

// affichage de l'élément à la troisième position de la liste
System.out.println(liste1.get(2).getNom());

// recherche d'un élément dans la liste
int position;
position=liste1.indexOf(p4);
if(position==-1)
    System.out.println("non trouve dans la liste");
else
    System.out.println(liste1.get(position).getNom());

// recherche d'un élément inexistant dans la liste.
// John Lennon a été remplacé par Cary Grant
// La recherche débute à la fin de la liste

position=liste1.lastIndexOf(p3);
if(position==-1)
    System.out.println("non trouve dans la liste");
else
    System.out.println(liste1.get(position).getNom());

// suppression sélective de la liste
// l'expression lambda détermine quels éléments seront supprimés
liste1.removeIf((Personne pe) ->pe.getDate_nais().getYear()<1940);
// parcours de la liste pour voir le résultat
it=liste1.iterator();
// tant qu'il reste des éléments
while (it.hasNext())
{
    // récupération de l'élément courant
    p=it.next();
    System.out.println(p.getNom()+" ne en "+
p.getDate_nais().getYear() );
}
// une autre manière de parcourir la liste
// l'expression lambda est exécutée pour chaque
// élément de la liste

```

```

        liste1.forEach((Personne per)->System.out.println(per.getNom()+" "
age " + per.calculAge()));

    }
}

```

2. La classe HashSet

Cette classe est une implémentation de l'interface `Set`. Son fonctionnement est pratiquement similaire à celui de la classe `ArrayList`. La principale différence est liée à la gestion des doublons.

Contrairement à une `ArrayList`, un `HashSet` n'accepte pas le stockage de deux éléments identiques. Lorsqu'un élément est ajouté à un `HashSet` avec la méthode `add`, celle-ci vérifie s'il n'y a pas déjà un élément identique en comparant le `hashCode` de l'élément avec ceux des éléments déjà présents dans la liste. Si l'ajout est effectué, la méthode `add` retourne un booléen `true` et bien sûr `false` dans le cas inverse. Ceci nous impose donc de redéfinir la méthode `hashCode` de toutes les classes pour lesquelles nous envisageons de stocker des instances dans un `HashSet`.

Cette méthode doit respecter une règle bien précise : si l'on considère que deux objets sont identiques, la méthode `hashCode` doit renvoyer la même valeur pour chacun d'eux.

Pour conserver une cohérence, il est également indispensable de redéfinir la méthode `equals` de la classe avec les mêmes critères d'égalité.

Pour pouvoir stocker dans un `HashSet` des instances de la classe `Personne`, nous devons donc y ajouter ces deux méthodes.

```

import java.time.LocalDate;
import java.time.temporal.ChronoUnit;
public class Personne           implements Classable

{
    private String nom;
    private String prenom;
    private LocalDate date_nais=LocalDate.of(1963,11,29);

    public Personne()
    {
    }

    public Personne(String n,String p,LocalDate d)
    {
        this.nom=n;
        this.prenom=p;
        this.date_nais=d;
    }

    public String getNom()
    {
        return nom;
    }

    public void setNom(String nom)
    {
        this.nom = nom;
    }

    public String getPrenom()
    {
        return prenom;
    }

    public void setPrenom(String prenom)
    {
        this.prenom = prenom;
    }
}

```

```

public LocalDate getDate_nais()
{
    return date_nais;
}

public void setDate_nais(LocalDate date_nais)
{
    this.date_nais = date_nais;
}

@Override
public int hashCode()
{
    // On choisit les deux nombres impairs
    int resultat = 7;
    final int multiplier = 17;

    // Pour chaque attribut, on calcule le hashcode
    // que l'on ajoute au résultat après l'avoir multiplié
    // par le nombre "multiplieur" :
    resultat = multiplier*resultat + (nom==null ? 0 :
nom.hashCode());
    resultat = multiplier*resultat + (prenom==null ? 0 :
prenom.hashCode());
    resultat = multiplier*resultat + (date_nais==null ? 0 :
date_nais.hashCode());

    return resultat;
}

@Override
public boolean equals(Object obj)
{
    // si le deuxième objet est null il ne
    // peut pas y avoir égalité
    if (obj == null)
    {
        return false;
    }
    // si les deux objets ne sont pas de même type
    // il ne peut pas y avoir égalité
    if (getClass() != obj.getClass())
    {
        return false;
    }
    // il faut maintenant vérifier l'égalité de chacun
    // des attributs
    Personne p = (Personne) obj;
    if (!nom.equals(p.getNom()))
        return false;
    if (!prenom.equals(p.getPrenom()))
        return false;
    if (!date_nais.equals(getDate_nais()))
        return false;
    return true;
}

public long calculAge()
{
    return date_nais.until(LocalDate.now(),ChronoUnit.YEARS);
}

public int compare(Object o)
{
    Personne p;
    if (o instanceof Personne)
    {
        p=(Personne)o;
    }
}

```

```

        else
        {
            return Classable.ERREUR;
        }
        if (getNom().compareTo(p.getNom())<0)
        {
            return Classable.INFERIEUR;
        }
        if (getNom().compareTo(p.getNom())>0)
        {
            return Classable.SUPERIEUR;
        }

        return Classable.EGAL;
    }
}

```

Vérifions maintenant nos modifications de la classe Personne avec le code suivant :

```

public static void main(String[] args)
{
    Personne p1,p2,p3;
    p1 = new Personne("Wayne", "John",LocalDate.of(1907,5,26));
    p2 = new Personne("McQueen", "Steeve",LocalDate.of(1930,3,24));
    p3 = new Personne("Wayne", "John",LocalDate.of(1907,5,26));

    System.out.println("hashCode de p1 : " + p1.hashCode());
    System.out.println("hashCode de p2 : " + p2.hashCode());
    System.out.println("hashCode de p3 : " + p3.hashCode());

    if(p1.equals(p2))
        System.out.println("p1 et p2 sont identiques");
    else
        System.out.println("p1 et p2 sont différents");

    if(p1.equals(p3))
        System.out.println("p1 et p3 sont identiques");
    else
        System.out.println("p1 et p3 sont différents");
}

```

Nous obtenons le résultat suivant :

```

hashCode de p1 : -1636676846
hashCode de p2 : 633943827
hashCode de p3 : -1636676846
p1 et p2 sont différents
p1 et p3 sont identiques

```

ce qui valide nos modifications.

Nous pouvons maintenant tester les fonctionnalités de la classe HashSet. Elles sont pratiquement identiques à celles de la classe ArrayList à un petit détail près. En effet, dans un HashSet il n'est pas possible d'accéder à un élément particulier puisqu'il n'y a pas de notion d'index. La seule solution est d'utiliser l'objet iterator associé au HashSet.

```

import java.time.LocalDate;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.Iterator;
import java.util.ListIterator;
public class TestHashSet
{
    public static void main(String[] args)
    {
        HashSet<Personne> hash1;

```

```

HashSet<Personne> hash2;
// création des deux instances
hash1=new HashSet<Personne>();
hash2=new HashSet<Personne>();

// création des personnes pour remplir le HashSet
Personne p1,p2,p3,p4,p5;
p1 = new Personne("Wayne", "John",LocalDate.of(1907,5,26));
p2 = new Personne("McQueen", "Steeve",LocalDate.of(1930,3,24));
p3 = new Personne("Lennon", "John",LocalDate.of(1940,10,9));
p4 = new Personne("Gibson", "Mel",LocalDate.of(1956,1,3));
p5 = new Personne("Willis", "Bruce",LocalDate.of(1955,3,19));

// ajout de quatre personnes au HashSet
hash1.add(p1);
hash1.add(p3);
hash1.add(p4);
hash1.add(p5);

// ajout du contenu d'un HashSet à un autre HashSet
// les deux HashSet contiennent maintenant les mêmes
// objets.
// !!!! ne pas confondre avec hash2=hash1; !!!
hash2.addAll(hash1);

// affichage du nombre d'éléments du HashSet
System.out.println("il y a " + hash1.size() + " personne(s)
dans le HashSet");

// parcours du premier HashSet du début vers la fin
Iterator<Personne> it;
it=hash1.iterator();
// tant qu'il reste des éléments dans le HashSet
Personne p;
while (it.hasNext())
{
    // récupération de l'élément courant
    p=it.next();
    System.out.println(p.getNom());
}
// suppression sélective dans le HashSet
// l'expression détermine quels éléments seront supprimés
hash1.removeIf((Personne pe)->pe.getDate_nais().getYear()<1940);
// parcours du HashSet pour voir le résultat
it=hash1.iterator();
// tant qu'il reste des éléments dans le HashSet
while (it.hasNext())
{
    // récupération de l'élément courant
    p=it.next();
    System.out.println(p.getNom()+" ne en "+p.getDate_nais().getYear());
}
// une autre manière de parcourir le HashSet
// l'expression lambda est exécutée pour chaque
// élément du HashSet

hash1.forEach((Personne per)->System.out.println(
per.getNom()+" age "+per.calculAge()));

}
}

```

Malgré cette petite restriction, les HashSet sont particulièrement efficaces pour réaliser des opérations ensemblistes, comme les unions et les intersections. Avant de tester ces opérations avec des HashSet, regardons leurs définitions théoriques.

Supposons que nous avons à notre disposition deux ensembles contenant des Personnes. L'un contient des chanteurs, l'autre contient des acteurs.

- Notion de sous-ensemble : on considère qu'un ensemble est un sous-ensemble d'un autre si tous ses éléments sont contenus dans celui-ci. Par exemple, on peut dire que chanteurs et un sous-ensemble d'acteurs si tous les chanteurs sont également acteurs.
- Notion d'union : l'union de deux ensembles est l'ensemble constitué par tous les éléments contenus dans chacun des deux ensembles. On peut par exemple définir un ensemble nommé artistes qui est l'union des acteurs et des chanteurs.
- Notion d'intersection : l'intersection de deux ensembles est constituée par les éléments contenus à la fois dans le premier et le deuxième ensemble. Dans notre cas, il contient les personnes étant à la fois acteurs et chanteurs.
- Notion de différence : la différence entre deux ensembles est constituée par tous les éléments présents dans l'un mais pas dans l'autre. Dans notre cas, ils représentent les personnes qui sont uniquement chanteurs et celles qui sont uniquement acteurs.

Quatre fonctions de la classe HashSet permettent de traduire très facilement ces notions. Il s'agit de la fonction `containsAll` pour la notion de sous-ensemble, de la fonction `addAll` pour l'union, de la fonction `retainAll` pour l'intersection et de la fonction `removeAll` pour la différence.

Ces fonctions modifient le contenu du HashSet sur lequel elles sont appelées, il est donc prudent d'en faire une copie et de travailler sur cette copie.

L'exemple de code suivant illustre ces différentes notions.

```

import java.time.LocalDate;
import java.util.HashSet;
import java.util.Iterator;

public class TestHashSet
{
    public static void main(String[] args)
    {
        HashSet<Personne> acteurs;
        HashSet<Personne> chanteurs;
        acteurs=new HashSet<Personne>();
        chanteurs=new HashSet<Personne>();

        // création des personnes pour remplir la liste
        Personne p1,p2,p3,p4,p5;
        p1 = new Personne("Wayne", "John",LocalDate.of(1907,5,26));
        p2 = new Personne("McQueen", "Steeve",LocalDate.of(1930,3,24));
        p3 = new Personne("Lennon", "John",LocalDate.of(1940,10,9));
        p4 = new Personne("Gibson", "Mel",LocalDate.of(1956,1,3));
        p5 = new Personne("Willis", "Bruce",LocalDate.of(1955,3,19));

        acteurs.add(p1);
        acteurs.add(p2);
        acteurs.add(p4);
        acteurs.add(p5);

        chanteurs.add(p1);
        chanteurs.add(p3);

        // test si les chanteurs sont également acteurs
        if (acteurs.containsAll(chanteurs))
            System.out.println("tous les chanteurs sont aussi acteurs");
        else
            System.out.println("certains chanteurs ne sont pas aussi acteurs");
        System.out.println("***** les artistes *****");
        // création d'un HashSet artistes contenant chanteurs
        // et acteurs
        HashSet<Personne> artistes;
        artistes=new HashSet<Personne>(chanteurs);
        artistes.addAll(acteurs);
        // parcours du premier HashSet des artistes
    }
}

```

```

Iterator<Personne> it;
it=artistes.iterator();
// tant qu'il reste des éléments dans le HashSet
Personne p;
while (it.hasNext())
{
    // récupération de l'élément courant
    p=it.next();
    System.out.println(p.getNom());
}
System.out.println("***** chanteurs et acteurs *****");
// création d'un HashSet des personnes qui sont
// chanteurs et acteurs

HashSet<Personne> act_chant;
act_chant=new HashSet<Personne>(chanteurs);
act_chant.addAll(acteurs);
it=act_chant.iterator();
// tant qu'il reste des éléments dans le HashSet
while (it.hasNext())
{
    // récupération de l'élément courant
    p=it.next();
    System.out.println(p.getNom());
}
System.out.println("***** chanteurs uniquement *****");
// création d'un HashSet des personnes
// uniquement acteurs
HashSet<Personne> uniquementChanteurs;
uniquementChanteurs=new HashSet<Personne>(chanteurs);
uniquementChanteurs.removeAll(acteurs);
for(Personne pe:uniquementChanteurs)
{
    System.out.println(pe.getNom());
}

System.out.println("***** acteurs uniquement *****");
// création d'un HashSet des personnes
// uniquement acteurs
HashSet<Personne> uniquementActeurs;
uniquementActeurs=new HashSet<Personne>(acteurs);
uniquementActeurs.removeAll(chanteurs);
for(Personne pe:uniquementActeurs)
{
    System.out.println(pe.getNom());
}
}
}

```

3. La classe LinkedList

La classe `LinkedList` est très complète puisqu'elle implémente les interfaces `Iterable`, `Collection`, `Deque`, `List` et `Queue`. Grâce à l'implémentation des interfaces `Collection`, `List` et `Iterable`, elle possède un fonctionnement identique à la classe `ArrayList`. L'implémentation des interfaces `Deque` et `Queue` apporte quelques fonctionnalités supplémentaires que nous allons étudier.

Cette classe est recommandée si vous avez besoin d'accéder aux éléments dans le même ordre que celui dans lequel ils ont été stockés ou dans l'ordre inverse. Ces mécanismes sont appelés *First in - First out* (FIFO) ou *Last in - First out* (LIFO).

L'ajout d'un élément s'effectue avec les méthodes `addFirst` ou `addLast` pour un ajout soit en début de liste soit en fin de liste.

Pour obtenir un élément présent dans la liste, deux options sont possibles :

- obtenir l'élément en laissant celui-ci dans la liste, dans ce cas il faut utiliser les

méthodes `peekFirst` ou `peekLast`.

- obtenir l'élément et le retirer de la liste, dans ce cas il faut utiliser les méthodes `pollFirst` ou `pollLast`.

```
import java.time.LocalDate;
import java.util.LinkedList;

public class TestLinkedList
{
    public static void main(String[] args)
    {
        LinkedList<Personne> ll;
        ll=new LinkedList<Personne>();
        // création des personnes pour remplir le HashSet
        Personne p1,p2,p3,p4,p5;
        p1 = new Personne("Wayne", "John",LocalDate.of(1907,5,26));
        p2 = new Personne("McQueen", "Steeve",LocalDate.of(1930,3,24));
        p3 = new Personne("Lennon", "John",LocalDate.of(1940,10,9));
        p4 = new Personne("Gibson", "Mel",LocalDate.of(1956,1,3));
        p5 = new Personne("Willis", "Bruce",LocalDate.of(1955,3,19));
        // ajout des éléments dans la liste
        ll.addFirst(p1);
        ll.addFirst(p2);
        ll.addFirst(p3);
        ll.addFirst(p4);
        ll.addFirst(p5);
        Personne p=null;
        // extraction et suppression des éléments
        // de la liste en commençant par le plus ancien
        do
        {
            p=ll.pollLast();
            if (p!=null)
                System.out.println(p.getNom());
        } while(p!=null);

        ll.clear();
    }
}
```

4. streams et pipelines

Généralement les collections sont utilisées pour stocker des informations pour pouvoir les récupérer ou effectuer des traitements dessus. La technique classique pour réaliser des opérations sur les éléments contenus dans une collection consiste à faire une boucle `for` ou à utiliser l'`Iterator` associé à la collection. Cette solution peut être avantageusement remplacée par l'utilisation de `Stream` et `depipeline`. Pour illustrer le fonctionnement de ces deux éléments, on peut faire la correspondance avec le fonctionnement d'une usine. Notre usine reçoit la matière première, laquelle traverse différentes unités de transformation pour obtenir en sortie le produit fini. Dans notre cas, les données présentes dans la collection représentent la matière première. Celles-ci sont transportées par l'intermédiaire d'objets implémentant l'interface `Stream`. Les unités de transformation sont représentées par les `Pipelines`. Ceux-ci peuvent produire un résultat directement exploitable (le produit fini) ou un autre objet `Stream` qui va alimenter un nouveau `Pipeline`.

Toutes les classes implémentant l'interface `Collection` sont capables de fournir un objet de type `Stream`. C'est cet objet qui est responsable de l'itération sur les éléments présents dans la collection.

Les `Pipelines` sont un peu plus complexes puisqu'ils sont constitués de plusieurs éléments.

Le premier représente la source à partir de laquelle le `Pipeline` va obtenir les informations sur lesquelles il va effectuer le ou les traitements. C'est bien sûr un objet `Stream` qui sera la source d'un `Pipeline`. Les traitements sont représentés par les différentes méthodes définies dans l'`interface Stream`. Ces méthodes correspondent à différents types de traitement pouvant être réalisés sur les données. Par défaut, ces méthodes ne font rien. Leurs comportements doivent être adaptés en fonction du type des objets qu'elles auront à traiter. Une expression lambda ou une référence de méthode est généralement utilisée

pour cela. Ces méthodes peuvent générer le résultat final, un autre objet Stream constitué d'objets de même type que les objets d'origine ou un autre objet Stream contenant un autre type de données. On peut par exemple citer les méthodes suivantes représentatives de chacun de ces cas de figure.

boolean allMatch(Predicate<? super T> predicate) : cette fonction retourne un booléen indiquant si tous les éléments répondent à la condition indiquée par l'objet Predicate.

IntStream mapToInt(ToIntFunction<? super T> mapper) : cette fonction retourne un objet Stream d'entiers générés par le résultat de la fonction ToIntFunction.

Stream<T> filter(Predicate<? super T> predicate) : cette fonction génère un nouvel objet Stream contenant uniquement les objets du flux d'origine remplissant la condition définie par l'objet Predicate.

L'interface Stream contient plusieurs dizaines de méthodes répondant aux besoins les plus couramment rencontrés. Il est donc primordial de consulter la documentation la concernant.

L'exemple ci-dessous présente un petit aperçu des possibilités disponibles.

```
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.ListIterator;
public class TestStream
{
    public static void main(String[] args)
    {
        ArrayList<Personne> liste;

        // création des deux instances
        liste=new ArrayList<Personne>();

        // création des personnes pour remplir la liste
        Personne p1,p2,p3,p4,p5;
        p1 = new Personne("Wayne", "John",LocalDate.of(1907,5,26));
        p2 = new Personne("McQueen", "Steeve",LocalDate.of(1930,3,24));
        p3 = new Personne("Lennon", "John",LocalDate.of(1940,10,9));
        p4 = new Personne("Gibson", "Mel",LocalDate.of(1956,1,3));
        p5 = new Personne("Willis", "Bruce",LocalDate.of(1955,3,19));

        // ajout de cinq personnes à la liste
        liste.add(p1);
        liste.add(p2);
        liste.add(p3);
        liste.add(p4);
        liste.add(p5);

        // la fonction allMatch retourne true si tous les éléments
        // de la liste remplissent la condition exprimée
        // dans l'expression lambda

        if(liste.stream().allMatch(p->p.getDate_nais().getYear()>1945))
            System.out.println("toutes les personnes sont nées après 1945");
        else
            System.out.println("certaines personnes sont nées avant 1945");

        // filtrage de personnes nées au mois de mars
        // ce filtrage génère un nouveau Stream
        // forEach parcours ce nouveau Stream et
        // exécute l'expression lambda pour chaque élément

        liste.stream().filter(p->p.getDate_nais().getMonthValue()
==3).forEach(p->System.out.println(p.getNom()));

        // recherche de la personne la plus âgée de la liste
        // l'expression lambda représente l'implémentation
        // de l'interface Comparator
```

```
System.out.println(liste.stream().max((pe1,pe2)->
{
    if (pe1.calculAge()>pe2.calculAge())
        return 1;
    if (pe1.calculAge()<pe2.calculAge())
        return -1;
    return 0;
}).get().getNom());

// calcul de l'âge moyen des personnes présentes dans la
// liste mapToLong génère un nouveau Stream de type long
// obtenu à partir de la référence de fonction
// average calcule la moyenne de ce nouveau flux
// getAsDouble la tranforme en type double
double ageMoyen = liste
.stream()
.mapToLong(Personne::calculAge)
.average()
.getAsDouble();

System.out.println("age moyen des personnes de la liste :
" + ageMoyen + " ans");
}
}
```

Exercices

Exercice 1

Créer une classe représentant un article d'un magasin de vente par correspondance. Un article est caractérisé par sa référence, sa désignation, son prix. Créer ensuite une méthode main permettant de tester le bon fonctionnement de la classe précédente.

Exercice 2

Ajouter les deux classes Livre et Dvd héritant de la classe Article.

Un livre possède un numéro ISBN, contient un certain nombre de pages et à été écrit par un auteur, un Dvd a une certaine durée et a été produit par un réalisateur.

Ajouter les attributs nécessaires aux classes Livre et Dvd pour avoir le nom de l'auteur ou du réalisateur. Tester ensuite le fonctionnement de ces deux nouvelles classes.

Exercice 3

Modifier les classes Livre et Dvd pour avoir disponibles les informations suivantes concernant l'auteur ou le réalisateur :

- son nom
- son prénom
- sa date de naissance

Indice : les auteurs et les réalisateurs sont des Personnes.

Exercice 4

Modifier le code précédent pour pouvoir obtenir rapidement la liste des articles concernant un auteur ou un réalisateur.

Corrections

Correction de l'exercice 1

La classe Article est relativement simple. Les lignes 5 à 7 contiennent la déclaration des attributs représentant les informations que l'on souhaite mémoriser dans les instances de la classe. Ceux-ci sont déclarés avec la visibilité private pour garantir que seul le code de la classe pourra y accéder. Vient ensuite la déclaration des constructeurs (lignes 9 à 29). Les différentes versions permettent la création d'instances en fonction des informations disponibles. Pour éviter la duplication de code, les constructeurs peuvent s'appeler mutuellement (lignes 16, 21, 27).

Bien qu'il soit possible d'utiliser directement les attributs dans les constructeurs, il est préférable d'utiliser les méthodes setXXXX pour y accéder afin de profiter des éventuelles vérifications effectuées sur les valeurs à affecter aux attributs.

Pour rendre les attributs accessibles de l'extérieur de la classe, chacun doit avoir une méthode getXXXX et setXXXX permettant d'obtenir la valeur de l'attribut ou de lui affecter une valeur.

La méthode `toString` permet d'obtenir une représentation sous forme d'une chaîne de caractères de l'instance de la classe.

La méthode main crée ensuite deux instances de la classe Article, soit en utilisant le constructeur par défaut et en initialisant ensuite les attributs un à un (lignes 9 à 12), soit en utilisant le constructeur correspondant à la liste des attributs disponibles (ligne 14). Les deux instances sont ensuite affichées soit par l'appel de la fonction test qui permet de définir le format d'affichage, soit en utilisant la méthode `toString`. Dans ce cas, le formatage est imposé par la classe Article.

```
***** Article *****
1. package exercices.chapitre3.exercice1;
2.
3. public class Article
4. {
5.     private int reference;
6.     private String designation;
7.     private double prix;
8.
9.     public Article()
10.    {
11.        super();
12.    }
13.
14.     public Article(int reference)
15.    {
16.         this();
17.         setReference(reference);
18.     }
19.     public Article(int reference, String designation)
20.    {
21.         this(reference);
22.         setDesignation(designation);
23.     }
24.
25.     public Article(int reference, String designation,
double prix)
26.    {
27.        this(reference, designation);
28.        setPrix(prix);
29.    }
30.
31.     public int getReference()
32.    {
33.        return reference;
34.    }
35.
36.     public void setReference(int reference)
```

```

37.     {
38.         this.reference = reference;
39.     }
40.
41.     public String getDesignation()
42.     {
43.         return designation;
44.     }
45.
46.     public void setDesignation(String designation)
47.     {
48.         this.designation = designation;
49.     }
50.
51.     public double getPrix()
52.     {
53.         return prix;
54.     }
55.
56.     public void setPrix(double prix)
57.     {
58.         this.prix = prix;
59.     }
60.
61.     public String toString()
62.     {
63.         return getReference() + " " + getDesignation() +
" " + getPrix();
64.     }
65.
66. }
67.
***** Classe Principale *****/
1. package exercices.chapitre3.exercice1;
2.
3.
4. public class Principale
5. {
6.     public static void main(String[] args)
7.     {
8.         Article a1,a2;
9.         a1=new Article();
10.        a1.setReference(100);
11.        a1.setDesignation("tintin au congo");
12.        a1.setPrix(8.5);
13.
14.        a2=new Article(110,"Le Crabe aux pinces d'or",8.5);
15.
16.        test(a1);
17.        System.out.println(a1.toString());
18.
19.        test(a2);
20.        System.out.println(a2.toString());
21.
22.    }
23.
24.    public static void test(Article a)
25.    {
26.        System.out.println("reference : " + a.getReference());
27.        System.out.println("designation : "
+ a.getDesignation());
28.        System.out.println("prix : " + a.getPrix() + " €");
29.    }
30. }

```

Correction de l'exercice 2

Les deux classes Livre et Dvd héritent de la classe Article grâce au mot clé extends ajouté à la suite de leurs définitions. La structure de ces deux classes est similaire à celle de la classe Article. Les ajouts

correspondent aux attributs exigés par les classes Livre et Dvd. Les autres attributs et méthodes sont hérités de la classe Article. L'appel aux méthodes de la classe Article est effectuée par l'intermédiaire du mot clé super.

```
***** Article *****/
1. package exercices.chapitre3.exercice2;
2.
3. public class Article
4. {
5.     private int reference;
6.     private String designation;
7.     private double prix;
8.
9.     public Article()
10.    {
11.        super();
12.    }
13.
14.    public Article(int reference)
15.    {
16.        this();
17.        setReference(reference);
18.    }
19.    public Article(int reference, String designation)
20.    {
21.        this(reference);
22.        setDesignation(designation);
23.    }
24.
25.    public Article(int reference, String designation,
double prix)
26.    {
27.        this(reference, designation);
28.        setPrix(prix);
29.    }
30.
31.    public int getReference()
32.    {
33.        return reference;
34.    }
35.
36.    public void setReference(int reference)
37.    {
38.        this.reference = reference;
39.    }
40.
41.    public String getDesignation()
42.    {
43.        return designation;
44.    }
45.
46.    public void setDesignation(String designation)
47.    {
48.        this.designation = designation;
49.    }
50.
51.    public double getPrix() {
52.        return prix;
53.    }
54.
55.    public void setPrix(double prix)
56.    {
57.        this.prix = prix;
58.    }
59.
60.    public String toString()
61.    {
62.        return getReference() + " " + getDesignation() +
" " + getPrix();
```

```
63.      }
64.
65.  }
66.
/***** Livre *****/
1. package exercices.chapitre3.exercice2;
2.
3. public class Livre extends Article
4. {
5.     private String isbn;
6.     private int nbPages;
7.     private String auteur;
8.
9.     public Livre()
10.    {
11.        super();
12.    }
13.
14.     public Livre(int reference, String designation, double
prix, String isbn, int nbPages, String auteur)
15.    {
16.        super(reference, designation, prix);
17.        setISBN(isbn);
18.        setNbPages(nbPages);
19.        setAuteur(auteur);
20.    }
21.
22.     public String getISBN() {
23.         return isbn;
24.     }
25.
26.     public void setISBN(String isbn) {
27.         this.isbn = isbn;
28.     }
29.
30.     public int getNbPages() {
31.         return nbPages;
32.     }
33.
34.     public void setNbPages(int nbPages) {
35.         this.nbPages = nbPages;
36.     }
37.
38.     public String getAuteur() {
39.         return auteur;
40.     }
41.
42.     public void setAuteur(String auteur) {
43.         this.auteur = auteur;
44.     }
45.     public String toString()
46.     {
47.         return super.toString() + " " + getNbPages() +
" " + getAuteur();
48.     }
49.
50. }
/***** Dvd *****/
1. package exercices.chapitre3.exercice2;
2.
3. import java.time.Duration;
4.
5.
6. public class Dvd extends Article
7. {
8.     private Duration duree;
9.     private String realisateur;
10.
11.    public Dvd()
```

```

12.     {
13.         super();
14.     }
15.
16.     public Dvd(int reference, String designation, double
17.                prix, Duration duree, String realisateur)
17.     {
18.         super(reference, designation, prix);
19.
20.         setDuree(duree);
21.         setRealisateur(realisateur);
22.     }
23.
24.     public Duration getDuree()
25.     {
26.         return duree;
27.     }
28.
29.     public void setDuree(Duration duree)
30.     {
31.         this.duree = duree;
32.     }
33.
34.     public String getRealisateur()
35.     {
36.         return realisateur;
37.     }
38.
39.     public void setRealisateur(String realisateur)
40.     {
41.         this.realisateur = realisateur;
42.     }
43.
44.     public String toString()
45.     {
46.         return super.toString() + " " +
47.                getDuree().toMinutes() + " " + getRealisateur();
47.     }
48. }

***** Classe Principale *****/
1. package exercices.chapitre3.exercice2;
2.
3. import java.time.Duration;
4.
5. public class Principale
6. {
7.     public static void main(String[] args)
8.     {
9.         Livre l;
10.        Dvd d;
11.        l=new Livre();
12.        l.setReference(100);
13.        l.setDesignation("Le Crabe aux pinces d'or");
14.        l.setPrix(8.5);
15.        l.setNbPages(86);
16.        l.setAuteur("Hergé");
17.        testLivre(l);
18.        System.out.println(l.toString());
19.
20.        d=new Dvd();
21.        d.setReference(110);
22.        d.setDesignation("La soupe aux choux");
23.        d.setPrix(19.50);
24.        d.setDuree(Duration.ofMinutes(98));
25.        d.setRealisateur("Girault");
26.        testDvd(d);
27.        System.out.println(d.toString());
28.

```

```
29.     }
30. }
```

Correction de l'exercice 3

Pour ajouter les informations supplémentaires concernant l'auteur ou le réalisateur, il préférable de créer une classe Personne regroupant nom, prénom et date de naissance. Cette classe est ensuite utilisée pour remplacer le type String des attributs auteur et réalisateur des classes Livre et Dvd. Il faut bien sûr modifier la signature des constructeurs et des méthodes getXXXX et setXXXX pour prendre en compte cette modification.

```
***** Article *****/
1. package exercices.chapitre3.exercice3;
2.
3. public class Article
4. {
5.     private int reference;
6.     private String designation;
7.     private double prix;
8.
9.     public Article()
10.    {
11.        super();
12.    }
13.
14.    public Article(int reference)
15.    {
16.        this();
17.        setReference(reference);
18.    }
19.    public Article(int reference, String designation)
20.    {
21.        this(reference);
22.        setDesignation(designation);
23.    }
24.
25.    public Article(int reference, String
designation, double prix)
26.    {
27.        this(reference, designation);
28.        setPrix(prix);
29.    }
30.
31.    public int getReference()
32.    {
33.        return reference;
34.    }
35.
36.    public void setReference(int reference)
37.    {
38.        this.reference = reference;
39.    }
40.
41.    public String getDesignation()
42.    {
43.        return designation;
44.    }
45.
46.    public void setDesignation(String designation)
47.    {
48.        this.designation = designation;
49.    }
50.
51.    public double getPrix() {
52.        return prix;
53.    }
54.
```

```
55.     public void setPrix(double prix)
56.     {
57.         this.prix = prix;
58.     }
59.
60.     public String toString()
61.     {
62.         return getReference() + " " + getDesignation() +
" " + getPrix();
63.     }
64.
65. }
```

***** Livre *****

```
1. package exercices.chapitre3.exercice3;
2.
3. public class Livre extends Article
4. {
5.     private String isbn;
6.     private int nbPages;
7.     private Personne auteur;
8.
9.     public Livre()
10.    {
11.        super();
12.    }
13.
14.    public Livre(int reference, String designation, double
prix, String isbn, int nbPages, Personne auteur)
15.    {
16.        super(reference, designation, prix);
17.        setISBN(isbn);
18.        setNbPages(nbPages);
19.        setAuteur(auteur);
20.    }
21.
22.    public String getISBN() {
23.        return isbn;
24.    }
25.
26.    public void setISBN(String isbn) {
27.        this.isbn = isbn;
28.    }
29.
30.    public int getNbPages() {
31.        return nbPages;
32.    }
33.
34.    public void setNbPages(int nbPages) {
35.        this.nbPages = nbPages;
36.    }
37.
38.    public Personne getAuteur() {
39.        return auteur;
40.    }
41.
42.    public void setAuteur(Personne auteur) {
43.        this.auteur = auteur;
44.    }
45.    public String toString()
46.    {
47.        return super.toString() + " " + getNbPages() + "
" + getAuteur();
48.    }
49.
50. }
```

***** Dvd *****

```
1. package exercices.chapitre3.exercice3;
```

```
2.
3. import java.time.Duration;
4.
5.
6. public class Dvd extends Article
7. {
8.     private Duration duree;
9.     private Personne realisateur;
10.
11.    public Dvd()
12.    {
13.        super();
14.    }
15.
16.    public Dvd(int reference, String designation, double
17. prix, Duration duree, Personne realisateur)
17.    {
18.        super(reference, designation, prix);
19.
20.        setDuree(duree);
21.        setRealisateur(realisateur);
22.    }
23.
24.    public Duration getDuree()
25.    {
26.        return duree;
27.    }
28.
29.    public void setDuree(Duration duree)
30.    {
31.        this.duree = duree;
32.    }
33.
34.    public Personne getRealisateur()
35.    {
36.        return realisateur;
37.    }
38.
39.    public void setRealisateur(Personne realisateur)
40.    {
41.        this.realisateur = realisateur;
42.    }
43.
44.    public String toString()
45.    {
46.        return super.toString() + "
47. " + getDuree().toMinutes() + " " + getRealisateur();
48.    }

```

```
***** Personne *****/
1. package exercices.chapitre3.exercice3;
2.
3. import java.time.LocalDate;
4.
5. public class Personne
6. {
7.     private String nom;
8.     private String prenom;
9.     private LocalDate date_nais;
10.
11.    public Personne()
12.    {
13.        super();
14.    }
15.
16.    public Personne(String n, String p, LocalDate d)
17.    {
18.        this.nom=n;
```

```

19.         this.prenom=p;
20.         this.date_nais=d;
21.     }
22.
23.     public String getNom()
24.     {
25.         return nom;
26.     }
27.
28.     public void setNom(String nom)
29.     {
30.         this.nom = nom;
31.     }
32.
33.     public String getPrenom()
34.     {
35.         return prenom;
36.     }
37.
38.     public void setPrenom(String prenom)
39.     {
40.         this.prenom = prenom;
41.     }
42.
43.     public LocalDate getDate_nais()
44.     {
45.         return date_nais;
46.     }
47.
48.     public void setDate_nais(LocalDate date_nais)
49.     {
50.         this.date_nais = date_nais;
51.     }
52.
53.     public String toString() {
54.         return prenom + " " + nom;
55.     }
56.
57. }
***** Classe Principale *****/

```

```

1. package exercices.chapitre3.exercice3;
2.
3. import java.time.Duration;
4. import java.time.LocalDate;
5.
6.
7.
8. public class Principale
9. {
10.     public static void main(String[] args)
11.     {
12.         Livre l;
13.         Dvd d;
14.         l=new Livre();
15.         l.setReference(100);
16.         l.setDesignation("Le Crabe aux pinces d'or");
17.         l.setPrix(8.5);
18.         l.setNbPages(86);
19.         l.setAuteur(new
Personne("Hergé","Georges",LocalDate.of(1907,05,22)));
20.         testLivre(l);
21.         System.out.println(l.toString());
22.
23.         d=new Dvd();
24.         d.setReference(110);
25.         d.setDesignation("La soupe aux choux");
26.         d.setPrix(19.50);
27.         d.setDuree(Duration.ofMinutes(98));
28.         d.setRealisateur(new

```

```

Personne("Girault","jean",LocalDate.of(1924,05,9)));
29.      testDvd(d);
30.      System.out.println(d.toString());
31.
32.  }
33.
34.  public static void test(Article a)
35.  {
36.      System.out.println("reference : " +
a.getReference());
37.      System.out.println("designation : " +
a.getDesignation());
38.      System.out.println("prix : " + a.getPrix() + " €");
39.  }
40.
41.  public static void testLivre(Livre l)
42.  {
43.      test(l);
44.      System.out.println("nombre de pages : " +
l.getNbPages());
45.      System.out.println("auteur : " +
l.getAuteur().toString());
46.  }
47.
48.  public static void testDvd(Dvd d)
49.  {
50.      test(d);
51.      System.out.println("durée : " +
d.getDuree().toMinutes() + " minutes");
52.      System.out.println("réalisateur : " +
d.getRealisateur().toString());
53.  }
54. }
```

Correction de l'exercice 4

Dans l'exercice précédent, nous avons mis en place un lien entre un dvd ou un livre et l'auteur ou le réalisateur correspondant. Il est très souvent intéressant d'avoir un lien bidirectionnel. Dans notre cas, celui-ci permet d'obtenir très facile-ment la liste des œuvres d'un auteur ou d'un réalisateur. Il suffit simplement d'ajouter à la classe Personne un attribut de type ArrayList. Celui-ci permet d'associer à un auteur ou à un réalisateur la liste de toutes ses œuvres. L'instance de l'objet ArrayList est créée dans le constructeur pour permettre le stockage des œuvres de cette personne. Les méthodes setAuteur et setRealisateur des classes Livre et Dvd sont également modifiées. Lors de l'association, on vérifie si le Dvd ou le Livre est déjà dans la liste des œuvres de cette personne. Si ce n'est pas le cas, le Dvd ou le Livre est ajouté à la liste.

```

***** Article *****/
1. package exercices.chapitre3.exercice4;
2.
3. public class Article
4. {
5.     private int reference;
6.     private String designation;
7.     private double prix;
8.
9.     public Article()
10.    {
11.        super();
12.    }
13.
14.     public Article(int reference)
15.    {
16.        this();
17.        setReference(reference);
18.    }
19.    public Article(int reference, String designation)
```

```
20.     {
21.         this(reference);
22.         setDesignation(designation);
23.     }
24.
25.     public Article(int reference, String designation,
double prix)
26.     {
27.         this(reference, designation);
28.         setPrix(prix);
29.     }
30.
31.     public int getReference()
32.     {
33.         return reference;
34.     }
35.
36.     public void setReference(int reference)
37.     {
38.         this.reference = reference;
39.     }
40.
41.     public String getDesignation()
42.     {
43.         return designation;
44.     }
45.
46.     public void setDesignation(String designation)
47.     {
48.         this.designation = designation;
49.     }
50.
51.     public double getPrix() {
52.         return prix;
53.     }
54.
55.     public void setPrix(double prix)
56.     {
57.         this.prix = prix;
58.     }
59.
60.     public String toString()
61.     {
62.         return getReference() + " " + getDesignation() + "
" + getPrix();
63.     }
64.
65. }
```

***** Livre *****

```
1. package exercices.chapitre3.exercice4;
2.
3. import java.util.ArrayList;
4.
5.
6.
7. public class Livre extends Article
8. {
9.     private String isbn;
10.    private int nbPages;
11.    private Personne auteur;
12.
13.    public Livre()
14.    {
15.        super();
16.    }
17.
18.    public Livre(int reference, String designation, double
prix, String isbn, int nbPages, Personne auteur)
```

```

19.     {
20.         super(reference,designation,prix);
21.         setISBN(isbn);
22.         setNbPages(nbPages);
23.         setAuteur(auteur);
24.     }
25.
26.     public String getISBN() {
27.         return isbn;
28.     }
29.
30.     public void setISBN(String isbn) {
31.         this.isbn = isbn;
32.     }
33.
34.     public int getNbPages() {
35.         return nbPages;
36.     }
37.
38.     public void setNbPages(int nbPages) {
39.         this.nbPages = nbPages;
40.     }
41.
42.     public Personne getAuteur() {
43.         return auteur;
44.     }
45.
46.     public void setAuteur(Personne auteur) {
47.         this.auteur = auteur;
48.         ArrayList<Article> lst;
49.         lst=auteur.getOeuvres();
50.         if (!lst.contains(this))
51.         {
52.             lst.add(this);
53.         }
54.     }
55.     public String toString()
56.     {
57.         return super.toString() + " " + getNbPages() + "
" + getAuteur();
58.     }
59.
60. }

```

***** Dvd *****

```

1. package exercices.chapitre3.exercice4;
2.
3. import java.time.Duration;
4. import java.util.ArrayList;
5.
6.
7. public class Dvd extends Article
8. {
9.     private Duration duree;
10.    private Personne realisateur;
11.
12.    public Dvd()
13.    {
14.        super();
15.    }
16.
17.    public Dvd(int reference,String designation,double
prix,Duration duree,Personne realisateur)
18.    {
19.        super(reference,designation,prix);
20.
21.        setDuree(duree);
22.        setRealisateur(realisateur);
23.    }

```

```

24.
25.     public Duration getDuree()
26.     {
27.         return duree;
28.     }
29.
30.     public void setDuree(Duration duree)
31.     {
32.         this.duree = duree;
33.     }
34.
35.     public Personne getRealisateur()
36.     {
37.         return realisateur;
38.     }
39.
40.     public void setRealisateur(Personne realisateur)
41.     {
42.         this.realisateur = realisateur;
43.         ArrayList<Article> lst;
44.         lst=realisateur.getOeuvres();
45.         if (!lst.contains(this))
46.         {
47.             lst.add(this);
48.         }
49.     }
50.
51.     public String toString()
52.     {
53.         return super.toString() + " " +
getDuree().toMinutes() + " " + getRealisateur();
54.     }
55. }
```

***** Personne *****

```

1. package exercices.chapitre3.exercice4;
2.
3. import java.time.LocalDate;
4. import java.util.ArrayList;
5.
6. public class Personne
7. {
8.     private String nom;
9.     private String prenom;
10.    private LocalDate date_nais;
11.    private ArrayList<Article> oeuvres;
12.
13.    public Personne()
14.    {
15.        super();
16.        oeuvres=new ArrayList<> ();
17.    }
18.
19.    public Personne(String n,String p,LocalDate d)
20.    {
21.        this();
22.        this.nom=n;
23.        this.prenom=p;
24.        this.date_nais=d;
25.    }
26.
27.    public String getNom()
28.    {
29.        return nom;
30.    }
31.
32.    public void setNom(String nom)
33.    {
34.        this.nom = nom;
```

```
35.     }
36.
37.     public String getPrenom()
38.     {
39.         return prenom;
40.     }
41.
42.     public void setPrenom(String prenom)
43.     {
44.         this.prenom = prenom;
45.     }
46.
47.     public LocalDate getDate_nais()
48.     {
49.         return date_nais;
50.     }
51.
52.     public void setDate_nais(LocalDate date_nais)
53.     {
54.         this.date_nais = date_nais;
55.     }
56.
57.     public ArrayList<Article> getOeuvres()
58.     {
59.         return oeuvres;
60.     }
61.
62.     public String toString() {
63.         return prenom + " " + nom;
64.     }
65. }
```

Introduction

Jusqu'à présent, tous les exemples de code que nous avons réalisés fonctionnent exclusivement en mode caractères. Les informations sont affichées dans une console et également saisies à partir de celle-ci. La simplicité de ce mode de fonctionnement est un atout indéniable pour l'apprentissage d'un langage. Par contre, la plupart des utilisateurs de vos futures applications s'attendent certainement à avoir une interface un petit peu moins austère qu'un écran en mode caractères. Nous allons donc étudier dans ce chapitre comment fonctionnent les interfaces graphiques avec Java. Vous allez rapidement vous apercevoir que la conception d'interfaces graphiques en Java n'est pas très simple et nécessite l'écriture de nombreuses lignes de code. Dans la pratique, de nombreux outils de développement sont capables de prendre en charge la génération d'une grande partie de ce code en fonction de la conception graphique de l'application que vous dessinez. Il est cependant important de bien comprendre les principes de fonctionnement de ce code pour éventuellement intervenir dessus et l'optimiser. Nous n'utiliserons pas dans ce chapitre d'outil spécifique mais nous conserverons notre trio éditeur de texte, compilateur, machine virtuelle.

1. Les bibliothèques graphiques

Le langage Java propose deux bibliothèques dédiées à la conception d'interfaces graphiques. La bibliothèque AWT et la bibliothèque SWING. Les principes d'utilisation sont quasiment identiques pour ces deux bibliothèques. L'utilisation simultanée de ces deux bibliothèques dans une même application peut provoquer des problèmes de fonctionnement et doit être évitée.

a. La bibliothèque AWT

Cette bibliothèque est la toute première disponible pour le développement d'interfaces graphiques. Elle contient une multitude de classes et interfaces permettant la définition et la gestion d'interfaces graphiques. Cette bibliothèque utilise en fait les fonctionnalités graphiques du système d'exploitation. Ce n'est donc pas le code présent dans cette bibliothèque qui assure le rendu graphique des différents composants. Ce code sert uniquement d'intermédiaire avec le système d'exploitation. Son utilisation est de ce fait relativement économique en ressources. Par contre elle souffre de plusieurs inconvénients.

- L'aspect visuel de chaque composant étant lié à la représentation qu'en fait le système d'exploitation, il est parfois délicat de développer une application ayant une apparence cohérente sur tous les systèmes. La taille et la position des différents composants étant les deux éléments principalement affectés par ce problème.
- Pour que cette bibliothèque soit compatible avec tous les systèmes d'exploitation, les composants qu'elle contient sont donc limités aux plus courants (boutons, zone de texte, listes...).

b. La bibliothèque Swing

Cette bibliothèque a été conçue pour pallier les principales insuffisances de la bibliothèque AWT. Cette amélioration a été obtenue en écrivant entièrement cette bibliothèque en Java sans pratiquement faire appel aux services du système d'exploitation. Seuls quelques éléments graphiques (fenêtres et boîtes de dialogue) sont encore en relation avec le système d'exploitation. Pour les autres composants, c'est le code de la bibliothèque Swing qui détermine entièrement leurs aspects et comportements.

La bibliothèque Swing contient donc une quantité impressionnante de classes servant à redéfinir les composants graphiques. Il ne faut cependant pas penser que la bibliothèque Swing rend complètement obsolète la bibliothèque AWT. Beaucoup d'éléments de la bibliothèque AWT sont d'ailleurs repris dans la bibliothèque Swing. Nous utiliserons principalement cette bibliothèque dans le reste de ce chapitre.

2. Constitution de l'interface graphique d'une application

La conception de l'interface graphique d'une application consiste essentiellement à créer des instances des classes représentant les différents éléments nécessaires, modifier les caractéristiques de ces instances de

classe, les assembler et prévoir le code de gestion des différents événements pouvant intervenir au cours du fonctionnement de l'application. Une application graphique est donc constituée d'une multitude d'éléments superposés ou imbriqués. Parmi ces éléments, l'un d'entre eux joue un rôle prépondérant dans l'application. Il est souvent appelé conteneur de premier niveau. C'est lui qui va interagir avec le système d'exploitation et contenir tous les autres éléments. En général ce conteneur de premier niveau ne contient pas directement les composants graphiques mais d'autres conteneurs sur lesquels sont placés les composants graphiques. Pour faciliter la disposition de ces éléments les uns par rapport aux autres, nous pouvons utiliser l'aide d'un gestionnaire de mise en page. Cette superposition d'éléments peut être assimilée à une arborescence au sommet de laquelle nous avons le conteneur de premier niveau et dont les différentes branches sont constituées d'autres conteneurs. Les feuilles de l'arborescence correspondant aux composants graphiques.

Le conteneur de premier niveau étant l'élément indispensable de toute application graphique, nous allons donc commencer par étudier en détail ces caractéristiques et son utilisation puis nous étudierons les principaux composants graphiques.

Conception d'une interface graphique

Nous avons vu un petit peu plus haut que toute application graphique est au moins constituée d'un conteneur de premier niveau. La bibliothèque Swing dispose de trois classes permettant de jouer ce rôle :

JApplet : représente une fenêtre graphique embarquée à l'intérieur d'une page html pour être prise en charge par un navigateur. Cet élément est étudié en détail dans le chapitre qui lui est consacré.

JWindow : représente une fenêtre graphique la plus rudimentaire qui soit. Celle-ci ne dispose pas de barre de titre, de menu système, pas de bordure, c'est en fait un simple rectangle sur l'écran. Cette classe est rarement utilisée sauf pour l'affichage d'un écran d'accueil lors du démarrage d'une application (*splash screen*).

JFrame : représente une fenêtre graphique complète et pleinement fonctionnelle. Elle dispose d'une barre de titre, d'un menu système, d'une bordure, elle peut facilement accueillir un menu, c'est bien sûr cet élément que nous allons utiliser dans la très grande majorité des cas.

1. Les fenêtres

La classe JFrame est l'élément indispensable de toute application graphique. Comme pour une classe normale nous devons créer une instance, modifier éventuellement les propriétés et utiliser les méthodes. Voici donc le code de la première application graphique.

```
package fr.eni;
import javax.swing.JFrame;
public class Principale {
    public static void main(String[] args)
    {
        JFrame fenetre;
        // création de l'instance de la classe JFrame
        fenetre=new JFrame();
        // modification de la position et de la
        // taille de la fenêtre
        fenetre.setBounds(0,0,300,400);
        // modification du titre de la fenêtre
        fenetre.setTitle("première fenêtre en JAVA");
        // affichage de la fenêtre
        fenetre.setVisible(true);
    }
}
```

Et le résultat de son exécution :



C'est simple d'utilisation et très efficace. C'est d'ailleurs tellement efficace que vous ne pouvez pas arrêter l'application. En effet même si la fenêtre est fermée par l'utilisateur, cette fermeture ne provoque pas la suppression de l'instance de la JFrame de la mémoire. La seule solution pour arrêter l'application est de stopper la machine virtuelle Java avec la combinaison de touches [Ctrl] C. Il faut bien sûr prévoir une autre solution pour terminer plus facilement l'exécution de l'application et faire en sorte que celle-ci s'arrête à la fermeture de la fenêtre.

La première solution consiste à gérer les événements se produisant lors de la fermeture de la fenêtre et dans un de ces événements provoquer l'arrêt de l'application. Cette solution sera étudiée dans le paragraphe consacré à la gestion des événements.

La deuxième solution utilise des comportements prédéfinis pour la fermeture de la fenêtre. Ces comportements sont déterminés par la méthode `setDefaultCloseOperation`. Plusieurs constantes sont définies pour déterminer l'action entreprise à la fermeture de la fenêtre.

DISPOSE_ON_CLOSE : cette option provoque l'arrêt de l'application lors de la fermeture de la dernière fenêtre prise en charge par la machine virtuelle.

`DO NOTHING ON CLOSE` : avec cette option, il ne se passe rien lorsque l'utilisateur demande la fermeture de la fenêtre. Il est dans ce cas obligatoire de gérer les événements pour que l'action de l'utilisateur provoque un effet sur la fenêtre ou l'application.

`EXIT_ON_CLOSE` : cette option provoque l'arrêt de l'application même si d'autres fenêtres sont encore visibles.

`HIDE_ON_CLOSE` : avec cette option, la fenêtre est simplement masquée par un appel à sa méthode `setVisible(false)`.

La classe `JFrame` se trouve située à la fin d'une hiérarchie de classes assez importante et implémente de nombreuses interfaces. De ce fait elle possède donc de nombreuses méthodes et attributs.

Class JFrame

```
graph TD; Object[java.lang.Object] --> Component[java.awt.Component]; Component --> Container[java.awt.Container]; Container --> Window[java.awt.Window]; Window --> Frame[java.awt.Frame]; Frame --> JFrame[javax.swing.JFrame]
```

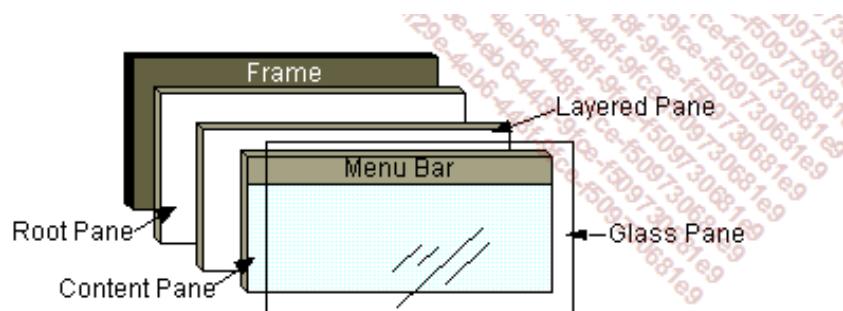
The diagram illustrates the class hierarchy in Java. It starts with `java.lang.Object` at the top, which branches down to `java.awt.Component`. From there, it further branches down to `java.awt.Container`, then to `java.awt.Window`, and finally to `java.awt.Frame`. The `java.awt.Frame` class then branches down to `javax.swing.JFrame`.

All Implemented Interfaces:

[ImageObserver](#), [MenuContainer](#), [Serializable](#), [Accessible](#), [RootPaneContainer](#), [WindowConstants](#)

Le but de cet ouvrage n'est pas de reprendre entièrement la documentation du JDK, aussi il ne présente pas toutes les méthodes disponibles mais simplement les plus couramment utilisées au fur et à mesure des besoins. Il peut être cependant intéressant de parcourir la documentation avant de se lancer dans la conception d'une méthode pour déterminer si ce que l'on souhaite réaliser n'a pas déjà été prévu par les concepteurs de Java.

Maintenant que nous sommes capables d'afficher une fenêtre, le plus gros de notre travail va consister à ajouter un contenu à la fenêtre. Avant de pouvoir ajouter quelque chose sur une fenêtre, il faut bien comprendre sa structure qui est relativement complexe. Un objet `JFrame` est composé de plusieurs éléments superposés jouant chacun un rôle bien spécifique dans la gestion de la fenêtre.



L'élément `RootPane` correspond au conteneur des trois autres éléments. L'élément `LayeredPane` est lui responsable de la gestion de la position des éléments aussi bien sur les axes X et Y que sur l'axe Z ce qui permet la superposition de différents éléments. L'élément `ContentPane` est le conteneur de base de tous

les éléments ajoutés sur la fenêtre. C'est bien sûr à lui que nous allons confier les différents composants de l'interface de l'application. Par-dessous le ContentPane, se superpose le GlassPane comme on le fait avec une vitre placée sur une photo. Il présente d'ailleurs beaucoup de similitudes avec la vitre.

- Il est transparent par défaut.
- Ce qui est dessiné sur le GlassPane masque les autres éléments.
- Il est capable d'intercepter les événements liés à la souris avant que ceux-ci n'atteignent les autres composants.

De tous ces éléments, c'est incontestablement le ContentPane que nous allons le plus utiliser. Celui-ci est accessible par la méthode `getContentPane` de la classe `JFrame`. Il est techniquement possible de placer des composants directement sur l'objet `ContentPane` mais c'est une pratique déconseillée par Oracle. Il est préférable d'intercaler un conteneur intermédiaire qui lui va contenir les composants et de placer ce conteneur sur le `ContentPane`. Le composant `JPanel` est le plus couramment utilisé dans ce rôle.

Le scénario classique de conception d'une interface graphique consiste donc à créer les différents composants puis à les placer sur un conteneur et enfin à placer ce conteneur sur le `ContentPane` de la fenêtre. L'exemple suivant met cela en application en créant une interface utilisateur composée de trois boutons.

```
package fr.eni;

import java.awt.Graphics;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Principale {

    public static void main(String[] args)
    {
        // création de la fenêtre
        JFrame fenetre;
        fenetre=new JFrame();
        fenetre.setTitle("première fenêtre en JAVA");
        fenetre.setBounds(0,0,300,100);
        fenetre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // création des trois boutons
        JButton b1,b2,b3;
        b1=new JButton("Rouge");
        b2=new JButton("Vert");
        b3=new JButton("Bleu");
        // création du conteneur intermédiaire
        JPanel pano;
        pano=new JPanel();
        // ajout des boutons sur le conteneur intermédiaire
        pano.add(b1);
        pano.add(b2);
        pano.add(b3);
        // ajout du conteneur intermédiaire sur le ContentPane
        fenetre.getContentPane().add(pano);
        // affichage de la fenêtre
        fenetre.setVisible(true);
    }
}
```

À l'exécution, ce code affiche la fenêtre suivante :



L'étape suivante de notre étude va nous permettre de déterminer ce que doit faire l'application lorsque l'utilisateur va cliquer sur l'un des boutons.

2. La gestion des événements

Tous les systèmes d'exploitation utilisant une interface graphique doivent en permanence surveiller les différents périphériques de saisie pour détecter les actions de l'utilisateur et transmettre ces actions aux différentes applications. Pour chaque action de l'utilisateur, un événement est créé. Ces événements sont ensuite proposés à chaque application qui détermine si elle est concernée par l'événement et si c'est le cas, que doit-elle faire pour y répondre. La manière de gérer ces événements diffère suivant les langages. Dans certains cas, chaque composant dispose d'une portion de code prédefinie associée automatiquement à chaque type d'événement. Le rôle du développeur consiste dans ce cas à personnaliser les différentes portions de code associées aux événements. Dans d'autres langages, les événements sont simplement placés dans une file par le système et c'est alors le développeur qui doit surveiller cette file pour déterminer quel composant est concerné par l'événement et provoquer l'exécution de la portion de code qu'il aura prévu. La démarche utilisée par Java est une technique intermédiaire. Java se charge de déterminer quel événement vient de se produire et sur quel élément. Le développeur est responsable du choix de la portion de code qui va traiter l'événement. D'un point de vue plus technique, l'élément à l'origine de l'événement est appelé source d'événement et l'élément contenant la portion de code chargée de gérer l'événement est appelée écouteur d'événement. Les sources d'événements gèrent, pour chaque événement qu'elles peuvent déclencher, une liste leur permettant de savoir quels écouteurs doivent être avertis si l'événement se produit. Les sources d'événements et les écouteurs d'événements sont bien entendu des objets.

Il faut bien sûr que les écouteurs soient prévus pour gérer les événements que va leur transmettre la source d'événement. Pour garantir cela, à chaque type d'événement correspond une interface que doit implémenter un objet s'il veut être candidat pour la gestion de cet événement. Pour éviter la multiplication des interfaces (déjà très nombreuses), les événements sont regroupés par catégories. Le nom de ces interfaces respecte toujours la convention suivante :

La première partie du nom est représentative de la catégorie d'événements pouvant être gérés par les objets implémentant cette interface. Le nom se termine toujours par Listener.

Nous avons par exemple l'interface MouseMotionListener correspondant aux événements déclenchés par les déplacements de la souris ou l'interface ActionListener correspondant à un clic sur un bouton. C'est dans chacune de ces interfaces que nous trouvons les signatures des différentes méthodes associées à chaque événement.

```
public interface MouseMotionListener
extends EventListener
{
    void mouseDragged(MouseEvent e);
    void mouseMoved(MouseEvent e);
}
```

Chacune des méthodes attend comme argument un objet représentant l'événement lui-même. Cet objet est créé automatiquement lors du déclenchement de l'événement puis passé comme argument à la méthode chargée de gérer l'événement dans l'écouteur d'événement. Il contient généralement des informations complémentaires concernant l'événement et il est spécifique à chaque type d'événement.

Nous avons donc besoin de créer des classes implémentant ces interfaces. De ce point de vue, nous avons une multitude de possibilités :

- Créer une classe "normale" implémentant l'interface.

- Implémenter l'interface dans une classe déjà existante.
- Créer une classe interne implémentant l'interface.
- Créer une classe interne anonyme implémentant l'interface.

Dans certains cas, vous n'avez peut-être pas besoin de gérer tous les événements présents dans l'interface. Cependant vous êtes tout de même obligé d'écrire toutes les méthodes exigées par l'interface même si plusieurs d'entre elles ne contiennent aucun code. Ceci peut nuire à la lisibilité du code. Pour pallier ce problème, Java propose pour pratiquement chaque interface XXXXXListener une classe abstraite correspondante implémentant déjà l'interface donc contenant les méthodes exigées par l'interface. Ces méthodes ne contiennent aucun code puisque le traitement de chaque événement doit être spécifique à chaque application. Ces classes utilisent la même convention de nommage que les interfaces hormis le fait que Listener est remplacé par Adapter. Nous avons par exemple la classe MouseMotionAdapter implémentant l'interface MouseMotionListener. Ces classes sont utilisables de plusieurs façons :

- Créer une classe "normale" héritant d'une de ces classes.
- Créer une classe interne héritant d'une de ces classes.
- Créer une classe interne anonyme héritant d'une de ces classes.

L'utilisation d'une classe interne anonyme est la solution la plus fréquemment utilisée avec le petit inconvénient d'avoir une syntaxe difficilement lisible lorsque l'on n'y est pas habitué.

Pour clarifier tout cela, nous allons illustrer chacune de ces possibilités par un petit exemple. Cet exemple va nous permettre de terminer proprement l'application lors de la fermeture de la fenêtre principale en appelant la méthode System.exit(0). Cette solution permet d'effectuer des vérifications avant l'arrêt de l'application (sauvegarde, affichage d'un message de confirmation, déconnexion de l'utilisateur...). Il faut, dans ce cas, modifier la propriété DefaultCloseOperation de la fenêtre avec la valeur DO NOTHING ON CLOSE pour qu'il n'y ait plus d'action par défaut.

Cette méthode doit être appelée lors de la détection de la fermeture de la fenêtre. Pour cela, nous devons gérer les événements liés à la fenêtre et plus particulièrement l'événement windowClosing qui est déclenché au moment où l'utilisateur demande la fermeture de la fenêtre par le menu système. L'interface WindowListener est tout à fait adaptée pour ce genre de travail.

Notre base de travail est constituée des deux classes suivantes :

```
package fr.eni;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Ecran
extends JFrame
{
    public Ecran()
    {
        setTitle("première fenêtre en JAVA");
        setBounds(0,0,300,100);
        setDefaultCloseOperation(JFrame.DO NOTHING_ON_CLOSE);
        // création des trois boutons
        JButton b1,b2,b3;
        b1=new JButton("Rouge");
        b2=new JButton("Vert");
        b3=new JButton("Bleu");
        // création du conteneur intermédiaire
        JPanel pano;
        pano=new JPanel();
        // ajout des boutons sur le conteneur intermédiaire
        pano.add(b1);
        pano.add(b2);
        pano.add(b3);
        // ajout du conteneur sur le ContentPane
```

```

        getContentPane().add(pano);
    }

}

package fr.eni;

public class Principale {

    public static void main(String[] args)

    {
        // création de la fenêtre
        Ecran fenetre;
        fenetre=new Ecran();
        // affichage de la fenêtre
        fenetre.setVisible(true);
    }
}

```

Si nous exécutons ce code, la fenêtre apparaît mais il n'est plus possible de la fermer et encore moins d'arrêter l'application. Voyons maintenant comment remédier à ce problème avec les différentes solutions évoquées plus haut.

Utilisation d'une classe "normale" implémentant l'interface

```

package fr.eni;

import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

public class EcouteurFenetre implements WindowListener {

    public void windowActivated(WindowEvent arg0)
    {
    }
    public void windowClosed(WindowEvent arg0)
    {
    }
    public void windowClosing(WindowEvent arg0)
    {
        System.exit(0);
    }
    public void windowDeactivated(WindowEvent arg0)
    {
    }
    public void windowDeiconified(WindowEvent arg0)
    {
    }
    public void windowIconified(WindowEvent arg0)
    {
    }
    public void windowOpened(WindowEvent arg0)
    {
    }
}

```

```

package fr.eni;

public class Principale {

    public static void main(String[] args)

    {
        // création de la fenêtre
        Ecran fenetre;
        fenetre=new Ecran();

```

```

        // création d'une instance de la classe chargée
        // de gérer les événements
        EcouteurFenetre ef;
        ef=new EcouteurFenetre();
        // référencement de cette instance de classe
        // comme écouteur d'événement pour la fenêtre
        fenetre.addWindowListener(ef);
        // affichage de la fenêtre
        fenetre.setVisible(true);
    }
}

```

Implémenter l'interface dans une classe déjà existante

Dans cette solution, nous allons confier à la classe représentant la fenêtre le soin de gérer ses propres événements en lui faisant implémenter l'interface WindowListener.

```

package fr.eni;

import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Ecran extends JFrame
    implements WindowListener

{
    public Ecran()
    {
        setTitle("première fenêtre en JAVA");
        setBounds(0,0,300,100);
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        // création des trois boutons
        JButton b1,b2,b3;
        b1=new JButton("Rouge");
        b2=new JButton("Vert");
        b3=new JButton("Bleu");
        // création du conteneur intermédiaire
        JPanel pano;
        pano=new JPanel();
        // ajout des boutons sur le conteneur intermédiaire
        pano.add(b1);
        pano.add(b2);
        pano.add(b3);
        // ajout du conteneur intermédiaire sur le ContentPane
        getContentPane().add(pano);
        // référencement de la fenêtre elle-même
        // comme écouteur de ses propres événements

        addWindowListener(this);
    }
    public void windowActivated(WindowEvent arg0)
    {
    }
    public void windowClosed(WindowEvent arg0)
    {
    }
    public void windowClosing(WindowEvent arg0)
    {
        System.exit(0);
    }
    public void windowDeactivated(WindowEvent arg0)
    {
    }
    public void windowDeiconified(WindowEvent arg0)
    {
    }
}

```

```

    }
}

public void windowIconified(WindowEvent arg0)
{
}

public void windowOpened(WindowEvent arg0)
{
}

}

package fr.eni;

public class Principale {

    public static void main(String[] args)

    {
        // création de la fenêtre
        Ecran fenetre;
        fenetre=new Ecran();
        // affichage de la fenêtre
        fenetre.setVisible(true);
    }
}

```

Avec cette solution, le code est centralisé dans une seule et unique classe. S'il y a de nombreux événements à gérer, cette classe va comporter une multitude de méthodes.

Créer une classe interne implémentant l'interface

Cette solution est un mélange des deux précédentes puisque nous avons une classe spécifique pour la gestion des événements mais celle-ci est définie à l'intérieur de la classe correspondant à la fenêtre.

```

package fr.eni;

import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Ecran extends JFrame

{
    public Ecran()
    {
        setTitle("première fenêtre en JAVA");
        setBounds(0,0,300,100);
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        // création des trois boutons
        JButton b1,b2,b3;
        b1=new JButton("Rouge");
        b2=new JButton("Vert");
        b3=new JButton("Bleu");
        // création du conteneur intermédiaire
        JPanel pano;
        pano=new JPanel();
        // ajout des boutons sur le conteneur intermédiaire
        pano.add(b1);
        pano.add(b2);
        pano.add(b3);
        // ajout du conteneur intermédiaire sur le ContentPane
        getContentPane().add(pano);
        // création d'une instance de la classe chargée
        // de gérer les événements
        EcouteurFenetre ef;
        ef=new EcouteurFenetre();
    }
}

```

```

        // référencement de cette instance de classe
        // comme écouteur d'événement pour la fenêtre

        addWindowListener(ef);
    }

    public class EcouteurFenetre implements WindowListener
    {
        public void windowActivated(WindowEvent arg0)
        {
        }
        public void windowClosed(WindowEvent arg0)
        {
        }
        public void windowClosing(WindowEvent arg0)
        {
            System.exit(0);
        }
        public void windowDeactivated(WindowEvent arg0)
        {
        }
        public void windowDeiconified(WindowEvent arg0)
        {
        }
        public void windowIconified(WindowEvent arg0)
        {
        }
        public void windowOpened(WindowEvent arg0)
        {
        }
    }
}

package fr.eni;

public class Principale {

    public static void main(String[] args)
    {
        // création de la fenêtre
        Ecran fenetre;
        fenetre=new Ecran();
        // affichage de la fenêtre
        fenetre.setVisible(true);
    }
}

```

Avec cette solution les responsabilités sont bien réparties entre plusieurs classes, par contre nous allons avoir une multiplication du nombre de classes.

Créer une classe interne anonyme implémentant l'interface

Cette solution est une petite variante de la précédente puisque nous avons toujours une classe spécifique chargée de la gestion des événements mais celle-ci est déclarée au moment de son instanciation.

```

package fr.eni;

import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Ecran extends JFrame
{

```

```

public Ecran()
{
    setTitle("première fenêtre en JAVA");
    setBounds(0,0,300,100);
    setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
    // création des trois boutons
    JButton b1,b2,b3;
    b1=new JButton("Rouge");
    b2=new JButton("Vert");
    b3=new JButton("Bleu");
    // création du conteneur intermédiaire
    JPanel pano;
    pano=new JPanel();
    // ajout des boutons sur le conteneur intermédiaire
    pano.add(b1);
    pano.add(b2);
    pano.add(b3);
    // ajout du conteneur intermédiaire sur le ContentPane
    getContentPane().add(pano);
    // création d'une instance d'une classe anonyme
    // chargée de gérer les événements
    addWindowListener(new WindowListener()
    // début de la définition de la classe
    {
        public void windowActivated(WindowEvent arg0)
        {
        }
        public void windowClosed(WindowEvent arg0)
        {
        }
        public void windowClosing(WindowEvent arg0)
        {
            System.exit(0);
        }
        public void windowDeactivated(WindowEvent arg0)
        {
        }
        public void windowDeiconified(WindowEvent arg0)
        {
        }
        public void windowIconified(WindowEvent arg0)
        {
        }
        public void windowOpened(WindowEvent arg0)
        {
        }
    } // fin de la définition de la classe
    ); // fin de l'appel de la méthode addWindowListener
} // fin du constructeur
}// fin de la classe Ecran

package fr.eni;

public class Principale {

    public static void main(String[] args)

    {
        // création de la fenêtre
        Ecran fenetre;
        fenetre=new Ecran();
        // affichage de la fenêtre
        fenetre.setVisible(true);
    }
}

```

Le seul reproche que l'on puisse faire à cette solution réside dans la relative complexité de la syntaxe. Les commentaires placés sur les différentes lignes fournissent une aide précieuse pour s'y retrouver dans les accolades et parenthèses.

Par contre, il y a un reproche que l'on peut faire de manière globale à toutes ces solutions. Pour une seule méthode réellement utile, nous sommes obligés d'en écrire sept.

Pour éviter ce code inutile, nous pouvons travailler avec une classe implémentant déjà la bonne interface et simplement redéfinir uniquement les méthodes nous intéressant.

Créer une classe "normale" héritant d'une classe XXXXAdapter

```
package fr.eni;

import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

public class Ecouteurfenetre extends WindowAdapter
{
    public void windowClosing(WindowEvent arg0)
    {
        System.exit(0);
    }
}

package fr.eni;

public class Principale {

    public static void main(String[] args)

    {
        // création de la fenêtre
        Ecran fenetre;
        fenetre=new Ecran();
        // création d'une instance de la classe chargée
        // de gérer les événements
        EcouteurFenetre ef;
        ef=new EcouteurFenetre();
        // référencement de cette instance de classe
        // comme écouteur d'événement pour la fenêtre
        fenetre.addWindowListener(ef);
        // affichage de la fenêtre
        fenetre.setVisible(true);
    }
}
```

Créer une classe interne héritant d'une classe XXXXAdapter

```
package fr.eni;

import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Ecran extends JFrame

{
    public Ecran()
    {
        setTitle("première fenêtre en JAVA");
        setBounds(0,0,300,100);
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        // création des trois boutons
        JButton b1,b2,b3;
```

```

        b1=new JButton("Rouge");
        b2=new JButton("Vert");
        b3=new JButton("Bleu");
        // création du conteneur intermédiaire
        JPanel pano;
        pano=new JPanel();
        // ajout des boutons sur le conteneur intermédiaire
        pano.add(b1);
        pano.add(b2);
        pano.add(b3);
        // ajout du conteneur intermédiaire sur le ContentPane
        getContentPane().add(pano);
        // création d'une instance de la classe chargée
        // de gérer les événements
        EcouteurFenetre ef;
        ef=new EcouteurFenetre();
        // comme écouteur d'événement pour la fenêtre

        addWindowListener(ef);
    }

    public class Ecouteurfenetre extends WindowAdapter
    {
        public void windowClosing(WindowEvent arg0)
        {
            System.exit(0);
        }
    }
}

package fr.eni;

public class Principale {

    public static void main(String[] args)

    {
        // création de la fenêtre
        Ecran fenetre;
        fenetre=new Ecran();
        // affichage de la fenêtre
        fenetre.setVisible(true);
    }
}

```

Créer une classe interne anonyme héritant d'une classe XXXXAdapter

```

package fr.eni;

import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Ecran extends JFrame

{
    public Ecran()
    {
        setTitle("première fenêtre en JAVA");
        setBounds(0,0,300,100);
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        // création des trois boutons
        JButton b1,b2,b3;
        b1=new JButton("Rouge");
        b2=new JButton("Vert");

```

```

b3=new JButton("Bleu");
// création du conteneur intermédiaire
JPanel pano;
pano=new JPanel();
// ajout des boutons sur le conteneur intermédiaire
pano.add(b1);
pano.add(b2);
pano.add(b3);
// ajout du conteneur intermédiaire sur le ContentPane
getContentPane().add(pano);
// création d'une instance d'une classe anonyme
// chargée de gérer les événements
addWindowListener(new WindowAdapter()
// début de la définition de la classe
{
    public void windowClosing(WindowEvent arg0)
    {
        System.exit(0);
    }
} // fin de la définition de la classe
); // fin de l'appel de la méthode addWindowListener
}// fin du constructeur
}// fin de la classe Ecran

package fr.eni;

public class Principale {

    public static void main(String[] args)

    {
        // création de la fenêtre
        Ecran fenetre;
        fenetre=new Ecran();
        // affichage de la fenêtre
        fenetre.setVisible(true);
    }
}

```

Cette solution est bien sûr la plus économique en nombre de lignes et c'est celle qui est utilisée par de nombreux outils de développement générant automatiquement du code. La relative complexité du code peut parfois être troublante lorsque l'on n'y est pas habitué.

Jusqu'à présent, nous avons une source d'événement et un écouteur pour cette source d'événement. Nous pouvons avoir dans certains cas la situation où nous avons plusieurs sources d'événements et souhaiter utiliser le même écouteur ou avoir une source d'événement et prévenir plusieurs écouteurs. La situation classique où nous avons plusieurs sources d'événements et un seul écouteur se produit lorsque nous fournissons à l'utilisateur plusieurs solutions pour lancer l'exécution d'une même action (menu et barre d'outils ou boutons).

Quel que soit le moyen utilisé pour lancer l'action, le code à exécuter reste le même.

Nous pouvons dans ce cas de figure utiliser le même écouteur pour les deux sources d'événements. Pour illustrer cela, nous allons ajouter un menu à l'application et faire en sorte que l'utilisation du menu ou d'un des boutons lance la même action en modifiant la couleur de fond correspondante au bouton ou au menu utilisé. Comme nous devons utiliser le même écouteur pour deux sources d'événements il est préférable d'utiliser une classe interne pour la création de l'écouteur. Voici ci-dessous le code correspondant.

```

package fr.eni;

import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JButton;
import javax.swing.JFrame;

```

```
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JPanel;

public class Ecran extends JFrame

{
    JPanel pano;
    public Ecran ()
    {
        setTitle("première fenêtre en JAVA");
        setBounds(0,0,300,100);
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        // création des trois boutons
        JButton btnRouge,btnVert,btnBleu;
        btnRouge=new JButton("Rouge");
        btnVert=new JButton("Vert");
        btnBleu=new JButton("Bleu");
        // création des trois écouteurs
        EcouteurRouge ecR;
        EcouteurVert ecV;
        EcouteurBleu ecB;
        ecR=new EcouteurRouge();
        ecV=new EcouteurVert();
        ecB=new EcouteurBleu();
        // association de l'écouteur à chaque bouton
        btnRouge.addActionListener(ecR);
        btnVert.addActionListener(ecV);
        btnBleu.addActionListener(ecB);
        // Création du menu
        JMenuBar barreMenu;
        barreMenu=new JMenuBar();
        JMenu mnuCouleurs;
        mnuCouleurs=new JMenu("Couleurs");
        barreMenu.add(mnuCouleurs);
        JMenuItem mnuRouge,mnuVert,mnuBleu;
        mnuRouge=new JMenuItem("Rouge");
        mnuVert=new JMenuItem("Vert");
        mnuBleu=new JMenuItem("Bleu");
        mnuCouleurs.add(mnuRouge);
        mnuCouleurs.add(mnuVert);
        mnuCouleurs.add(mnuBleu);
        // association de l'écouteur à chaque menu
        // (les mêmes que pour les boutons)
        mnuRouge.addActionListener(ecR);
        mnuVert.addActionListener(ecV);
        mnuBleu.addActionListener(ecB);
        // ajout du menu sur la fenêtre
        setJMenuBar(barreMenu);
        // création du conteneur intermédiaire
        pano=new JPanel();
        // ajout des boutons sur le conteneur intermédiaire
        pano.add(btnRouge);
        pano.add(btnVert);
        pano.add(btnBleu);
        // ajout du conteneur intermédiaire sur le ContentPane
        getContentPane().add(pano);
        // création d'une instance d'une classe anonyme
        // chargée de gérer les événements de la fenêtre
        addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent arg0)
            {
                System.exit(0);
            }
        });
    }
}

public class EcouteurRouge implements ActionListener
```

```

{
    public void actionPerformed(ActionEvent arg0)
    {
        pano.setBackground(Color.RED);
    }
}
public class EcouteurVert implements ActionListener
{
    public void actionPerformed(ActionEvent arg0)
    {
        pano.setBackground(Color.GREEN);
    }
}
public class EcouteurBleu implements ActionListener
{
    public void actionPerformed(ActionEvent arg0)
    {
        pano.setBackground(Color.BLUE);
    }
}
}

```

Dans ce code, nous avons nos trois classes écouteur qui sont très similaires. Avec une petite astuce, nous allons pouvoir simplifier le code pour n'avoir plus qu'une seule classe écouteur pour les trois boutons. La même méthode `actionPerformed` sera appelée suite à un clic sur n'importe lequel des boutons. Le choix de l'action à exécuter sera fait à l'intérieur de cette méthode. Pour cela, nous allons utiliser le paramètre `ActionEvent` fourni à cette méthode. Celui-ci permet d'obtenir une référence sur objet à l'origine de l'événement par l'intermédiaire de la méthode `getSource`. Le code simplifié est présenté ci-dessous :

```

package fr.eni;

import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JPanel;

public class Ecran extends JFrame
{
    JPanel pano;
    JButton btnRouge,btnVert,btnBleu;
    JMenuItem mnuRouge,mnuVert,mnuBleu;
    public Ecran ()
    {
        setTitle("première fenêtre en JAVA");
        setBounds(0,0,300,100);
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        // création des trois boutons
        btnRouge=new JButton("Rouge");
        btnVert=new JButton("Vert");
        btnBleu=new JButton("Bleu");
        // création des trois écouteurs
        EcouteurCouleur ec;
        ec=new EcouteurCouleur();
        // association de l'écouteur à chaque bouton
        btnRouge.addActionListener(ec);
        btnVert.addActionListener(ec);
        btnBleu.addActionListener(ec);
        // Création du menu

```

```

JMenuBar barreMenu;
barreMenu=new JMenuBar();
JMenu mnuCouleurs;
mnuCouleurs=new JMenu("Couleurs");
barreMenu.add(mnuCouleurs);

mnuRouge=new JMenuItem("Rouge");
mnuVert=new JMenuItem("Vert");
mnuBleu=new JMenuItem("Bleu");
mnuCouleurs.add(mnuRouge);
mnuCouleurs.add(mnuVert);
mnuCouleurs.add(mnuBleu);
// association de l'écouteur à chaque menu
// (le même que pour les boutons)
mnuRouge.addActionListener(ec);
mnuVert.addActionListener(ec);
mnuBleu.addActionListener(ec);
// ajout du menu sur la fenêtre
setJMenuBar(barreMenu);
// création du conteneur intermédiaire
pano=new JPanel();
// ajout des boutons sur le conteneur intermédiaire
pano.add(btnRouge);
pano.add(btnVert);
pano.add(btnBleu);
// ajout du conteneur intermédiaire sur le ContentPane
getContentPane().add(pano);
// création d'une instance d'une classe anonyme
// chargée de gérer les événements
addWindowListener(new WindowAdapter()
{
    public void windowClosing(WindowEvent arg0)
    {
        System.exit(0);
    }
});
}
}

public class EcouteurCouleur implements ActionListener
{
    public void actionPerformed(ActionEvent arg0)
    {
        if (arg0.getSource()==btnRouge | arg0.getSource()==mnuRouge)
        {
            pano.setBackground(Color.RED);
        }
        if (arg0.getSource()==btnVert | arg0.getSource()==mnuVert)
        {
            pano.setBackground(Color.GREEN);
        }
        if (arg0.getSource()==btnBleu | arg0.getSource()==mnuBleu)
        {
            pano.setBackground(Color.BLUE);
        }
    }
}
}

```

À noter que pour que cette solution puisse fonctionner, les objets source d'événements doivent être accessibles depuis la classe écouteur d'événements. La déclaration des boutons et des éléments de menu a donc été déplacée au niveau de la classe elle-même et non plus dans le constructeur comme c'était le cas dans la version précédente. Cette solution est possible uniquement si la classe écouteur est une classe interne. Dans le cas où la classe écouteur est parfaitement indépendante de la classe où sont créés les objets source d'événement, il faut revoir le code de la méthode actionPerformed. Le paramètre ActionEvent de la méthode actionPerformed nous fournit une autre solution pour contourner ce problème. Par l'intermédiaire de la méthode getActionCommand nous avons accès à une chaîne de caractères représentant l'objet à l'origine de l'événement. Par défaut cette chaîne de caractères correspond au libellé du composant ayant déclenché l'événement mais elle peut être modifiée par la

méthode `setActionCommand` de chaque composant. C'est d'ailleurs une pratique recommandée puisqu'elle nous permet d'avoir un code identique pour une application fonctionnant en plusieurs langues. Voici les modifications correspondantes.

```
package fr.eni;

import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JPanel;

public class Ecran extends JFrame

{
    JPanel pano;
    JButton btnRouge,btnVert,btnBleu;
    JMenuItem mnuRouge,mnuVert,mnuBleu;

    public Ecran ()
    {
        setTitle("première fenêtre en JAVA");
        setBounds(0,0,300,100);
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        // création des trois boutons
        btnRouge=new JButton("Rouge");
        btnRouge.setActionCommand("red");
        btnVert=new JButton("Vert");
        btnVert.setActionCommand("green");
        btnBleu=new JButton("Bleu");
        btnBleu.setActionCommand("blue");
        // création des trois écouteurs
        EcouteurCouleur ec;
        ec=new EcouteurCouleur();
        // association de l'écouteur à chaque bouton
        btnRouge.addActionListener(ec);
        btnVert.addActionListener(ec);
        btnBleu.addActionListener(ec);
        // Création du menu
        JMenuBar barreMenu;
        barreMenu=new JMenuBar();
        JMenu mnuCouleurs;
        mnuCouleurs=new JMenu("Couleurs");
        barreMenu.add(mnuCouleurs);
        mnuRouge=new JMenuItem("Rouge");
        mnuRouge.setActionCommand("red");
        mnuVert=new JMenuItem("Vert");
        mnuVert.setActionCommand("green");
        mnuBleu=new JMenuItem("Bleu");
        mnuBleu.setActionCommand("blue");
        mnuCouleurs.add(mnuRouge);
        mnuCouleurs.add(mnuVert);
        mnuCouleurs.add(mnuBleu);
        // association de l'écouteur à chaque menu
        // (le même que pour les boutons)
        mnuRouge.addActionListener(ec);
        mnuVert.addActionListener(ec);
        mnuBleu.addActionListener(ec);
        // ajout du menu sur la fenêtre
        setJMenuBar(barreMenu);
        // création du conteneur intermédiaire
        pano=new JPanel();
```

```

        // ajout des boutons sur le conteneur intermédiaire
        pano.add(btnRouge);
        pano.add(btnVert);
        pano.add(btnBleu);
        // ajout du conteneur intermédiaire sur le ContentPane
        getContentPane().add(pano);
        // création d'une instance d'une classe anonyme
        // chargée de gérer les événements
        addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent arg0)
            {
                System.exit(0);
            }
        });
    }
}

public class EcouteurCouleur implements ActionListener
{
    public void actionPerformed(ActionEvent arg0)
    {
        String commande;
        commande=arg0.getActionCommand();
        if (commande.equals("red"))
        {
            pano.setBackground(Color.RED);
        }
        if (commande.equals("green"))
        {
            pano.setBackground(Color.GREEN);
        }
        if (commande.equals("blue"))
        {
            pano.setBackground(Color.BLUE);
        }
    }
}
}

```

À noter que dans cette solution, la déclaration des boutons et des éléments de menu peut être réintégrée dans le constructeur puisque nous n'en avons plus besoin au niveau de la classe.

La dernière étape de notre marathon sur les événements va nous permettre d'avoir plusieurs écouteurs pour une même source d'événements et éventuellement supprimer un écouteur existant. Pour cela, nous allons créer une nouvelle classe écouteur qui va nous permettre d'afficher sur la console la date et heure de l'événement et l'objet à l'origine de l'événement.

```

package fr.eni;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.text.SimpleDateFormat;
import java.util.Date;

import javax.swing.AbstractButton;
import javax.swing.JButton;
import javax.swing.JMenuItem;

public class ConsoleLog implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        String message;
        SimpleDateFormat sdf;
        sdf=new SimpleDateFormat("dd/MM/yyyy hh:mm:ss");
        message=sdf.format(new Date());
        message=message + " clic sur le ";
    }
}

```

```

        if (e.getSource() instanceof JButton)
        {
            message=message+ "bouton ";
        }
        if (e.getSource() instanceof JMenuItem)
        {
            message=message+ "menu ";
        }
    message=message + ((AbstractButton)e.getSource()).getText();
    System.out.println(message);
}
}

```

Dans notre application, nous ajoutons ensuite une case à cocher permettant de choisir si les événements sont affichés sur la console. En fonction de l'état de cette case à cocher nous ajoutons, avec la méthode `addActionListener`, ou supprimons, avec la méthode `removeActionListener`, un écouteur aux boutons et menus. Ces deux méthodes attendent comme argument l'instance de l'écouteur à ajouter ou supprimer.

```

package fr.eni;

import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JPanel;

public class Ecran extends JFrame

{
    JPanel pano;
    JButton btnRouge,btnVert,btnBleu;
    JMenuItem mnuRouge,mnuVert,mnuBleu;
    ConsoleLog lg;
    public Ecran ()
    {
        setTitle("premiere fenetre en JAVA");
        setBounds(0,0,300,100);
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        // création des trois boutons

        btnRouge=new JButton("Rouge");
        btnRouge.setActionCommand("red");
        btnVert=new JButton("Vert");
        btnVert.setActionCommand("green");
        btnBleu=new JButton("Bleu");
        btnBleu.setActionCommand("blue");
        // création des trois écouteurs
        EcouteurCouleur ec;
        ec=new EcouteurCouleur();
        // association de l'écouteur à chaque bouton
        btnRouge.addActionListener(ec);
        btnVert.addActionListener(ec);
        btnBleu.addActionListener(ec);
        // création de la case à cocher
        JCheckBox chkLog;
        chkLog=new JCheckBox("log sur console");
        // ajout d'un écouteur à la case à cocher
        chkLog.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent arg0)

```

```

        {
            JCheckBox chk;
            chk=(JCheckBox)arg0.getSource();
            if (chk.isSelected())
            {
                // ajout d'un écouteur supplémentaire
                // aux boutons et menus
                lg=new ConsoleLog();
                btnBleu.addActionListener(lg);
                btnRouge.addActionListener(lg);
                btnVert.addActionListener(lg);
                mnuBleu.addActionListener(lg);
                mnuRouge.addActionListener(lg);
                mnuVert.addActionListener(lg);
            }
            else
            {
                // suppression de l'écouteur supplémentaire
                // des boutons et menus
                btnBleu.removeActionListener(lg);
                btnRouge.removeActionListener(lg);
                btnVert.removeActionListener(lg);
                mnuBleu.removeActionListener(lg);
                mnuRouge.removeActionListener(lg);
                mnuVert.removeActionListener(lg);
            }
        }
    });
    // Création du menu
    JMenuBar barreMenu;
    barreMenu=new JMenuBar();
    JMenu mnuCouleurs;
    mnuCouleurs=new JMenu("Couleurs");
    barreMenu.add(mnuCouleurs);
    mnuRouge=new JMenuItem("Rouge");
    mnuRouge.setActionCommand("red");
    mnuVert=new JMenuItem("Vert");
    mnuVert.setActionCommand("green");
    mnuBleu=new JMenuItem("Bleu");
    mnuBleu.setActionCommand("blue");
    mnuCouleurs.add(mnuRouge);
    mnuCouleurs.add(mnuVert);
    mnuCouleurs.add(mnuBleu);
    // association de l'écouteur à chaque menu
    // (le même que pour les boutons)
    mnuRouge.addActionListener(ec);
    mnuVert.addActionListener(ec);
    mnuBleu.addActionListener(ec);
    // ajout du menu sur la fenêtre
    setJMenuBar(barreMenu);
    // création du conteneur intermédiaire
    pano=new JPanel();
    // ajout des boutons sur le conteneur intermédiaire
    pano.add(btnRouge);
    pano.add(btnVert);
    pano.add(btnBleu);
    pano.add(chkLog);
    // ajout du conteneur intermédiaire sur le ContentPane
    getContentPane().add(pano);
    // création d'une instance d'une classe anonyme
    // chargée de gérer les événements
    addWindowListener(new WindowAdapter()
    {
        public void windowClosing(WindowEvent arg0)
        {
            System.exit(0);
        }
    });

```

```
        }
    public class EcouteurCouleur implements ActionListener
    {
        public void actionPerformed(ActionEvent arg0)
        {
            String commande;
            commande=arg0.getActionCommand();
            if (commande.equals("red"))
            {
                pano.setBackground(Color.RED);
            }
            if (commande.equals("green"))
            {
                pano.setBackground(Color.GREEN);
            }
            if (commande.equals("blue"))
            {
                pano.setBackground(Color.BLUE);
            }
        }
    }
}
```

À l'exécution et suivant l'état de la case à cocher, nous voyons apparaître les messages suivants sur la console.

27/11/2008 09:19:43 clic sur le bouton Rouge
27/11/2008 09:19:45 clic sur le bouton Vert
27/11/2008 09:19:47 clic sur le bouton Bleu
27/11/2008 09:19:51 clic sur le menu Rouge
27/11/2008 09:19:54 clic sur le menu Vert
27/11/2008 09:19:56 clic sur le menu Bleu

À noter que, indépendamment de l'affichage de ces messages, la couleur de la fenêtre change toujours lors de l'utilisation des boutons ou menus.

Tous ces principes de gestion sont identiques quels que soient les événements et les objets les déclenchant. Le seul problème que vous pourrez rencontrer au début c'est de savoir quels sont les événements disponibles pour un objet particulier. À ce niveau, seule la documentation Java va pouvoir vous venir en aide. Une petite astuce consiste à chercher dans la classe correspondant à l'objet concerné, les méthodes nommées `addXXXXListener` puis à partir de cette méthode remonter jusqu'à l'interface correspondante et découvrir, grâce aux méthodes définies dans l'interface, les différents événements possibles. L'analyse du type d'argument reçu dans les différentes méthodes vous permettra de savoir quelles informations supplémentaires sont disponibles pour un événement particulier. Par exemple, vous avez repéré la méthode `addMouseListener`.

addMouseListener

```
public void addMouseListener(MouseListener l)
```

Adds the specified mouse listener to receive mouse events from this component. If `Listener` is `null`, no exception is thrown and no action is performed.

Refer to [AWT Threading Issues](#) for details on AWT's threading model.

Parameters:

1 - the mouse listener

Since:

JDK 1.1

See Also:

```
MouseEvent, MouseListener, removeMouseListener(java.awt.event.MouseListener), getMouseListeners()
```

Vous pouvez déjà déterminer que cette méthode va vous permettre de travailler avec des événements liés à la souris. Pour avoir plus d'informations sur les différents événements vous devez aller voir l'interface `MouseListener`.

Method Summary

void	mouseClicked(MouseEvent e)	Invoked when the mouse button has been clicked (pressed and released) on a component.
void	mouseEntered(MouseEvent e)	Invoked when the mouse enters a component.
void	mouseExited(MouseEvent e)	Invoked when the mouse exits a component.
void	mousePressed(MouseEvent e)	Invoked when a mouse button has been pressed on a component.
void	mouseReleased(MouseEvent e)	Invoked when a mouse button has been released on a component.

La documentation de cette interface vous apprend que cinq méthodes sont prévues donc que cinq types d'événements sont disponibles et dans quelles circonstances ils sont déclenchés. Pour savoir quelles informations seront disponibles lors du déclenchement d'un de ces événements, vous poursuivez votre périple vers la classe correspondant au type des arguments reçus par ces méthodes.

Method Summary

int	getButton()	Returns which, if any, of the mouse buttons has changed state.
int	getClickCount()	Returns the number of mouse clicks associated with this event.
Point	getLocationOnScreen()	Returns the absolute x, y position of the event.
static String	getMouseModifiersText(int modifiers)	Returns a String describing the modifier keys and mouse buttons that were down during the event, such as "Shift", or "Ctrl+Shift".
Point	getPoint()	Returns the x,y position of the event relative to the source component.
int	getX()	Returns the horizontal x position of the event relative to the source component.
int	getXOnScreen()	Returns the absolute horizontal x position of the event.
int	getY()	Returns the vertical y position of the event relative to the source component.
int	getYOnScreen()	Returns the absolute vertical y position of the event.
boolean	isPopupTrigger()	Returns whether or not this mouse event is the popup menu trigger event for the platform.
String	 paramString()	Returns a parameter string identifying this event.
void	translatePoint(int x, int y)	Translates the event's coordinates to a new position by adding specified x (horizontal) and y (vertical) offsets.

Vous pensez certainement que tout ceci est fastidieux mais vous allez rapidement vous rendre compte que ce sont pratiquement toujours les mêmes événements dont on a besoin et vous allez très rapidement les connaître par cœur.

3. Aspect des composants

Une application Java étant susceptible d'être exécutée sur n'importe quelle plate-forme, l'aspect des composants doit s'adapter à cette plate-forme. Java fournit une solution avec le mécanisme du look and feel. Chaque composant graphique est en fait constitué de deux classes. L'une gère la partie fonctionnelle du composant et la deuxième concerne uniquement l'aspect visuel du composant. Cette dernière est adaptée en fonction de la plate-forme sur laquelle s'exécute l'application. La classe `UIManager` est responsable du fonctionnement de ce mécanisme. Lorsqu'un composant graphique est créé, le constructeur du composant utilise la classe `UIManager` pour obtenir une instance de la classe chargée de gérer l'aspect graphique du composant. La classe `UIManager` doit donc être informée du look and feel qu'elle doit utiliser.

L'indication du look and feel devant être utilisé peut être effectuée de trois façons différentes :

- Par programmation : vous devez utiliser la méthode `setLookAndFeel` de la classe `UIManager` en fournissant comme paramètre le nom complet de la classe implémentant le look and feel.

```
UIManager.setLookAndFeel("javax.swing.plaf.nimbus.NimbusLookAndFeel");
```

Il est recommandé d'effectuer la modification du look and feel dès le début de l'application avant même la création du premier composant graphique.

- Par un paramètre de la ligne de commande utilisée pour lancer l'application.

```
java -Dswing.defaultlaf=javax.swing.plaf.nimbus.NimbusLookAndFeel applicationDemo
```

- Par modification d'un fichier de configuration.

Vous pouvez également utiliser un fichier de configuration swing.properties pour définir le look and feel par défaut qu'appliquera la machine virtuelle Java lors du lancement d'une application graphique. Ce fichier doit être placé dans le répertoire lib de la machine virtuelle. Il doit contenir la ligne suivante :

```
swing.defaultlaf=javax.swing.plaf.nimbus.NimbusLookAndFeel
```

 Les exemples ci-dessus utilisent le look and feel nimbus disponible depuis la version 7 de Java. Celui-ci se démarque des autres look and feel par son mode de gestion graphique. Les versions précédentes utilisaient des images bitmap pour la représentation des composants. Cette nouvelle version utilise un mode graphique vectoriel qui procure un rendu plus précis que le mode bitmap.

4. Le positionnement des composants

Si vous avez déjà utilisé un autre langage de programmation permettant de développer des interfaces graphiques, une chose doit vous paraître étrange dans les exemples de code que nous avons utilisés.

- Nulle part, nous n'avons indiqué la taille et la position des composants utilisés.
- Si vous redimensionnez la fenêtre de l'application, les composants changent de place.

Et pourtant aucune ligne de code n'est prévue pour effectuer ce traitement. Ce petit miracle est lié à un concept très pratique de Java : les gestionnaires de mise en page (layout manager). En fait lorsque vous confiez des composants à un conteneur, celui-ci délègue à son gestionnaire de mise en page le soin d'organiser la disposition des composants sur sa surface.

De nombreux types de gestionnaire de mise en page sont disponibles, chacun d'eux ayant une stratégie différente pour le placement des composants. Chaque type de conteneur dispose d'un gestionnaire de mise en page par défaut, différent en fonction du type de conteneur. Si ce gestionnaire par défaut ne vous convient pas, vous êtes libre de le remplacer par un autre en appelant la méthode `setLayout` et en fournissant à cette méthode le nouveau gestionnaire à utiliser par ce conteneur.

Pour pouvoir travailler et organiser la disposition des composants, le gestionnaire de mise en page doit connaître la taille de chaque composant. La meilleure façon d'obtenir cette information est de s'adresser au composant lui-même. Pour cela, le gestionnaire de mise en page questionne chaque composant en appelant sa méthode `getPreferredSize`. Cette méthode calcule la taille du composant en fonction de son contenu, par exemple la longueur du libellé pour un bouton. Pour court-circuiter ce calcul, vous pouvez fixer une taille par défaut à chaque composant avec la méthode `setPreferredSize`.

La réussite de la conception d'une interface utilisateur nécessite donc de bien connaître comment fonctionnent les différents gestionnaires de mise en page. Nous allons donc étudier les caractéristiques des plus utilisés d'entre eux.

a. FlowLayout

Ce gestionnaire de mise en page est certainement le plus simple à utiliser. Il est associé par défaut à un composant JPanel. Sa stratégie de placement des composants consiste à les placer les uns à la suite des autres sur une ligne jusqu'à ce qu'il n'y ait plus de place sur cette ligne. Après le remplissage de la

première ligne, les composants suivants sont placés sur une nouvelle ligne et ainsi de suite. C'est l'ordre d'ajout des composants sur le conteneur qui détermine leurs positions sur les différentes lignes. Chaque composant est séparé horizontalement de son voisin par un espace de cinq pixels par défaut et chaque ligne de composants est séparée de sa voisine par un espace de cinq pixels par défaut également. Lorsque ce gestionnaire place les composants, il les aligne par défaut sur le centre du conteneur. Tous ces paramètres peuvent être modifiés pour chaque gestionnaire de mise en page FlowLayout. Vous pouvez le faire dès la création du FlowLayout en indiquant dans le constructeur les informations correspondantes. Trois constructeurs sont disponibles pour cette classe. Le premier n'attend aucun argument et crée un FlowLayout avec les caractéristiques par défaut décrites ci-dessus. Le deuxième attend comme argument une des constantes suivantes permettant de spécifier le type d'alignement utilisé.

- `FlowLayout.CENTER` : chaque ligne de composant est centrée dans le conteneur (valeur par défaut).
- `FlowLayout.LEFT` : chaque ligne de composant est alignée sur la gauche du conteneur.
- `FlowLayout.RIGHT` : chaque ligne de composant est alignée sur la droite du conteneur.

Le dernier constructeur disponible attend comme arguments deux entiers en plus de la constante indiquant l'alignement. Ces deux entiers indiquent l'espacement horizontal et vertical entre les composants. La ligne de code suivante crée un `FlowLayout` qui aligne les lignes de composants sur le centre du conteneur, laisse un espace horizontal de cinquante pixels et un espace vertical de vingt pixels entre les composants.

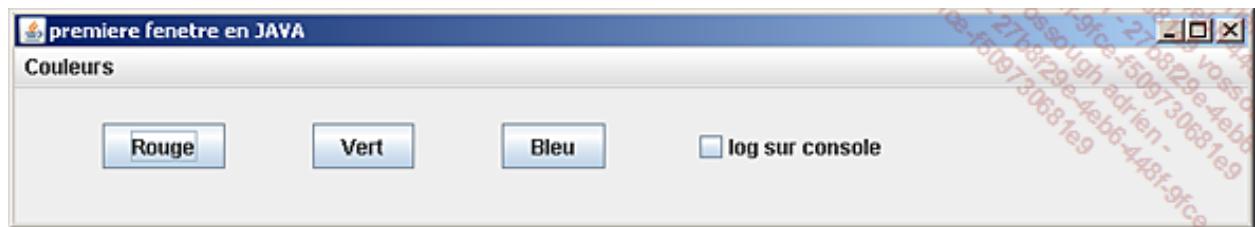
```
fl=new FlowLayout(FlowLayout.LEFT,50,20);
```

Si vous ne créez pas vous-même de `FlowLayout` mais que vous utilisez celui fourni par défaut avec un `JPanel`, vous pouvez intervenir sur ces paramètres de fonctionnement avec les méthodes suivantes :

- `setAlignment` : pour spécifier l'alignement des composants.
- `setHgap` : pour spécifier l'espacement horizontal entre les composants.
- `setVgap` : pour spécifier l'espacement vertical entre les composants.

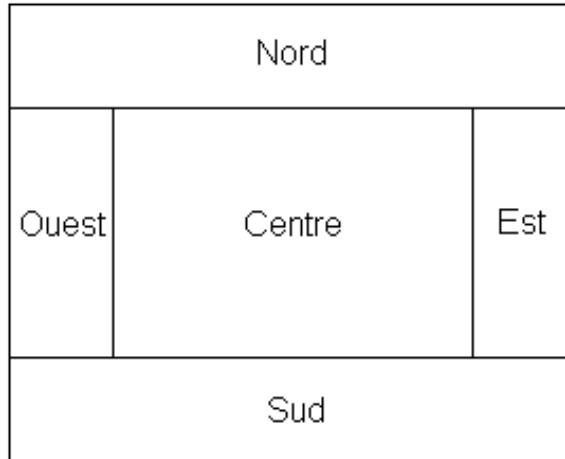
Il est dans ce cas nécessaire d'obtenir une référence sur le `FlowLayout` associé au `JPanel` en utilisant la méthode `getLayout` de celui-ci. Une opération de transposition est dans ce cas obligatoire pour pouvoir utiliser les méthodes de la classe `FlowLayout` sur la référence obtenue.

```
((FlowLayout)pano.getLayout()).setAlignment(FlowLayout.LEFT);
((FlowLayout)pano.getLayout()).setHgap(50);
((FlowLayout)pano.getLayout()).setVgap(20);
```

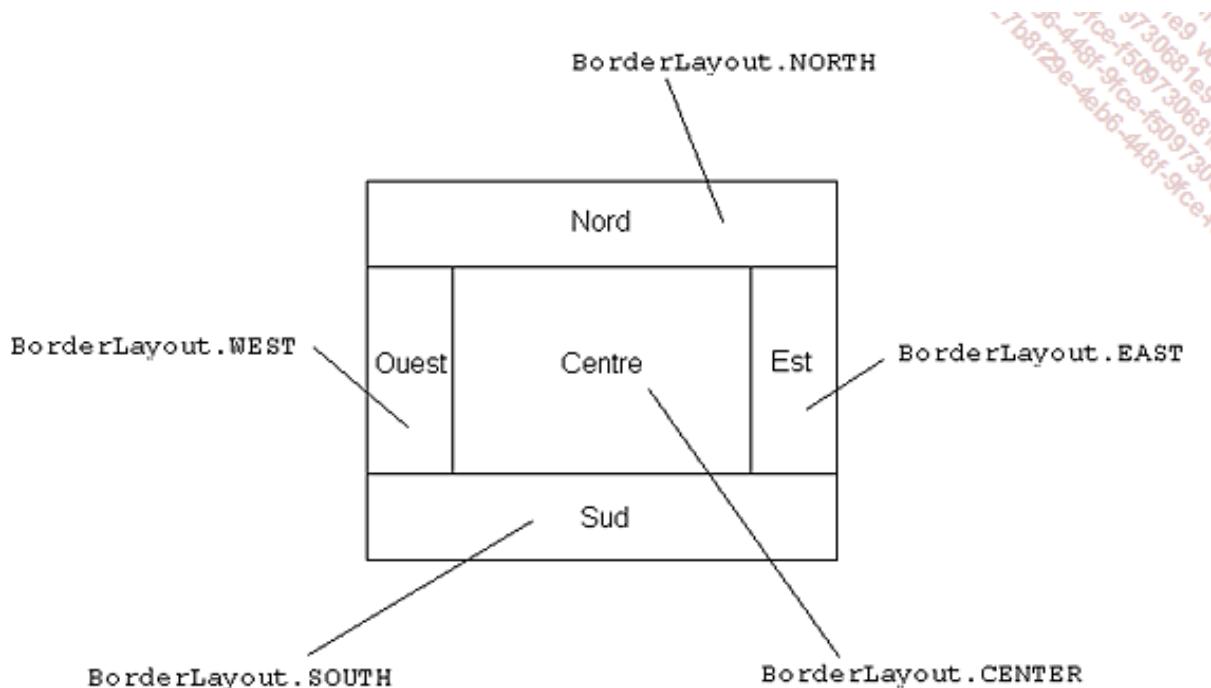


b. BorderLayout

Ce gestionnaire de mise en page découpe la surface qui lui est confiée en cinq zones suivant le schéma ci-dessous.



Lorsqu'un composant est confié à un `BorderLayout`, il convient d'indiquer dans quelle zone celui-ci doit être placé. Cette indication est fournie comme deuxième argument de la méthode `add`, le premier argument étant toujours le composant à insérer sur le conteneur. Les constantes suivantes sont disponibles pour identifier une zone du `BorderLayout`.



Si aucune information n'est indiquée pour identifier une zone lors de l'ajout d'un composant celui-ci est systématiquement placé dans la zone Centre.

Le `BorderLayout` n'aime pas le vide et de ce fait il redimensionne automatiquement tous les composants que vous lui confiez pour occuper tout l'espace disponible. Lorsque le conteneur est redimensionné, les composants placés en bordure ne changent pas de largeur pour les zones Ouest et Est, de hauteur pour les zones Nord et Sud. L'espace est donc gagné ou perdu par la zone Centre. Chaque zone du `BorderLayout` ne peut contenir qu'un seul composant. Si plusieurs composants sont malgré tout ajoutés à une même zone d'un `BorderLayout`, seul le dernier ajouté sera pris en charge par le `BorderLayout`, les autres ne seront pas visibles.

Du fait de cette limitation, le `BorderLayout` est généralement utilisé pour positionner des conteneurs les uns par rapport aux autres plutôt que des composants isolés.

Le `BorderLayout` est le gestionnaire de mise en page par défaut de l'élément `ContentPane` d'une `JFrame`. L'utilisation classique consiste à placer dans la zone Nord la ou les barres d'outils, dans la zone Sud la barre d'état de l'application et dans la zone Centre le document sur lequel l'utilisateur doit travailler. Comme pour le `FlowLayout` il est possible d'indiquer

au BorderLayout qu'il doit ménager un espace vertical ou horizontal entre ces différentes zones. Ceci peut être fait à la création du BorderLayout en indiquant l'espace horizontal et vertical dans l'appel du constructeur ou, si vous utilisez un BorderLayout existant, en appelant les méthodes setHgap et setVgap du BorderLayout. Pour illustrer tout cela, nous allons remanier l'interface de notre application en plaçant les boutons dans la zone Nord, la case à cocher dans la zone Sud et en modifiant la couleur de la zone Centre lors du clic sur les boutons ou menus.

```
package fr.eni;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JPanel;

public class Ecran9 extends JFrame

{
    JPanel panoBoutons;
    JPanel panoChk;
    JPanel panoCouleur;
    JButton btnRouge,btnVert,btnBleu;
    JMenuItem mnuRouge,mnuVert,mnuBleu;
    ConsoleLog lg;
    public Ecran9()
    {
        setTitle("première fenêtre en JAVA");
        setBounds(0,0,300,100);
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        // création des trois boutons

        btnRouge=new JButton("Rouge");
        btnRouge.setActionCommand("red");
        btnVert=new JButton("Vert");
        btnVert.setActionCommand("green");
        btnBleu=new JButton("Bleu");
        btnBleu.setActionCommand("blue");
        // création des trois écouteurs
        EcouteurCouleur ec;
        ec=new EcouteurCouleur();
        // association de l'écouteur à chaque bouton
        btnRouge.addActionListener(ec);
        btnVert.addActionListener(ec);
        btnBleu.addActionListener(ec);
        // création de la case à cocher
        JCheckBox chkLog;
        chkLog=new JCheckBox("log sur console");
        // ajout d'un écouteur à la case à cocher
        chkLog.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent arg0)
            {
                JCheckBox chk;
                chk=(JCheckBox)arg0.getSource();
                if (chk.isSelected())
                {
                    // ajout d'un écouteur supplémentaire
                    // aux boutons et menus
                    lg=new ConsoleLog();
                }
            }
        });
        panoBoutons.add(btnRouge);
        panoBoutons.add(btnVert);
        panoBoutons.add(btnBleu);
        panoBoutons.add(chkLog);
        panoBoutons.setLayout(null);
        panoBoutons.setBackground(Color.pink);
        panoBoutons.setBounds(10,10,280,100);
        add(panoBoutons, "North");
        panoChk.add(chkLog);
        panoChk.setLayout(null);
        panoChk.setBackground(Color.pink);
        panoChk.setBounds(10,110,280,100);
        add(panoChk, "South");
        panoCouleur.add(mnuRouge);
        panoCouleur.add(mnuVert);
        panoCouleur.add(mnuBleu);
        panoCouleur.setLayout(null);
        panoCouleur.setBackground(Color.pink);
        panoCouleur.setBounds(10,120,280,100);
        add(panoCouleur, "Center");
    }
}
```

```

        btnBleu.addActionListener(lg);
        btnRouge.addActionListener(lg);
        btnVert.addActionListener(lg);
        mnuBleu.addActionListener(lg);
        mnuRouge.addActionListener(lg);
        mnuVert.addActionListener(lg);
    }
    else
    {
        // suppression de l'écouteur supplémentaire
        // des boutons et menus
        btnBleu.removeActionListener(lg);
        btnRouge.removeActionListener(lg);
        btnVert.removeActionListener(lg);
        mnuBleu.removeActionListener(lg);
        mnuRouge.removeActionListener(lg);
        mnuVert.removeActionListener(lg);
    }
}

});

// Création du menu
JMenuBar barreMenu;
barreMenu=new JMenuBar();
JMenu mnuCouleurs;
mnuCouleurs=new JMenu("Couleurs");
barreMenu.add(mnuCouleurs);
mnuRouge=new JMenuItem("Rouge");
mnuRouge.setActionCommand("red");
mnuVert=new JMenuItem("Vert");
mnuVert.setActionCommand("green");
mnuBleu=new JMenuItem("Bleu");
mnuBleu.setActionCommand("blue");
mnuCouleurs.add(mnuRouge);
mnuCouleurs.add(mnuVert);
mnuCouleurs.add(mnuBleu);
// association de l'écouteur à chaque menu
// (le même que pour les boutons)
mnuRouge.addActionListener(ec);
mnuVert.addActionListener(ec);
mnuBleu.addActionListener(ec);
// ajout du menu sur la fenêtre
setJMenuBar(barreMenu);
// création du conteneur intermédiaire
panoBoutons=new JPanel();
// ajout des boutons sur le conteneur intermédiaire
panoBoutons.add(btnRouge);
panoBoutons.add(btnVert);
panoBoutons.add(btnBleu);
// ajout du conteneur intermédiaire sur le ContentPane
// zone nord
getContentPane().add(panoBoutons,BorderLayout.NORTH);
// création du conteneur pour la case à cocher
panoChk=new JPanel();
panoChk.add(chkLog);
// ajout du conteneur dans la zone sud
getContentPane().add(panoChk,BorderLayout.SOUTH);
// création du conteneur pour affichage de la couleur
panoCouleur=new JPanel();
// ajout du conteneur dans la zone centre
getContentPane().add(panoCouleur,BorderLayout.CENTER);
// création d'une instance d'une classe anonyme
// chargée de gérer les événements
addWindowListener(new WindowAdapter()
{
    public void windowClosing(WindowEvent arg0)
    {
        System.exit(0);
    }
})

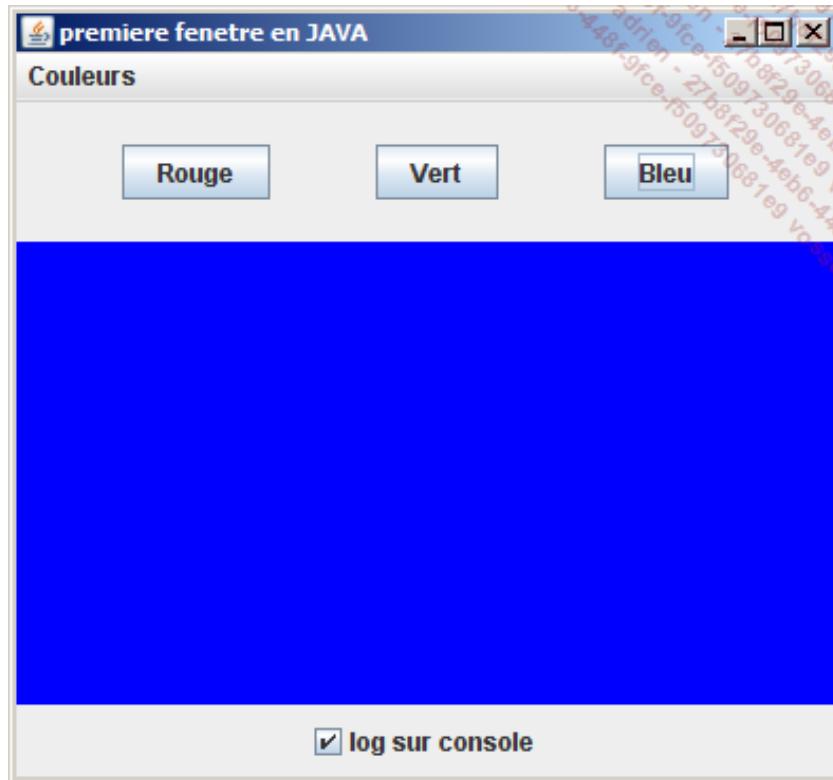
```

```

        );
        ((FlowLayout)panoBoutons.getLayout()).setAlignment
(FlowLayout.LEFT);
        ((FlowLayout)panoBoutons.getLayout()).setHgap(50);
        ((FlowLayout)panoBoutons.getLayout()).setVgap(20);
    }
public class EcouteurCouleur implements ActionListener
{
    public void actionPerformed(ActionEvent arg0)
    {
        String commande;
        commande=arg0.getActionCommand();
        if (commande.equals("red"))
        {
            panoCouleur.setBackground(Color.RED);
        }
        if (commande.equals("green"))
        {
            panoCouleur.setBackground(Color.GREEN);
        }
        if (commande.equals("blue"))
        {
            panoCouleur.setBackground(Color.BLUE);
        }
    }
}
}

```

Notre fenêtre présente maintenant la disposition suivante :



c. GridLayout

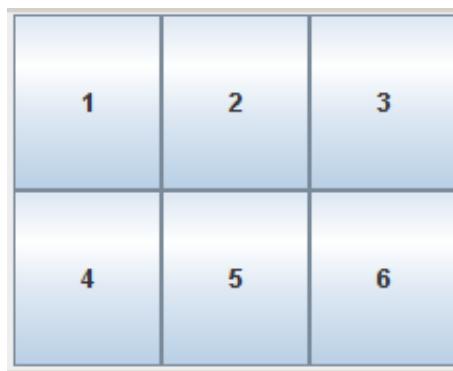
Ce gestionnaire de mise en page découpe la surface qui lui est confiée sous forme d'une grille invisible. Le nombre de lignes et de colonnes doit être indiqué dans l'appel du constructeur. Lorsque ce gestionnaire place les composants qu'on lui confie, il commence par remplir toutes les cases de la première ligne puis passe à la ligne suivante et ainsi de suite. Toutes les cases ont une taille identique et leurs contenus sont donc redimensionnés pour occuper tout l'espace disponible de chaque case. Par défaut les composants sont placés côté à côté. Un espace vertical et horizontal peut être inséré entre chaque composant en

spécifiant deux arguments supplémentaires lors de l'appel du constructeur. Ces deux arguments indiquent l'espacement horizontal et l'espacement vertical. Ces espacements peuvent également être modifiés avec les méthodes `setHgap` et `setVgap`.

Si vous insérez plus de composants qu'il y a de cases dans le `GridLayout`, celui-ci crée des cases supplémentaires en augmentant le nombre de colonnes et en respectant le nombre de lignes d'origine. Par exemple avec le code suivant, nous prévoyons une grille de 2x2 cases mais nous ajoutons six composants.

```
JPanel grille;
grille=new JPanel();
GridLayout grl;
grl=new GridLayout(2,2);
grille.setLayout(grl);
grille.add(new JButton("1"));
grille.add(new JButton("2"));
grille.add(new JButton("3"));
grille.add(new JButton("4"));
grille.add(new JButton("5"));
grille.add(new JButton("6"));
pano.add(grille);
```

Les composants sont disposés selon le schéma suivant :



Si vous souhaitez que le `GridLayout` respecte le nombre de colonnes, vous devez indiquer que le nombre de lignes vous est indifférent en spécifiant la valeur 0 pour cet argument dans le constructeur. Dans ce cas, le gestionnaire de mise en page ajoute des lignes supplémentaires pour les composants en surplus.

```
JPanel grille;
grille=new JPanel();
GridLayout grl;
grl=new GridLayout(0,2);
grille.setLayout(grl);
grille.add(new JButton("1"));
grille.add(new JButton("2"));
grille.add(new JButton("3"));
grille.add(new JButton("4"));
grille.add(new JButton("5"));
grille.add(new JButton("6"));
pano.add(grille);
```

1	2
3	4
5	6

d. BoxLayout

Ce gestionnaire est utilisé lorsque l'on a besoin de placer des composants sur une ligne ou une colonne. Bien que l'on puisse obtenir le même résultat avec un `GridLayout` d'une seule ligne ou d'une seule colonne, ce gestionnaire procure plus de fonctionnalités. Le prix à payer pour ces fonctionnalités supplémentaires est une relative complexité d'utilisation. Le constructeur de la classe `BoxLayout` attend comme premier argument le conteneur dont il va gérer le contenu et comme deuxième argument une constante indiquant si ce conteneur gère une ligne (`BoxLayout.X_AXIS`) ou une colonne (`BoxLayout.Y_AXIS`).

Bien que l'on fournit une référence sur le conteneur lors de la création de gestionnaire de mise en page, il est tout de même obligatoire d'associer le gestionnaire au conteneur avec la méthode `setLayout` comme nous l'avons déjà fait avec les autres gestionnaires.

Étudions maintenant la stratégie de fonctionnement de ce gestionnaire. Nous allons prendre le cas d'un gestionnaire horizontal.

- Le gestionnaire recherche le composant le plus haut.
- Il tente d'agrandir les autres composants jusqu'à cette hauteur.
- Si un composant ne peut pas atteindre cette hauteur, il est aligné verticalement sur les autres composants en fonction de la valeur renvoyée par la méthode `getAlignmentY` du composant.
 - 0 pour un alignement sur le haut des autres composants.
 - 1 pour un alignement sur le bas des autres composants.
 - 0.5 pour un centrage sur les autres composants.
- Les largeurs préférées de chaque composant sont additionnées.
- Si la somme est inférieure à la largeur du conteneur, les composants sont agrandis jusqu'à leur largeur maximale.
- Si la somme est supérieure à la largeur du conteneur, les composants sont réduits jusqu'à leur largeur minimale. Si ce n'est pas suffisant, les derniers composants ne seront pas affichés.
- Les composants sont ensuite placés sur le conteneur de gauche à droite sans espace de séparation.

Le fonctionnement est tout à fait similaire pour un gestionnaire vertical.

```
JPanel ligne;
ligne=new JPanel();
BoxLayout bl;
bl=new BoxLayout(ligne,BoxLayout.X_AXIS);
ligne.setLayout(bl);
JButton b1,b2,b3,b4,b5;
// création d'un bouton avec alignement sur le haut
b1=new JButton("petit");
b1.setAlignmentY(0);
ligne.add(b1);
```

```
// création d'un bouton avec alignement sur le bas
b2=new JButton("    moyen    ");
b2.setAlignmentY(1);
ligne.add(b2);
// utilisation de html pour le libellé du bouton
b3=new JButton("<html>très<BR>haut</html>");
ligne.add(b3);
b4=new JButton("      très      large      ");
ligne.add(b4);
b5=new JButton("<html>très haut<br>et<br>très large</html>");
ligne.add(b5);
getContentPane().add(ligne);
```

Ce code nous donne l'interface utilisateur suivante.



Par défaut il n'y a pas d'espace entre les composants gérés par un `BoxLayout` et l'aspect résultant n'est très aéré. Pour pouvoir ajouter un espace entre les composants, vous devez insérer des composants de réservation d'espace. Trois types sont disponibles :

Strut : cet élément est utilisé pour ajouter un espace fixe entre deux composants. Un `Strut` peut être horizontal ou vertical suivant le type de `BoxLayout` auquel il est destiné. Ces deux types d'éléments sont créés par les méthodes statiques `createVerticalStrut` et `createHorizontalStrut` de la classe `Box`. Elles attendent toutes les deux comme argument le nombre de pixels pour la largeur ou la hauteur.

```
// création d'un bouton avec alignement sur le haut
b1=new JButton("petit");
b1.setAlignmentY(0);
ligne.add(b1);
ligne.add(Box.createHorizontalStrut(10));
// création d'un bouton avec alignement sur le bas
b2=new JButton("    moyen    ");
b2.setAlignmentY(1);
ligne.add(b2);
```

RigidArea : cet élément a un fonctionnement similaire à celui d'un `Strut` en séparant les composants les uns des autres. Il force également une hauteur minimale pour le conteneur. Si un composant contenu dans le conteneur à une hauteur supérieure à celle-ci, c'est dans ce cas lui qui impose la hauteur du conteneur. Dans le cas contraire, c'est le composant le plus haut qui s'impose. Comme pour un `Strut`, c'est la méthode statique `createRigidArea` de la classe `Box` qui nous permet la création d'un `RigidArea`. Cette méthode attend comme argument un objet `Dimension` spécifiant l'espacement entre les composants et la hauteur minimale du conteneur.

```
// création d'un bouton avec alignement sur le haut
b1=new JButton("petit");
b1.setAlignmentY(0);
ligne.add(b1);
ligne.add(Box.createRigidArea(new Dimension(50,150)));
// création d'un bouton avec alignement sur le bas
b2=new JButton("    moyen    ");
b2.setAlignmentY(1);
ligne.add(b2);
```

Glue : le but de cet élément est toujours de séparer les composants mais cette fois l'espace entre les

composants n'a pas une taille fixe. On peut comparer un Glue à un ressort que l'on place entre deux composants et qui les éloigne toujours le plus possible. Pour avoir un fonctionnement correct du Glue, les composants doivent avoir une taille fixe ou une taille maximale fixée. Comme les autres éléments de séparation, il est créé à partir de la méthode statique `createGlue` de la classe `Box`.

```
JPanel ligne;
ligne=new JPanel();
BoxLayout bl;
bl=new BoxLayout(ligne,BoxLayout.X_AXIS);
ligne.setLayout(bl);
JButton b1,b2;
// création d'un bouton avec alignement sur le haut
b1=new JButton("petit");
b1.setAlignmentY(0);
b1.setMaximumSize(new Dimension(50,20));
ligne.add(b1);
ligne.add(Box.createGlue());
// création d'un bouton avec alignement sur le bas
b2=new JButton(" moyen ");
b2.setAlignmentY(1);
b2.setMaximumSize(new Dimension(50,20));
ligne.add(b2);
```

e. GridBagLayout

Vous pouvez considérer le `GridBagLayout` comme le "super" gestionnaire de mise en page tant du point de vue des performances que de la complexité d'utilisation. Il fonctionne à la base comme un `GridLayout` en plaçant les composants à l'intérieur d'une grille mais la comparaison s'arrête là. Dans un `GridBagLayout` les lignes et les colonnes ont des tailles variables, les cellules adjacentes peuvent être fusionnées pour accueillir les composants les plus grands. Les composants n'occupent pas obligatoirement toute la surface de leurs cellules et leurs positions à l'intérieur de la cellule peuvent être modifiées. La spécificité de ce gestionnaire se situe principalement dans la méthode d'ajout des composants. Cette méthode attend bien sûr comme argument le composant à ajouter mais également un objet `GridBagConstraints` indiquant la façon de positionner le composant dans le conteneur. C'est donc les caractéristiques de cet objet qui vont permettre le positionnement du composant par le `GridBagLayout`. Regardons donc les principales caractéristiques de cet objet :

- **gridx** : colonne du coin supérieur gauche du composant.
- **gridy** : ligne du coin supérieur gauche du composant.

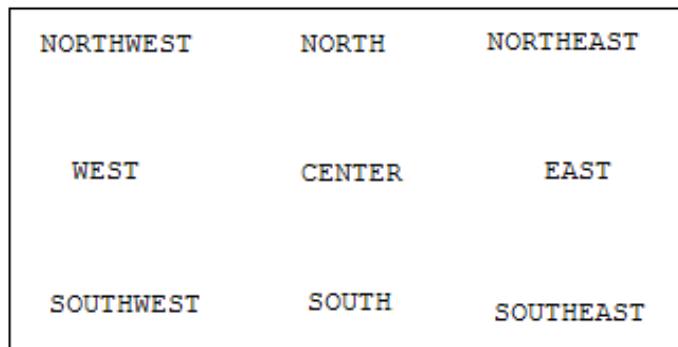
Pour ces deux propriétés la numérotation commence à zéro.

- **gridwidth** : nombre de colonnes occupées par le composant.
- **gridheight** : nombre de lignes occupées par le composant.
- **weightx** : indique comment est réparti l'espace supplémentaire disponible en largeur lorsque le conteneur est redimensionné. La répartition est faite au prorata de cette valeur. Si cette valeur est égale à zéro le composant n'est pas redimensionné. Si cette valeur est identique pour tous les composants l'espace est réparti équitablement.
- **weighty** : cette propriété joue le même rôle que la précédente mais sur l'axe vertical.
- **fill** : cette propriété est utilisée lorsque la zone allouée à un composant est supérieure à sa taille. Elle détermine si le composant est redimensionné et comment.

Les constantes suivantes sont possibles :

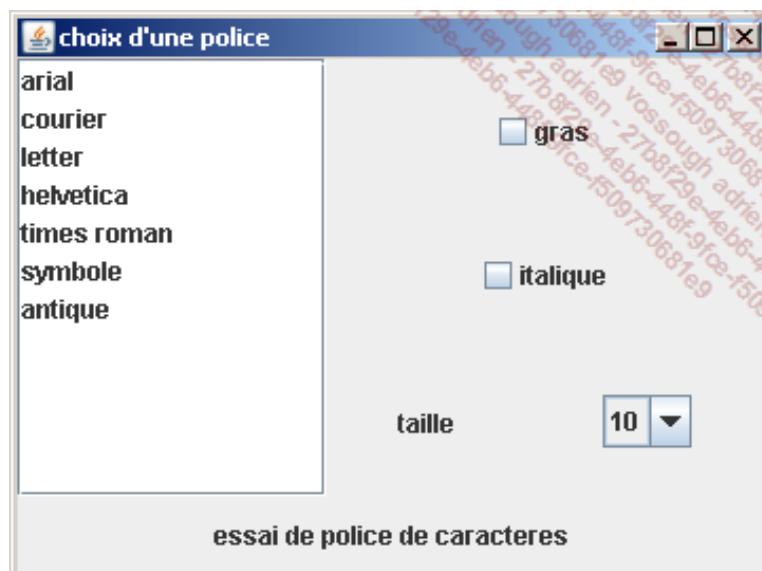
- **NONE** : le composant n'est pas redimensionné.
- **HORIZONTAL** : le composant est redimensionné en largeur et sa hauteur reste inchangée.
- **VERTICAL** : le composant est redimensionné en hauteur et sa largeur reste inchangée.
- **BOTH** : le composant est redimensionné en largeur et en hauteur pour remplir complètement la surface disponible.

- **anchor** : indique comment est positionné le composant dans l'espace disponible s'il n'est pas redimensionné. Les constantes suivantes sont disponibles.



Un même objet `GridBagConstraints` peut être utilisé pour placer plusieurs composants sur un `GridLayout`. Il faut bien penser dans ce cas à initialiser correctement les différents champs lors de l'ajout de chaque composant.

À titre d'exemple voici l'interface d'une application simple et le code correspondant.



```
package fr.eni;

import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import javax.swing.JCheckBox;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JList;
import javax.swing.JPanel;
import javax.swing.JScrollPane;

public class Ecran extends JFrame

{
    JPanel pano;
    JCheckBox chkGras,chkItalique;
    JLabel lblTaille,lblExemple;
    JComboBox cboTaille;
    JList lstPolice;
    JScrollPane defilPolice;

    public Ecran()
    {
```

```
setTitle("choix d'une police");
setBounds(0,0,300,100);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
// création des composants

pano=new JPanel();
chkGras=new JCheckBox("gras");
chkItalique=new JCheckBox("italique");
lblTaille=new JLabel("taille");
lblExemple=new JLabel("essai de police de caractères");
cboTaille=new JComboBox(new String[]
{"10","12","14","16","18","20"});
lstPolices=new JList(new String[]{ "arial", "courier",
"letter", "helvetica", "times roman", "symbole", "antique" });
defilPolices=new JScrollPane(lstPolices);

GridBagLayout gbl;
gbl=new GridBagLayout();
pano.setLayout(gbl);

GridBagConstraints gbc;
gbc=new GridBagConstraints();
// position dans la case 0,0
gbc.gridx=0;
// sur une colonne de largeur
gbc.gridwidth=1;
// et sur trois lignes en hauteur
gbc.gridheight=3;
// pondération en cas d'agrandissement du conteneur
gbc.weightx=100;
gbc.weighty=100;
// le composant est redimensionné pour occuper
// tout l'espace disponible dans son conteneur
gbc.fill=GridBagConstraints.BOTH;
pano.add(defilPolices,gbc);
// position dans la case 1,0
gbc.gridx=1;
gbc.gridy=0;
// sur deux colonnes de largeur
gbc.gridwidth=2;
// et sur une ligne en hauteur
gbc.gridheight=1;
// pondération en cas d'agrandissement du conteneur
gbc.weightx=100;
gbc.weighty=100;
// le composant n'est pas redimensionné pour occuper
// tout l'espace disponible dans son conteneur
gbc.fill=GridBagConstraints.NONE;
pano.add(chkGras,gbc);
// position dans la case 1,1
gbc.gridx=1;
gbc.gridy=1;
// sur deux colonnes de largeur
gbc.gridwidth=2;
// et sur une ligne en hauteur
gbc.gridheight=1;
// pondération en cas d'agrandissement du conteneur
gbc.weightx=100;
gbc.weighty=100;
pano.add(chkItalique,gbc);
// position dans la case 1,2
gbc.gridx=1;
gbc.gridy=2;
// sur une colonne de largeur
gbc.gridwidth=1;
// et sur une ligne en hauteur
gbc.gridheight=1;
// pondération en cas d'agrandissement du conteneur
gbc.weightx=100;
gbc.weighty=100;
```

```

pano.add(lblTaille,gbc);
// position dans la case 2,2
gbc.gridx=2;
gbc.gridy=2;
// sur une colonne de largeur
gbc.gridwidth=1;
// et sur une ligne en hauteur
gbc.gridheight=1;
// pondération en cas d'agrandissement du conteneur
gbc.weightx=100;
gbc.weighty=100;
pano.add(cboTaille,gbc);
// position dans la case 0,3
gbc.gridx=0;
gbc.gridy=3;
// sur trois colonnes de largeur
gbc.gridwidth=3;
// et sur une ligne en hauteur
gbc.gridheight=1;
// pondération en cas d'agrandissement du conteneur
gbc.weightx=100;
gbc.weighty=100;
pano.add(lblExemple,gbc); x
getContentPane().add(pano);
}
}

```

f. Sans gestionnaire de mise en page

Si vous n'êtes pas convaincu de l'utilité d'un gestionnaire de mise en page, vous avez la possibilité de gérer vous-même la taille et position des composants de l'interface utilisateur. Vous devez d'abord indiquer que vous renoncez à utiliser un gestionnaire de mise en forme en appelant la méthode `setLayout` du conteneur et en lui passant la valeur `null`. Vous pouvez ensuite ajouter les composants au conteneur avec la méthode `add` de celui-ci. Et finalement vous devez positionner et dimensionner les composants. Ces deux opérations peuvent être effectuées en deux étapes avec les méthodes `setLocation` et `setSize` ou en une seule étape avec la méthode `setBounds`.

```

package fr.eni;

import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import javax.swing.JCheckBox;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JList;
import javax.swing.JPanel;
import javax.swing.JScrollPane;

public class Ecran extends JFrame

{
    JPanel pano;
    JCheckBox chkGras,chkItalique;
    JLabel lblTaille,lblExemple;
    JComboBox cboTaille;
    JList lstPolices;
    JScrollPane defilPolices;

    public Ecran()
    {
        setTitle("choix d'une police");
        setBounds(0,0,300,100);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // création des composants
        pano=new JPanel();
        chkGras=new JCheckBox("gras");
        chkItalique=new JCheckBox("italique");

```

```

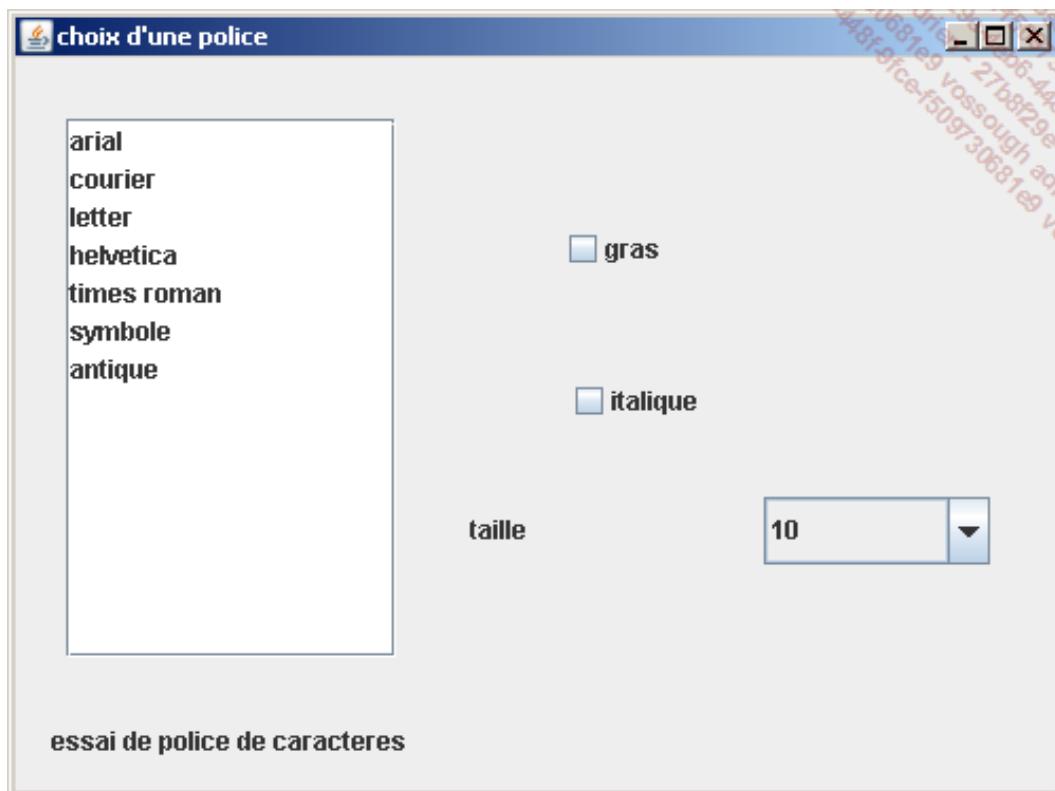
lblTaille=new JLabel("taille");
lblExemple=new JLabel("essai de police de caractères");
cboTaille=new JComboBox(new String[]{"10","12","14","16","18","20"});
lstPolices=new JList(new
String[]{"arial","courier","letter","helvetica","times roman","symbole",
"antique"});
defilPolices=new JScrollPane(lstPolices);

// ajout sur le conteneur
pano.setLayout(null);
pano.add(defilPolices);
pano.add(chkGras);
pano.add(chkItalique);
pano.add(lblTaille);
pano.add(cboTaille);
pano.add(lblExemple);

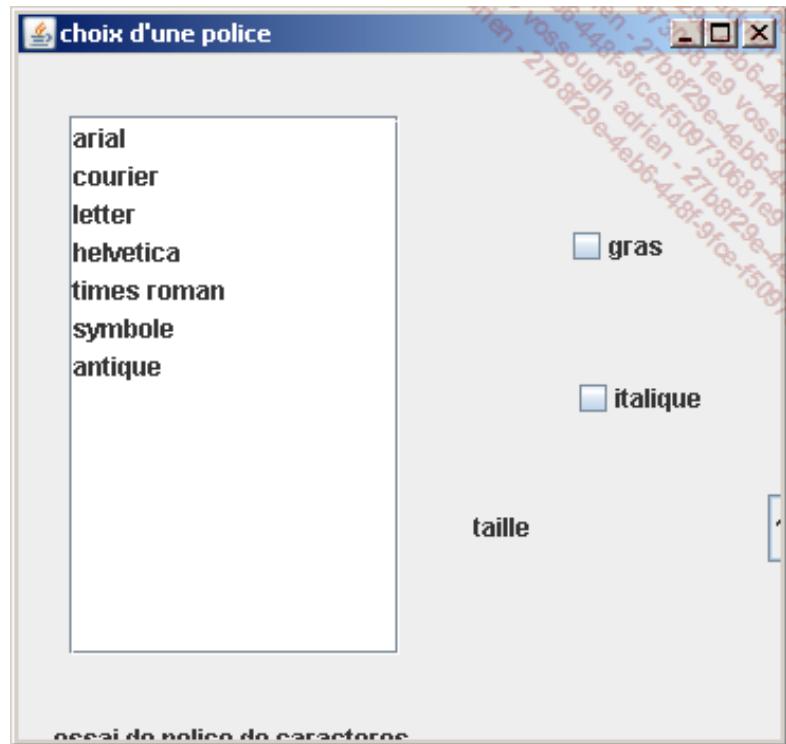
// position des composants.
defilPolices.setBounds(24, 29, 156, 255);
chkGras.setBounds(258, 78, 170, 25);
chkItalique.setBounds(261, 139, 167, 46);
lblTaille.setBounds(215, 211, 106, 24);
cboTaille.setBounds(354, 208, 107, 32);
lblExemple.setBounds(17, 309, 459, 28);
getContentPane().add(pano);

```

Notre application présente bien le même aspect que si elle utilisait un gestionnaire de mise en page.



Par contre si la fenêtre est redimensionnée, les composants conservent leurs tailles et positions et l'interface peut alors prendre un aspect incohérent.



5. Les composants graphiques

Chaque composant utilisable avec Java est représenté par une classe dont nous allons pouvoir créer des instances pour concevoir l'interface de l'application. La majorité des composants Swing dérivent de la classe `JComponent` et de ce fait, héritent d'un bon nombre de propriétés et de méthodes de cette classe ou des classes situées au-dessus dans la hiérarchie d'héritage.



Nous allons donc étudier les éléments les plus utiles de la classe `JComponent`.

a. La classe `JComponent`

Dimensions et position

Une multitude de méthodes interviennent dans la gestion de la position et de la taille des composants. Deux catégories d'informations sont disponibles pour la taille des composants :

- La taille actuelle est accessible par la méthode `getSize`. Cette dernière obtient la taille du composant au moment de l'appel de la méthode. Si le composant est pris en charge par un gestionnaire de mise en page, celui-ci est susceptible de modifier la taille du composant et donc l'appel de cette méthode peut renvoyer un résultat différent à chaque appel. La taille peut aussi être modifiée par la méthode `setSize`. Les effets de cette méthode peuvent être de courte durée car, si le composant est pris en charge par un gestionnaire de mise en page, celui-ci peut à tout moment redessiner le composant avec une taille différente.

- La taille préférée est accessible par la méthode `getPreferredSize`. Par défaut cette taille est calculée en fonction du contenu du composant. Cette méthode est surtout utilisée en interne par les gestionnaires de mise en page. Pour éviter que la taille du composant soit recalculée en fonction de son contenu, vous pouvez utiliser la méthode `setPreferredSize`. Les informations fournies par cette méthode sont ensuite utilisées par la méthode `getPreferredSize` lorsque le conteneur a besoin d'informations sur la taille du composant.

La position d'un composant à l'intérieur de son conteneur peut être obtenue ou définie avec les méthodes `getLocation` ou `setLocation`. Comme pour la méthode `setSize` l'effet de la méthode `setLocation` peut être à tout moment annulé par le gestionnaire de mise en page, si celui-ci doit redessiner les composants.

Les méthodes `getBounds` et `setBounds` permettent de combiner la modification de la taille et de la position du composant. Ces méthodes n'ont vraiment d'intérêt que si vous assumez complètement la présentation des composants à l'intérieur du conteneur sans passer par les services d'un gestionnaire de mise en page.

Apparence des composants

La couleur de fond du composant peut être modifiée par la méthode `setBackground` tandis que la couleur du texte du composant est modifiée par la propriété `setForeground`. Ces méthodes attendent comme argument un objet `Color`. Cet objet `Color` peut être obtenu par les constantes définies dans la classe `Color`.

```
lstPolices.setBackground(Color.red);
lstPolices.setForeground(Color.GREEN);
```

La classe `Color` propose également de nombreux constructeurs permettant d'obtenir une couleur particulière en effectuant un mélange des couleurs de base.

```
lstPolices.setBackground(new Color(23,67,89));
lstPolices.setForeground(new Color(167,86,23));
```

La police est modifiable par la méthode `setFont` du composant. On peut, pour l'occasion, créer une nouvelle instance de la classe `Font` et l'affecter au composant. Pour cela, nous utiliserons un des constructeurs de la classe `Font` en indiquant le nom de la police, le style de la police et sa taille.

```
lstPolices.setFont(new Font("Serif",Font.BOLD,24));
```

Ces deux propriétés s'appliquent sur l'ensemble du texte affiché sur le composant. Il est possible d'utiliser plusieurs couleurs et polices en formatant le texte du composant à l'aide de balises `html`. Il suffit simplement d'encadrer le texte du composant entre des balises `<html>` et `</html>`. À l'intérieur de ces balises, vous pouvez utiliser ensuite n'importe quelle balise de formatage `html`. L'exemple suivant modifie la couleur du texte et affiche en italique (sauf la première lettre) le libellé du bouton.

```
// création des trois boutons
JButton b1,b2,b3;
b1=new JButton("<html><font color=red>R<i>ouge</i></font>
</html>");
b2=new JButton("<html><font color=green>V<i>ert</i></font>
</html>");
b3=new JButton("<html><font color=blue>B<i>leu</i></font>
</html>");
```

La méthode `setCursor` permet de choisir l'apparence du curseur lorsque la souris se trouve sur la surface du composant. Plusieurs curseurs sont prédéfinis et sont accessibles en créant une instance de la classe `Cursor` avec une des constantes prédéfinies.

```
lstPolices.setCursor(new Cursor(Cursor.HAND_CURSOR));
```

La détection de l'entrée et de la sortie de la souris sur le composant et la modification du curseur en conséquence est gérée automatiquement par le composant lui-même.

Comportement des composants

Les composants placés sur un conteneur peuvent être masqués en appelant la méthode `setVisible(false)` ou désactivés en appelant la méthode `setEnabled(false)`. Dans ce cas, le composant est toujours visible mais apparaît avec un aspect grisé pour indiquer à l'utilisateur que ce composant est inactif pour le moment.

```
chkGras.setVisible(false);
chkItalique.setEnabled(false);
```

Les composants dans cet état ne peuvent bien sûr pas recevoir le focus dans l'application. Vous pouvez vérifier cela en appelant la méthode `isFocusable` qui renvoie un boolean. Vous pouvez également vérifier si un composant détient actuellement le focus, en utilisant la méthode `isFocusOwner`. Le focus peut être placé sur un composant sans l'intervention de l'utilisateur, en appelant la méthode `requestFocus` du composant.

Pour surveiller le passage du focus d'un composant à l'autre, deux événements sont à votre disposition :

- `focusGained` indique qu'un composant particulier a reçu le focus.
- `focusLost` indique qu'un composant a perdu le focus.

Par exemple, pour bien visualiser qu'un composant détient le focus, on peut utiliser le code suivant qui modifie la couleur du texte lorsque le composant reçoit ou perd le focus :

```
lstPolice.addFocusListener(new FocusListener()
{
    public void focusGained(FocusEvent arg0)
    {
        lstPolice.setForeground(Color.RED);
    }

    public void focusLost(FocusEvent arg0)
    {
        lstPolice.setForeground(Color.BLACK);
    }
});
```

b. Affichage d'informations

Le composant JLabel

Le composant `JLabel` est utilisé pour afficher sur un formulaire, un texte qui ne sera pas modifiable par l'utilisateur. Il sert essentiellement à fournir une légende à des contrôles qui n'en possèdent pas (zones de texte par exemple, liste déroulante...). Dans ce cas, il permettra également de fournir un raccourci-clavier pour atteindre ce composant.

Le texte affiché par le composant est indiqué lors de sa création ou par la méthode `setText`. Par défaut, le composant `JLabel` n'a pas de bordure. Vous pouvez en ajouter une en appelant la méthode `setBorder` en lui passant la bordure à utiliser.

```
JLabel legende;
legende=new JLabel("nom");
legende.setBorder(BorderFactory.createEtchedBorder());
```

Vous avez aussi la possibilité d'indiquer la position du texte dans le composant par l'intermédiaire des méthodes `setHorizontalAlignment` et `setVerticalAlignment` en spécifiant une des constantes prédefinies.

```
SwingConstants.RIGHT : alignement sur la droite
SwingConstants.CENTER : alignement sur le centre
```

```
SwingConstants.LEFT : alignement sur la gauche  
SwingConstants.TOP : alignement sur le haut  
SwingConstants.BOTTOM : alignement sur le bas
```

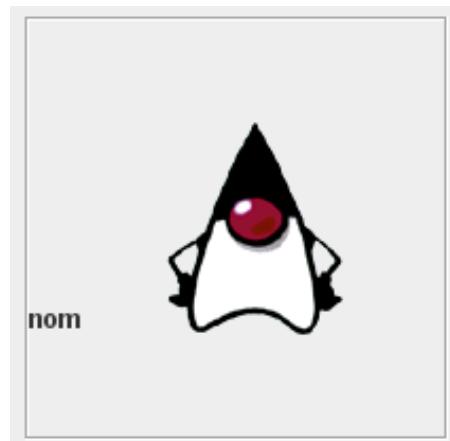
Ces méthodes n'ont aucun effet visible si le composant est dimensionné automatiquement par son conteneur puisque dans ce cas sa taille est automatiquement ajustée à son contenu. Pour que ces méthodes aient un effet visible, vous devez indiquer une taille préférée pour le composant.

```
legende=new JLabel("nom");  
legende.setBorder(BorderFactory.createEtchedBorder());  
legende.setPreferredSize(new Dimension(200,50));  
legende.setHorizontalTextPosition(SwingConstants.LEFT);  
legende.setVerticalTextPosition(SwingConstants.TOP);
```



Les composants `JLabel` peuvent également afficher des images. Vous pouvez indiquer l'image à afficher à l'aide de la méthode `setIcon`. L'image doit bien sûr être créée au préalable. Si du texte est également affiché sur le composant, sa position peut être spécifiée par rapport à l'image en utilisant les méthodes `setHorizontalTextPosition` et `setVerticalTextPosition`, ainsi que l'espace entre le texte et l'image avec la méthode `setIconTextGap`.

```
JLabel legende;  
legende=new JLabel("nom");  
legende.setBorder(BorderFactory.createEtchedBorder());  
legende.setPreferredSize(new Dimension(200,200));  
legende.setIcon(new ImageIcon("duke.gif"));  
legende.setHorizontalTextPosition(SwingConstants.LEFT);  
legende.setVerticalTextPosition(SwingConstants.BOTTOM);  
legende.setIconTextGap(40);
```



Nous avons indiqué que le composant `JLabel` pouvait être utilisé comme raccourci-clavier pour un autre composant. Pour cela, deux précautions sont à prendre.

- Vous devez indiquer, avec la méthode `setDisplayedMnemonic`, le caractère utilisé comme raccourci-clavier.
- Vous devez également indiquer au `JLabel` pour quel composant il joue le rôle de légende en utilisant la méthode `setLabelFor`.

```
legende.setDisplayedMnemonic('n');  
legende.setLabelFor(lstPolices);
```

Le composant JProgressBar

Ce composant est utilisé pour informer l'utilisateur sur la progression d'une action lancée dans l'application. Il affiche cette information sous la forme d'une zone rectangulaire, qui sera plus ou moins remplie en fonction de l'état d'avancement de l'action exécutée.

La position de la barre de progression est contrôlée par la méthode `setValue`. Cette méthode accepte un argument pouvant évoluer entre les deux extrêmes indiqués par les méthodes `setMinimum` et `setMaximum`.

Une légende peut également être affichée en incrustation sur la barre de progression. Le texte de la légende est indiqué par la méthode `setString` et son affichage est contrôlé par la méthode `setStringPainted`.

S'il est impossible d'évaluer la progression d'une opération et donc d'obtenir une valeur à fournir à la barre de progression, celle-ci peut être placée dans un état indéterminé avec la méthode `setIndeterminate`. Un curseur se déplaçant en permanence entre les deux extrêmes est dans ce cas affiché.

L'exemple suivant présente une horloge originale où l'heure est affichée par trois `JProgressBar` :

```
package fr.eni;

import java.util.GregorianCalendar;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JProgressBar;

public class Ecran13 extends JFrame

{
    JPanel pano;
    JProgressBar pgbHeure,pgbMinutes,pgbSeconde,pgbDefil;

    public Ecran13()
    {
        setTitle("horloge");
        setBounds(0,0,300,100);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // création des composants
        pgbHeure =new JProgressBar();
        pgbMinutes=new JProgressBar();
        pgbSeconde=new JProgressBar();
        pgbDefil=new JProgressBar();
        pgbHeure.setMinimum(0);
        pgbHeure.setMaximum(23);
        pgbMinutes.setMinimum(0);
        pgbMinutes.setMaximum(59);
        pgbSeconde.setMinimum(0);
        pgbSeconde.setMaximum(59);
        pgbHeure.setString("heure");
        pgbHeure.setStringPainted(true);
        pgbMinutes.setString("minute");
        pgbMinutes.setStringPainted(true);
        pgbSeconde.setString("seconde");
        pgbSeconde.setStringPainted(true);
        pgbDefil.setString("le temps passe");
        pgbDefil.setStringPainted(true);
        pgbDefil.setIndeterminate(true);
        pano=new JPanel();
        pano.add(pgbHeure);
        pano.add(pgbMinutes);
        pano.add(pgbSeconde);
        pano.add(pgbDefil);
        getContentPane().add(pano);
        Thread th;
        th=new Thread()
        {
            public void run()
            {
```

```

        while (true)
    {
        LocalTime d;
        d=LocalTime.now();
        pgbHeure.setValue(d.getHour());
        pgbMinutes.setValue(d.getMinute());
        pgbSeconde.setValue(d.getSecond());
        try
        {
            sleep(500);
        }
        catch (InterruptedException e)
        {
        }
    }

th.start();
}
}

```



c. Les composants d'édition de texte

JTextField

Le composant `JTextField` est utilisé pour permettre à l'utilisateur de saisir des informations. Ce composant ne permet que la saisie de texte sur une seule ligne. Ce composant est également capable de gérer la sélection de texte et les opérations avec le Presse-papiers. De nombreuses méthodes sont disponibles pour travailler avec ce composant. Par défaut, ce composant adapte automatiquement sa taille à son contenu ce qui peut provoquer un réaffichage permanent du composant. Pour pallier ce problème, il est préférable de spécifier une taille pour le composant soit en utilisant sa méthode `setPreferredSize` soit en indiquant le nombre de colonnes désiré pour l'affichage du texte dans ce composant. Ceci ne limite absolument pas le nombre de caractères pouvant être saisis mais uniquement la largeur du composant. Le texte affiché dans le composant peut être modifié ou récupéré par les méthodes `setText` ou `getText`.

La gestion de la sélection du texte se fait automatiquement par le composant. La méthode `getSelectedText` permet de récupérer la chaîne de caractères actuellement sélectionnée dans le contrôle. Les méthodes `getSelectionStart` et `getSelectionEnd` indiquent respectivement le caractère de début de la sélection (le premier caractère a l'indice 0) et le dernier caractère de la sélection.

La sélection de texte peut également s'effectuer avec la méthode `select`, en indiquant le caractère de début de la sélection et le caractère de fin la sélection.

La sélection de la totalité du texte peut être effectuée avec la méthode `selectAll`. Par exemple, on peut forcer la sélection de tout le texte lorsque le composant reçoit le focus.

```

txt=new JTextField(10);
txt.addFocusListener(new FocusListener()
{
    public void focusGained(FocusEvent e)
    {
        txt.selectAll();
    }
});

```

```

        }
        public void focusLost(FocusEvent e)
        {
            txt.setSelectionStart(0);
            txt.setSelectionEnd(0);
        }
    });
}

```

Pour la gestion du Presse-papiers, le composant `JTextField` gère automatiquement les raccourcis-clavier du système pour les opérations de copier, couper, coller. Vous avez cependant la possibilité d'appeler les méthodes `copy`, `cut` et `paste` pour gérer les opérations de copier, couper, coller d'une autre manière, par exemple par un menu de l'application ou un menu contextuel comme dans l'exemple ci-dessous :

```

txt=new JTextField(10);
mnu=new JPopupMenu();
JMenuItem mnuCopier;
JMenuItem mnuCouper;
JMenuItem mnuColler;
mnuCopier=new JMenuItem("Copier");
mnuCouper=new JMenuItem("Couper");
mnuColler=new JMenuItem("Coller");
mnuCopier.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        txt.copy();
    }
});
mnuCouper.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        txt.cut();
    }
});
mnuColler.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        txt.paste();
    }
});
mnu.add(mnuCopier);
mnu.add(mnuCouper);
mnu.add(mnuColler);
txt.addMouseListener(new MouseAdapter()
{
    public void mousePressed(MouseEvent e)
    {
        if (e.getButton()==MouseEvent.BUTTON3)
        {
            mnu.show((Component)e.getSource(), e.getX(),e.getY());
        }
    }
});
}
}

```

Les opérations couper et coller ne seront cependant pas possibles si le composant `JTextField` est configuré en lecture seule avec la méthode `setEditable(false)`, la modification du texte par l'utilisateur est bien sûr dans ce cas également impossible.

Tout le monde ayant droit à l'erreur, il est possible de provoquer l'annulation de la dernière modification de texte effectuée sur le contrôle. Pour cela, nous devons nous adjoindre l'aide d'un objet `UndoManager`.

C'est ce dernier qui prend effectivement en charge l'annulation de la dernière modification grâce à sa méthode `undo`. Cette méthode peut être appelée par une option du menu contextuel ou par le raccourci-clavier [Ctrl] Z. Il n'y a qu'un seul niveau de "Undo", vous ne pourrez revenir au texte que vous avez saisi, il y a deux heures !

```
// création du UndoManager
UndoManager udm;
udm=new UndoManager();
// association avec le JTextField
txt.getDocument().addUndoableEditListener(udm);
// ajout d'un écouteur pour intercepter le ctrl Z
txt.addKeyListener(new KeyAdapter()
{
    public void keyPressed(KeyEvent e)
    {
        if (e.getKeyChar()=='z' & e.isControlDown())
        {
            udm.undo();
        }
    }
});
```

JPasswordField

Ce composant est à la base un `JTextField` normal légèrement modifié pour être spécialisé dans la saisie de mot de passe. Les deux seuls ajouts réellement utiles sont la méthode `setEchoChar` permettant d'indiquer le caractère utilisé comme caractère de substitution lors de l'affichage et la méthode `getPassword` permettant d'obtenir le mot de passe saisi par l'utilisateur.

JTextArea

Ce composant offre des fonctionnalités plus évoluées qu'un simple `JTextField`. La première amélioration importante apportée par ce composant réside dans sa capacité à gérer la saisie sur plusieurs lignes. C'est d'ailleurs pour cette raison que lors de la construction d'un objet d'un objet `JTextArea` nous devons spécifier le nombre de lignes et le nombre de colonnes qu'il va comporter. Ces deux informations sont utilisées uniquement pour le dimensionnement du composant et n'influencent pas la quantité de texte que peut contenir le composant. Par contre, ce composant ne gère pas lui-même le défilement du texte qu'on lui confie. Pour lui adjoindre cette fonctionnalité, celui-ci doit être placé sur un conteneur de type `JScrollPane` qui va prendre en charge le défilement du contenu du `JTextArea`.

Si cette solution n'est pas utilisée, le composant `JTextArea` peut être configuré pour scinder automatiquement une ligne lors de son affichage. La méthode `setLineWrap` permet d'activer ce mode de fonctionnement. Par défaut le découpage se produit sur le dernier caractère visible d'une ligne avec le risque de voir certains mots affichés sur deux lignes. Pour éviter ce problème, le composant `JTextArea` peut être configuré pour effectuer son découpage sur les espaces séparant les mots avec la méthode `setWrapStyleWord`. Les sauts de ligne ajoutés par le composant ne font pas partie du texte et sont utilisés uniquement pour l'affichage de celui-ci.

```
package fr.eni;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;

public class Ecran extends JFrame
{
    JPanel pano;
    JTextArea txt;
    JCheckBox chkWrap,chkWrapWord;
```

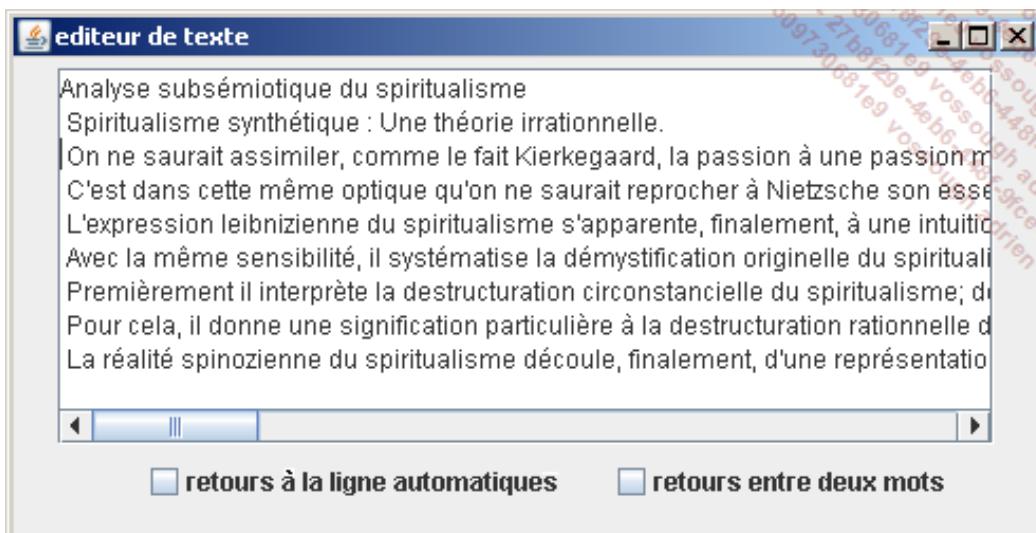
```

JScrollPane defil;

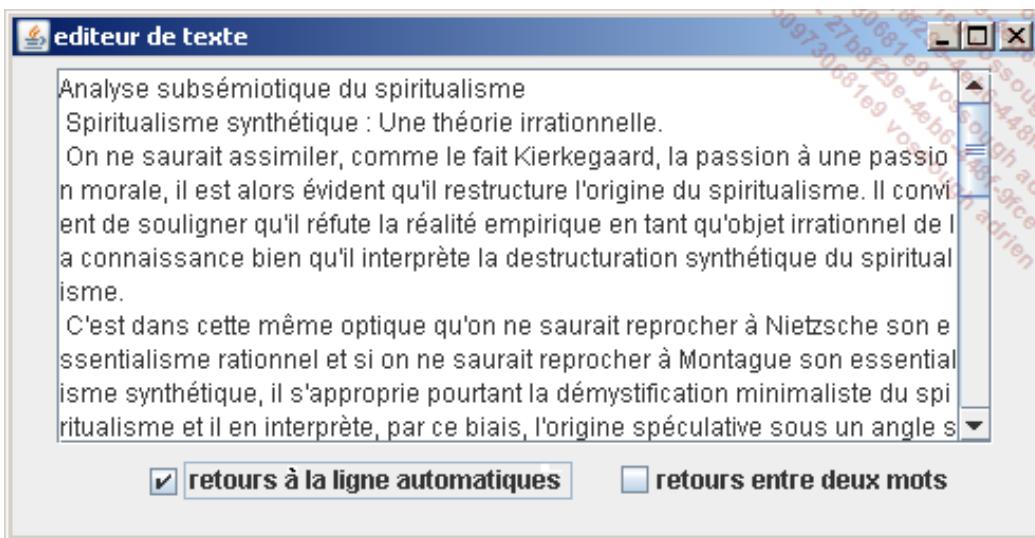
public Ecran()
{
    setTitle("éditeur de texte");
    setBounds(0,0,300,100);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    // création des composants
    pano=new JPanel();
    txt=new JTextArea(10,40);
    defil=new JScrollPane(txt);
    pano.add(defil);
    chkWrap=new JCheckBox("retours à la ligne automatiques");
    chkWrapWord=new JCheckBox("retours entre deux mots");
    chkWrap.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            txt.setLineWrap(chkWrap.isSelected());
        }
    });
    chkWrapWord.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            txt.setWrapStyleWord(chkWrapWord.isSelected());
        }
    });
    pano.add(chkWrap);
    pano.add(chkWrapWord);
    getContentPane().add(pano);
}
}

```

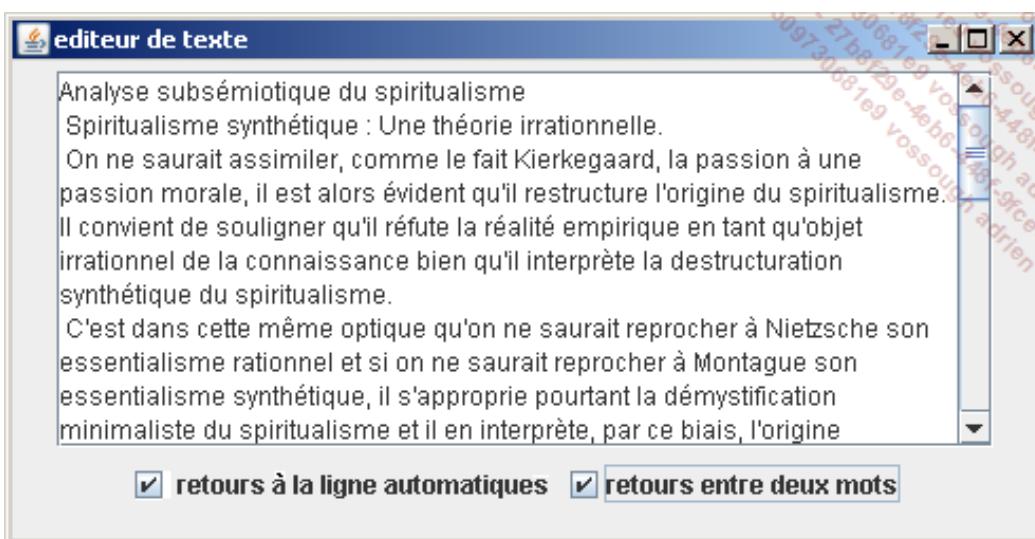
Avec les options par défaut les lignes trop longues ne sont pas visibles entièrement et la barre de défilement horizontal est activée automatiquement.



Avec l'option `LineWrap`, les lignes sont découpées automatiquement à la largeur du composant. Il n'y a plus besoin de barre de défilement horizontal. Par contre une barre de défilement vertical est maintenant nécessaire puisque le nombre de lignes est maintenant supérieur à la capacité du composant.



Pour améliorer la lisibilité du texte, le découpage peut être fait sur des mots entiers avec l'option `WrapStyleWord`.



La gestion du texte contenu dans ce composant est également facilitée grâce à plusieurs méthodes spécifiques.

- `append(String chaîne)` : ajoute la chaîne de caractères passée comme argument au texte déjà présent dans le composant.
- `insert(String chaîne, int position)` : insère la chaîne de caractères passée comme premier argument à la position indiquée par le deuxième argument.
- `replaceRange(String chaîne, int debut, int fin)` : remplace la portion de texte comprise entre la valeur fournie par le deuxième argument et celle fournie par le troisième argument par la chaîne indiquée comme premier argument.

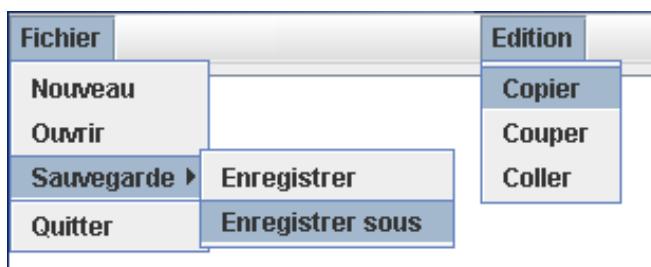
d. Les composants de déclenchement d'actions

JButton

Le composant `JButton` est principalement utilisé dans une application, pour lancer l'exécution d'une action. Cette action peut être l'exécution d'une portion de code ou la fermeture d'une boîte de dialogue. Comme pour les contrôles vus jusqu'à présent, le libellé du bouton est modifiable par la méthode `setText` du composant. Ce composant peut également contenir une image. Celle-ci est gérée exactement de la même façon que pour le composant `JLabel`. Ce composant est pratiquement toujours associé à un écouteur implémentant l'interface `ActionListener` afin de gérer les clics sur le bouton.

JMenuBar, JMenu, JMenuItem, JPopupMenu, JSeparator

Cet ensemble de composants va permettre la gestion des menus d'application ou des menus contextuels. Le composant `JMenuBar` constitue le conteneur qui va accueillir les menus représentés par des `JMenu` qui eux-mêmes vont contenir des éléments de menus représentés par des `JMenuItem`. Le composant `JMenuBar` se comporte comme un conteneur pour les `JMenu`. Les `JMenu` vont eux aussi se comporter comme un conteneur pour des `JMenuItem` ou d'autres `JMenu`. La conception d'un menu va donc consister à créer des instances de ces différentes classes puis de les associer les unes aux autres. Au final le `JMenuBar` obtenu est placé sur son conteneur qui n'est autre que la `JFrame` de l'application. Il faut bien sûr également penser à traiter les événements déclenchés par ces différents éléments. En général, seuls les `JMenuItem` sont associés à des écouteurs. Ils se comportent comme des `JButton`. Ils ont d'ailleurs un ancêtre commun puisque tous deux héritent de la classe `AbstractButton`. La conception d'un menu n'est pas très complexe. La seule difficulté réside dans la quantité de code relativement importante nécessaire. Pour illustrer cela, voici le code d'un éditeur de texte rudimentaire possédant les menus suivants :



```
package fr.eni;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.PrintWriter;

import javax.swing.JCheckBox;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JSeparator;
import javax.swing.JTextArea;

public class Ecran14 extends JFrame
{
    JPanel pano;
    JTextArea txt;
    JScrollPane defil;
    JMenuBar barre;
    JMenu mnuFichier, mnuEdition, mnuSauvegarde;
    JMenuItem mnuNouveau, mnuOuvrir, mnuEnregister,
    mnuEnregisterSous, mnuQuitter;
    JMenuItem mnuCopier, mnuCouper, mnuColler;
    File fichier;

    public Ecran14()
    {
        setTitle("éditeur de texte");
    }
}
```

```

setBounds(0,0,300,100);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
// création des composants
pano=new JPanel();
pano.setLayout(new BorderLayout());
txt=new JTextArea();
defil=new JScrollPane(txt);
pano.add(defil,BorderLayout.CENTER);
getContentPane().add(pano);
// création des composants des menus
barre=new JMenuBar();
mnuFichier=new JMenu("Fichier");
mnuEdition=new JMenu("Edition");
mnuSauvegarde=new JMenu("Sauvegarde");
mnuNouveau=new JMenuItem("Nouveau");
mnuOuvrir=new JMenuItem("Ouvrir");
mnuEnregister=new JMenuItem("Enregistrer");
mnuEnregister.setEnabled(false);
mnuEnregisterSous=new JMenuItem("Enregistrer sous");
mnuCopier=new JMenuItem("Copier");
mnuCouper=new JMenuItem("Couper");
mnuColler=new JMenuItem("Coller");
mnuQuitter=new JMenuItem("Quitter");
// association des éléments
barre.add(mnuFichier);
barre.add(mnuEdition);
mnuFichier.add(mnuNouveau);
mnuFichier.add(mnuOuvrir);
mnuFichier.add(mnuSauvegarde);
mnuSauvegarde.add(mnuEnregister);
mnuSauvegarde.add(mnuEnregisterSous);
mnuFichier.add(new JSeparator());
mnuFichier.add(mnuQuitter);
mnuEdition.add(mnuCopier);
mnuEdition.add(mnuCouper);
mnuEdition.add(mnuColler);
// association du menu avec la JFrame
setJMenuBar(barre);
// les écouteurs associés aux différents menus
mnuNouveau.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent arg0)
    {
        fichier=null;
        txt.setText("");
        mnuEnregister.setEnabled(false);
    }
});

mnuOuvrir.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent arg0)
    {
        JFileChooser dlg;
        dlg=new JFileChooser();
        dlg.showDialog(null,"Ouvrir");
        fichier=dlg.getSelectedFile();
        FileInputStream in;
        try
        {
            in=new FileInputStream(fichier);
            BufferedReader br;
            br=new BufferedReader(new
InputStreamReader(in));
            String ligne;
            txt.setText("");
            while ((ligne=br.readLine())!=null)
            {
                txt.append(ligne+"\r\n");
            }
        }
    }
});

```

```

                br.close();
                mnuEnregister.setEnabled(true);
            }
            catch (FileNotFoundException e)
            {
                e.printStackTrace();
            }
            catch (IOException e)
            {
                e.printStackTrace();
            }
        }
    });
mnuQuitter.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        System.exit(0);
    }
});
mnuCopier.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        txt.copy();
    }
});
mnuCouper.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        txt.cut();
    }
});
mnuColler.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        txt.paste();
    }
});
mnuEnregister.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent arg0)
    {
        try
        {
            PrintWriter pw;
            pw=new PrintWriter(fichier);
            pw.write(txt.getText());
            pw.close();
        }
        catch (FileNotFoundException e)
        {
            e.printStackTrace();
        }
    }
});
mnuEnregistrerSous.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent arg0)
    {
        try
        {
            JFileChooser dlg;
            dlg=new JFileChooser();
            dlg.showDialog(null,"enregistrer sous");
            fichier=dlg.getSelectedFile();
            PrintWriter pw;
            pw=new PrintWriter(fichier);

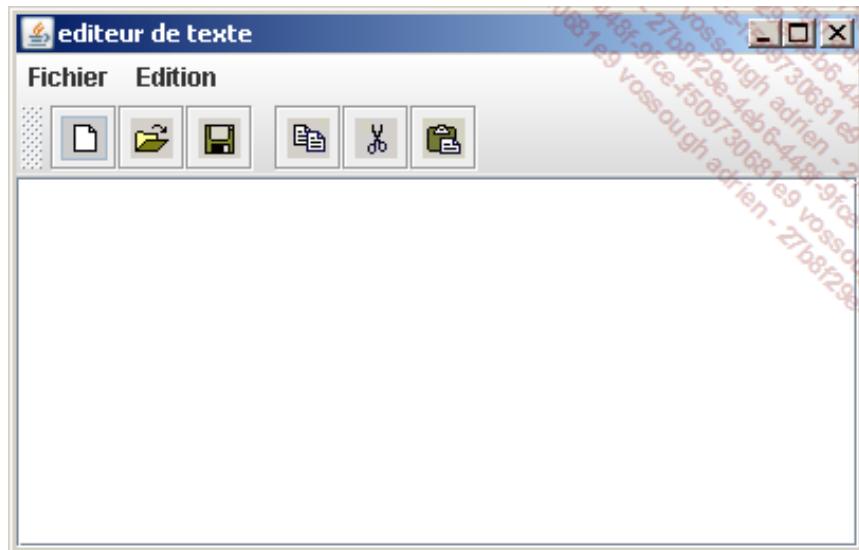
```

```
        pw.write(txt.getText());
        pw.close();
    }
    catch (FileNotFoundException e)
    {
        e.printStackTrace();
    }
}
});
```

JToolBar

Ce composant sert en fait de conteneur pour les éléments constituant une barre d'outils. Ces éléments sont ajoutés au composant `JToolBar` comme pour n'importe quel autre conteneur. Des `JButton` sans libellé mais affichant des images sont généralement placés sur ce composant. Ils fournissent un accès à une fonctionnalité déjà disponible par un menu. À ce titre, ils partagent généralement le même écouteur. Ce composant présente la particularité de pouvoir être déplacé par l'utilisateur sur n'importe laquelle des bordures de la fenêtre, voire dans certains cas à l'extérieur de la fenêtre. Pour que ce mécanisme fonctionne correctement, le composant `JToolBar` doit être placé dans un conteneur utilisant un `BorderLayout` comme gestionnaire de mise en page. Il est dans ce cas placé sur une des bordures (NORTH, SOUTH, WEST, EAST) et doit être le seul composant placé dans cette zone. Cette fonctionnalité peut être désactivée en appelant la méthode `setFloatable(false)`. La barre d'outils reste alors à sa position d'origine. Le code suivant ajoute une barre d'outils à notre éditeur de texte.

```
JToolBar tlbr;
tlbr=new JToolBar();
JButton btnNouveau,btnOuvrir,btnEnregister;
JButton btnCopier,btnCouper,btnColler;
// création des boutons
btnNouveau=new JButton(new ImageIcon("new.jpg"));
btnOuvrir=new JButton(new ImageIcon("open.jpg"));
btnEnregister=new JButton(new ImageIcon("save.jpg"));
btnCopier=new JButton(new ImageIcon("copy.jpg"));
btnColler=new JButton(new ImageIcon("paste.jpg"));
btnCouper=new JButton(new ImageIcon("cut.jpg"));
// ajout des boutons à la barre d'outils
tlbr.add(btnNouveau);
tlbr.add(btnOuvrir);
tlbr.add(btnEnregister);
tlbr.addSeparator();
tlbr.add(btnCopier);
tlbr.add(btnCouper);
tlbr.add(btnColler);
// ajout de la barre d'outils sur son conteneur
pano.add(tlbr,BorderLayout.NORTH);
// réutilisation écouteurs déjà associés aux menus
btnNouveau.addActionListener(mnuNouveau.getActionListeners()[0]);
btnOuvrir.addActionListener(mnuOuvrir.getActionListeners()[0]);
btnEnregister.addActionListener(mnuEnregister.getActionListeners()[0]);
btnCopier.addActionListener(mnuCopier.getActionListeners()[0]);
btnCouper.addActionListener(mnuCouper.getActionListeners()[0]);
btnColler.addActionListener(mnuColler.getActionListeners()[0]);
```



e. Les composants de sélection

JCheckBox

Le composant `JCheckBox` est utilisé pour proposer à l'utilisateur différentes options, parmi lesquelles il pourra en choisir aucune, une, ou plusieurs. Le composant `JCheckBox` peut prendre deux états : coché lorsque l'option est sélectionnée ou non coché lorsque l'option n'est pas sélectionnée. L'état de la case à cocher peut être vérifié ou modifié par les méthodes `isSelected` et `setSelected`. Comme beaucoup d'autres composants, celui-ci peut avoir un libelle et une image. La gestion de l'image est cependant un petit peu différente des `JLabel` puisqu'elle vient remplacer le dessin de la case à cocher. Pour pouvoir dans ce cas faire une distinction entre une `JCheckBox` cochée et une non cochée, vous devez lui associer une deuxième image correspondant à l'état coché. Cette deuxième image est indiquée par la méthode `setSelectedIcon`.

Le code suivant permet de choisir le style de la police utilisée dans notre éditeur de texte.

```
JPanel options;
GridLayout gl;
options=new JPanel();
gl=new GridLayout(2,1);
options.setLayout(gl);
chkGras=new JCheckBox("Gras");
chkItalique=new JCheckBox("Italique");
options.add(chkGras);
options.add(chkItalique);
chkGras.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        changePolice();
    }
});
chkItalique.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        changePolice();
    }
});
pano.add(options, BorderLayout.SOUTH);
}

public void changePolice()
{
```

```

int attributs;
attributs=0;
if (chkGras.isSelected())
{
    attributs=attributs+Font.BOLD;
}
if (chkItalique.isSelected())
{
    attributs=attributs+Font.ITALIC;
}

Font police;
police=new
Font(txt.getFont().getName(),attributs,txt.getFont().getSize());
txt.setFont(police);
}

```

JRadioButton

Le composant `JRadioButton` permet également de proposer à l'utilisateur différentes options parmi lesquelles il ne pourra en sélectionner qu'une seule. Comme son nom l'indique, ce contrôle fonctionne comme les boutons permettant de sélectionner une station sur un poste de radio (vous ne pouvez pas écouter trois stations de radio en même temps !). Les caractéristiques de ce composant sont identiques à celles disponibles dans le composant `JCheckBox`. Pour pouvoir fonctionner correctement, les composants `JRadioButton` doivent être regroupés logiquement. La classe `ButtonGroup` fournit cette fonctionnalité. Elle fait en sorte que parmi tous les `JRadioButton` qui lui sont confiés, grâce à sa méthode `add`, un seul d'entre eux pourra être sélectionné à la fois. Pour regrouper physiquement des `JRadioButton` ceux-ci doivent être placés sur un conteneur tel qu'un `JPanel`. Pour bien visualiser ce regroupement, on ajoute généralement une bordure sur le `JPanel` et éventuellement une légende.

C'est ce que nous allons faire en ajoutant deux ensembles de boutons permettant de choisir la couleur du texte et la couleur de fond de notre éditeur.

```

JRadioButton optFondRouge,optFondVert,optFondBleu;
JRadioButton optRouge,optVert,optBleu;
 JPanel couleur,couleurFond;
ButtonGroup grpCouleur,grpCouleurFond;
// création des boutons
optRouge=new JRadioButton("Rouge");
optVert=new JRadioButton("Vert");
optBleu=new JRadioButton("Bleu");
optFondRouge=new JRadioButton("Rouge");
optFondVert=new JRadioButton("Vert");
optFondBleu=new JRadioButton("Bleu");
// regroupement logique des boutons
grpCouleur=new ButtonGroup();
grpCouleur.add(optRouge);
grpCouleur.add(optVert);
grpCouleur.add(optBleu);
grpCouleurFond=new ButtonGroup();
grpCouleurFond.add(optFondRouge);
grpCouleurFond.add(optFondVert);
grpCouleurFond.add(optFondBleu);
// regroupement physique des boutons
couleur=new JPanel();
couleur.setLayout(new GridLayout(0,1));
couleur.add(optRouge);
couleur.add(optVert);
couleur.add(optBleu);
couleurFond=new JPanel();
couleurFond.setLayout(new GridLayout(0,1));
couleurFond.add(optFondRouge);
couleurFond.add(optFondVert);
couleurFond.add(optFondBleu);
// ajout d'une bordure avec titre
couleur.setBorder(BorderFactory.createTitledBorder(
BorderFactory.createEtchedBorder(),"couleur police"));

```

```

couleurFond.setBorder(BorderFactory.createTitledBorder(
    BorderFactory.createEtchedBorder(),"couleur fond"));
// référencement des écouteurs
optBleu.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        txt.setForeground(Color.BLUE);
    }
});
optVert.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        txt.setForeground(Color.GREEN);
    }
});
optRouge.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        txt.setForeground(Color.RED);
    }
});
optFondBleu.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        txt.setBackground(Color.BLUE);
    }
});
optFondRouge.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        txt.setBackground(Color.RED);
    }
});
optFondVert.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        txt.setBackground(Color.GREEN);
    }
});
}

```

JList

Le composant `JList` propose à l'utilisateur une liste de choix dans laquelle il pourra en sélectionner un ou plusieurs. Le composant gère automatiquement l'affichage des éléments cependant, il n'est pas capable d'en assurer le défilement. Ce composant doit donc être placé sur un conteneur prenant en charge cette fonctionnalité comme un `JScrollPane`. Les éléments affichés dans la liste peuvent être fournis au moment de la création du composant en indiquant comme argument au constructeur un tableau d'objet ou un `Vector`. Les méthodes `toString` de chacun des objets seront utilisées par le composant `JList` pour pouvoir afficher l'élément. Avec cette solution, la liste est créée en lecture seule et il n'est pas possible d'y ajouter d'autres éléments par la suite. Pour avoir une liste dans laquelle il sera possible d'ajouter d'autres éléments, il faut utiliser un objet de type `DefaultListModel` à qui l'on va confier le soin de gérer les éléments de la liste. Cet objet possède de nombreuses méthodes, similaires aux méthodes disponibles pour un `Vector`, permettant la gestion des éléments. Si vous souhaitez utiliser une liste dynamique, vous devez passer au constructeur de la classe `JList` un objet de ce type.

La liste peut être conçue pour autoriser différents types de sélection. Le choix du type de sélection s'effectue avec la méthode `setSelectionMode` à laquelle l'on passe comme argument une des constantes suivantes :

- `ListSelectionModel.SINGLE_SELECTION` : un seul élément de la liste peut être sélectionné à la fois.

- `ListSelectionModel.SINGLE_INTERVAL_SELECTION` : plusieurs éléments peuvent être sélectionnés mais ils doivent se suivre dans la liste.
- `ListSelectionModel.MULTIPLE_INTERVAL_SELECTION` : plusieurs éléments peuvent être sélectionnés dans la liste et ils ne sont pas forcément contigus.

La récupération de l'élément ou des éléments sélectionnés peut être effectuée avec deux techniques différentes. Vous pouvez obtenir l'index ou les index des éléments sélectionnés avec les méthodes `getSelectedIndex` ou `getSelectedIndices`. Ces méthodes retournent un entier ou un tableau d'entiers représentant l'index ou les index des éléments sélectionnés.

L'objet ou les objets sélectionnés sont eux disponibles grâce aux méthodes `getSelectedValue` et `getSelectedValues`. De manière similaire, ces deux méthodes retournent l'objet sélectionné ou un tableau contenant les objets sélectionnés. La présence d'un élément sélectionné dans la liste peut être testée avec la méthode `isSelectionEmpty`. La sélection peut être annulée avec la méthode `clearSelection`. La modification de la sélection dans la liste déclenche un événement de type `valueChanged`. Cet événement peut être géré en associant à la liste un écouteur de type `ListSelectionListener` avec la méthode `addListSelectionListener`. Il faut être vigilant avec cet événement car il se produit plusieurs fois de suite lorsqu'un élément est sélectionné. Le premier événement correspond à la désélection de l'élément précédent et le deuxième à la sélection de l'élément actuel. C'est donc ce dernier qu'il faut prendre en compte. Pour cela, l'argument `ListSelectionEvent` disponible avec cet événement dispose de la méthode `getValueIsAdjusting` permettant de savoir si la sélection est terminée ou si elle se trouve dans un état transitoire. Pour tester le fonctionnement, nous allons ajouter à notre éditeur de texte une liste proposant le choix d'une police de caractères.

```
JList polices ;
JScrollPane defilPolices;

String[] nomsPolices={"Dialog","DialogInput","Monospaced","Serif",
"SansSerif"};
polices=new JList(nomsPolices);
polices.setSelectedIndex(0);
defilPolices=new JScrollPane(polices);
defilPolices.setPreferredSize(new Dimension(100,60));
options.add(defilPolices);
polices.addListSelectionListener(new ListSelectionListener()
{
    public void valueChanged(ListSelectionEvent e)
    {
        if (!e.getValueIsAdjusting())
        {
            changePolice();
        }
    }
});
...
...
...
public void changePolice()
{
    int attributs;
    attributs=0;
    if (chkGras.isSelected())
    {
        attributs=attributs+Font.BOLD;
    }
    if (chkItalique.isSelected())
    {
        attributs=attributs+Font.ITALIC;
    }
    Font police;
    police=new Font(polices.getSelectedValue().toString(),
attributs,txt.getFont().getSize());
);
    txt.setFont(police);
}
```

JCombobox

Ce composant peut être assimilé à l'association de trois autres composants :

- un JTextField
- un JButton
- un JList

L'utilisateur peut choisir un élément dans la liste qui apparaît lorsqu'il clique sur le bouton ou bien saisir directement dans la zone de texte l'information attendue si celle-ci n'est pas disponible dans la liste. Deux modes de fonctionnement sont disponibles. Vous pouvez n'autoriser que la sélection d'un élément dans la liste en configurant la zone de texte en lecture seule avec la méthode setEditable(false) ou autoriser soit la sélection dans la liste, soit la saisie dans la zone de texte avec la méthode setEditable(true). La création d'une instance de ce composant suit les mêmes principes que la création d'un JList. La récupération de l'élément sélectionné est effectuée de manière semblable à une JList. Attention tout de même car il n'y a pas de sélections multiples dans ce type de composant. La détection de la sélection d'un nouvel élément peut être faite en gérant l'événement actionPerformed. Cet événement se déclenche lors de la sélection d'un élément dans la liste ou lorsque l'utilisateur valide la saisie avec la touche Enter. Nous terminons notre éditeur de texte en proposant à l'utilisateur le choix d'une taille de police de caractères.

```
JComboBox cboTaille ;
String tailles[]={ "10", "12", "14", "16", "20"};
cboTaille=new JComboBox(tailles);
cboTaille.setEditable(true);
options.add(cboTaille);
cboTaille.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        changePolice();
    }
});

});
...
...
...
public void changePolice()
{
    int attributs;
    attributs=0;
    if (chkGras.isSelected())
    {
        attributs=attributs+Font.BOLD;
    }
    if (chkItalique.isSelected())
    {
        attributs=attributs+Font.ITALIC;
    }

    Font police;
    System.out.println(cboTaille.getSelectedItem().toString());
    police=new Font(polices.getSelectedItem().toString(),
    attributs,Integer.parseInt (cboTaille.getSelectedItem().
    toString()));
    txt.setFont(police);
}
```

6. Les boîtes de dialogue

Les boîtes de dialogue sont des fenêtres qui ont une fonction spéciale dans une application. Elles sont en général utilisées pour demander la saisie d'informations à l'utilisateur, lui présenter un message ou lui poser une question. Pour s'assurer que la boîte de dialogue a bien été prise en compte par l'utilisateur avant qu'il

continue à utiliser l'application, celle-ci est affichée en mode modal, c'est-à-dire que le reste de l'application, est bloqué tant que la boîte de dialogue est affichée. Pour nous éviter d'avoir à recréer à chaque fois une nouvelle boîte de dialogue, nous avons à notre disposition plusieurs boîtes de dialogue prédéfinies.

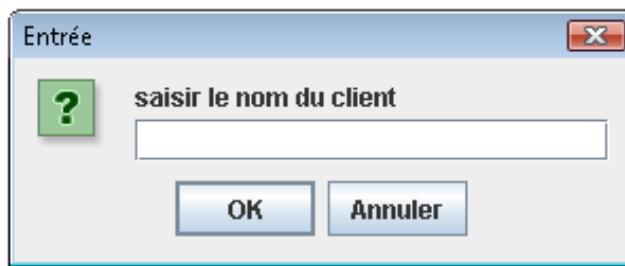
Elles sont disponibles par l'intermédiaire de méthodes static dans la classe JOptionPane.

a. La boîte de saisie

La boîte de saisie permet de demander à l'utilisateur la saisie d'une chaîne de caractères. Cette fonctionnalité est disponible par l'intermédiaire de la fonction showInputDialog de la classe JOptionPane. Plusieurs versions de cette fonction sont à notre disposition pour nous permettre de configurer différemment l'affichage de la boîte de dialogue. La plus simple n'attend comme paramètre qu'une seule chaîne de caractères représentant le message affiché sur la boîte de saisie. Généralement, ce message indique la nature des informations que l'utilisateur doit saisir. Le code suivant :

```
String nom;  
nom=JOptionPane.showInputDialog("saisir le nom du client");
```

affiche cette boîte de dialogue :



Une version plus complète permet de choisir :

- le mode d'affichage : fenêtre autonome ou fenêtre interne à un composant. Dans ce cas, il faut fournir comme premier paramètre le composant concerné et la valeur null si la fenêtre est autonome.
- le message affiché sur la boîte de saisie. C'est le deuxième paramètre qui est utilisé à cet effet.
- le titre de la boîte de saisie comme troisième paramètre.
- le type d'icône affichée sur la boîte de saisie. Une série de constantes définies dans la classe permet d'effectuer ce choix.

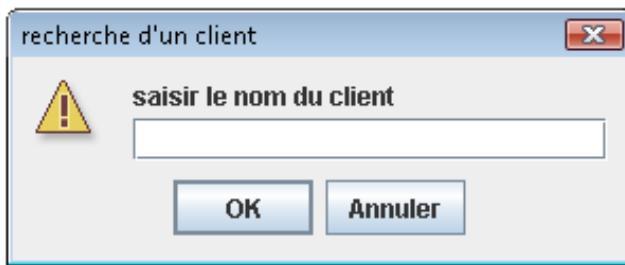
Constante	Icône
ERROR_MESSAGE	
INFORMATION_MESSAGE	
WARNING_MESSAGE	
QUESTION_MESSAGE	

PLAIN_MESSAGE

Pas d'icône

Voici un exemple d'utilisation de cette version plus évoluée et le résultat correspondant.

```
String nom;
nom=JOptionPane.showInputDialog(null,"saisir le nom du client",
"recherche d'un client",JOptionPane.WARNING_MESSAGE);
```



Une autre version ayant un fonctionnement légèrement différent est également disponible. Celle-ci ne propose pas une saisie libre dans une zone de texte, mais une liste d'objets parmi lesquels l'utilisateur devra faire son choix. La liste représente les objets en appelant la méthode `toString` de chacun d'eux. Elle attend comme paramètres les informations suivantes :

- le mode d'affichage : fenêtre autonome ou fenêtre interne à un composant. Dans ce cas, il faut fournir comme premier paramètre le composant concerné et la valeur `null` si la fenêtre est autonome.
- le message affiché sur la boîte de saisie. C'est le deuxième paramètre qui est utilisé à cet effet.
- le titre de la boîte de saisie comme troisième paramètre.
- le type d'icône affichée sur la boîte de saisie. Une série de constantes définies dans la classe permet d'effectuer ce choix.
- une icône permettant le remplacement de l'icône par défaut ou `null` pour conserver l'icône par défaut.
- un tableau d'objets constituant la liste des choix présentés à l'utilisateur.
- l'objet représentant le choix par défaut sélectionné à l'affichage de la boîte de saisie.

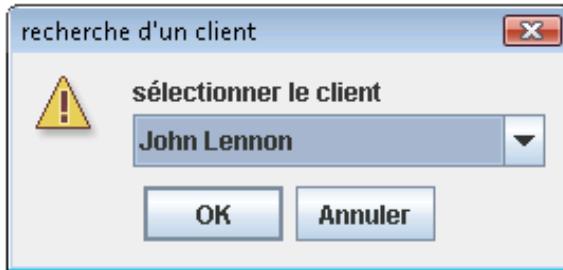
En retour, cette fonction fournit l'objet sélectionné ou `null` si le bouton **Annuler** est utilisé pour fermer la boîte de saisie.

L'exemple ci-dessous propose la sélection d'une personne parmi toutes celles proposées dans la liste.

```
Personne[] choix;
choix=new Personne[5];
choix[0] = new Personne("Wayne", "John", LocalDate.of(1907,5,26));
choix[1] = new Personne("McQueen", "Steeve",LocalDate.of(1930,3,24));
choix[2] = new Personne("Lennon", "John",LocalDate.of(1940,10,9));
choix[3] = new Personne("Gibson", "Mel",LocalDate.of(1956,1,3));
choix[4] = new Personne("Willis", "Bruce",LocalDate.of(1955,3,19));

Personne choisie;

choisie=(Personne)JOptionPane.showInputDialog(null,
"sélectionner le client","recherche d'un client",
JOptionPane.WARNING_MESSAGE,null,choix,choix[1]);
```



b. La boîte de message

La boîte de message permet de passer une information à l'utilisateur. La boîte de message est disponible par l'intermédiaire de la fonction `showMessageDialog`.

Cette fonction accepte les paramètres suivants :

- le mode d'affichage : fenêtre autonome ou fenêtre interne à un composant. Dans ce cas, il faut fournir comme premier paramètre le composant concerné et la valeur `null` si la fenêtre est autonome.
- le message affiché sur la boîte de message. C'est le deuxième paramètre qui est utilisé à cet effet.
- le titre de la boîte de message comme troisième paramètre.
- le type d'icône affichée sur la boîte de saisie. Une série de constantes définies dans la classe permet d'effectuer ce choix. Ces constantes sont identiques à celles utilisables pour la boîte de saisie.

Le code suivant :

```
JOptionPane.showMessageDialog(null,"le client est introuvable",
"recherche d'un client",JOptionPane.WARNING_MESSAGE);
```

affiche cette boîte de message :



c. La boîte de confirmation

Cette boîte de dialogue est utilisée pour poser une question à l'utilisateur et lui permet de répondre par l'intermédiaire de l'un des boutons qu'elle propose. Les boutons disponibles sont prédéfinis par la boîte de confirmation. La fonction `showConfirmDialog` provoque l'affichage de cette boîte de dialogue. Elle retourne un entier permettant d'identifier le bouton utilisé pour fermer la boîte de dialogue, et donc obtenir la réponse de l'utilisateur. Elle accepte les paramètres suivants :

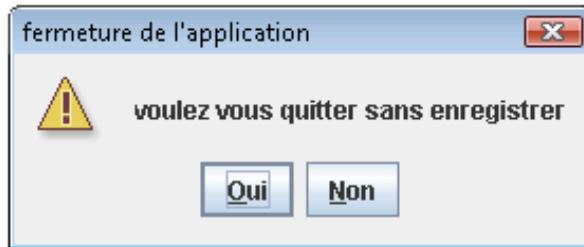
- le mode d'affichage : fenêtre autonome ou fenêtre interne à un composant. Dans ce cas, il faut fournir comme premier paramètre le composant concerné et la valeur `null` si la fenêtre est autonome.
- le message affiché sur la boîte de message. C'est le deuxième paramètre qui est utilisé à cet effet.
- le titre de la boîte de message comme troisième paramètre.
- la combinaison de boutons affichés. Les constantes `YES_NO_OPTION`, `YES_NO_CANCEL_OPTION` ou `OK_CANCEL_OPTION` sont

utilisables pour ce paramètre.

- le type d'icône affichée sur la boîte de saisie. Une série de constantes définies dans la classe permet d'effectuer ce choix. Ces constantes sont identiques à celles utilisables pour la boîte de saisie.

Cette fonction retourne un entier qu'il faut comparer aux constantes suivantes YES_OPTION,OK_OPTION, NO_OPTION, CANCEL_OPTION ou CLOSE_OPTION pour déterminer la réponse de l'utilisateur.

```
int reponse;  
reponse=JOptionPane.showConfirmDialog(null,"voulez vous quitter  
sans enregistrer",  
    "fermeture de l'application", JOptionPane.YES_NO_OPTION,  
    JOptionPane.WARNING_MESSAGE);
```



Principe de fonctionnement

Une applet est un type d'application Java spécifique permettant d'exécuter du code Java à l'intérieur d'une page web. Le but principal des applets est d'ajouter interactivité et dynamisme dans une page web. L'appel de l'applet est incorporé dans le code html de la page. Lorsque le navigateur analyse la page html que le serveur web vient de lui transmettre et rencontre une balise correspondant à une applet, il télécharge le code de l'applet et démarre l'exécution de ce code. L'avantage principal d'une applet par rapport à une application est incontestablement l'absence d'installation nécessaire sur les postes clients. D'autres avantages sont également appréciables lors de l'utilisation d'une applet :

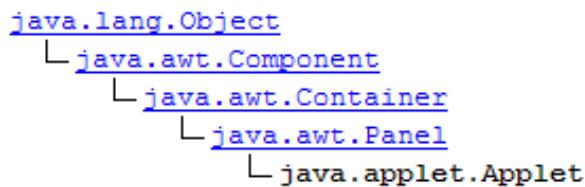
- Les clients disposeront toujours de la dernière version du code. À chaque chargement de la page web par le navigateur, le code de l'applet est également téléchargé à partir du serveur web. Si une nouvelle version du code est disponible, celle-ci doit uniquement être déployée sur le serveur web pour que tous les clients puissent en bénéficier.
- Toutes les ressources utiles pour le bon fonctionnement de l'application seront disponibles. Comme le code de l'applet, les ressources dont elle a besoin sont également transférées du serveur web vers le navigateur. Elles doivent donc uniquement être présentes sur le serveur web.
- L'exécution du code de l'applet ne risque pas de provoquer de dégâts sur le poste client. Lorsque la machine virtuelle Java du navigateur prend en charge l'exécution du code elle le considère comme potentiellement dangereux et l'exécute avec un ensemble limité de droits.

Les applets comportent cependant un petit désavantage puisqu'elles ne peuvent pas être exécutées de manière autonome par la machine virtuelle Java du jdk. Elles doivent obligatoirement être incorporées dans une page html pour être prises en charge par la machine virtuelle Java d'un navigateur. Le navigateur doit bien sûr disposer d'une telle machine virtuelle.

Création d'une applet

Pour qu'une applet puisse être prise en charge par un navigateur, elle doit disposer de caractéristiques bien précises. La solution utilisée pour être certain que l'applet dispose de ces caractéristiques consiste à créer une classe héritant d'une classe disposant déjà de ces caractéristiques.

La personnalisation est réalisée en substituant dans la classe ainsi créée, certaines méthodes héritées de la classe de base. Les classes Applet et JApplet sont utilisables comme classes de base pour la création d'une applet. La classe JApplet définie dans le package javax.swing permet l'utilisation de composants de ce même package pour la construction de l'interface utilisateur de l'applet. Pour les applets assurant elles-mêmes la gestion de leur aspect graphique ou utilisant les composants de la bibliothèque awt, la classe Applet doit être utilisée. La classe JApplet hérite de la classe Applet, elle étend donc ces fonctionnalités. La classe Applet fait également partie d'une importante hiérarchie.



Du fait de cette hiérarchie, une applet est donc un objet graphique qui est pris en charge par le navigateur qui l'affiche.

1. Cycle de vie d'une applet

Lorsque le navigateur prend en charge une applet, il exécute certaines de ces méthodes en fonction des circonstances. Ces méthodes peuvent être classées en deux catégories.

- méthodes liées au cycle de vie de l'applet.
- méthodes de gestion de l'aspect graphique de l'applet.

Pour concevoir des applets performantes il faut bien comprendre quand sont appelées ces méthodes et qu'est-ce que l'on peut en attendre. L'implémentation de certaines de ces méthodes dans la classe Applet est vide. Il est donc indispensable de les substituer dans notre classe.

a. Méthodes liées au cycle de vie de l'applet

```
public void init()
```

Cette méthode est exécutée dès la fin du chargement ou du rechargement de l'applet à partir du serveur web. C'est en fait la première méthode de l'applet qui est exécutée par le navigateur. Elle permet d'initialiser le contexte dans lequel va fonctionner l'applet.

On effectue notamment dans cette méthode les traitements suivants :

- création des instances des autres classes utiles au fonctionnement de l'applet ;
- initialisation des variables ;
- création des threads ;
- chargement des images utilisées par l'applet ;
- récupération des paramètres passés depuis la page html.

Dans une application classique, ces opérations sont en général exécutées dans le constructeur de la classe.

```
public void start()
```

Cette méthode est appelée par le navigateur après la phase d'initialisation de l'applet réalisée par la méthode `init`. Elle est également appelée par la suite à chaque fois que la page sur laquelle elle est insérée est réaffichée par le navigateur. C'est par exemple le cas si l'utilisateur change de page et revient ensuite sur la page précédente (celle contenant l'applet). Certains navigateurs appellent également à nouveau la méthode `init`. Elle correspond à l'étape de démarrage ou de redémarrage de l'applet. On y trouve par exemple le code permettant de démarrer ou redémarrer les threads créés dans la méthode `init`.

```
public void stop()
```

Cette méthode est utilisée lors de la phase d'arrêt de l'applet. Cette phase intervient principalement lorsque l'utilisateur change de page. Elle peut également être exécutée lorsque l'exécution de l'applet se termine normalement. C'est dans ce cas l'applet elle-même qui doit appeler cette méthode. La fermeture du navigateur provoque également l'exécution de cette méthode juste avant l'appel de la méthode `destroy`.

```
public void destroy()
```

Cette méthode est la dernière du cycle de vie d'une applet. Le navigateur l'appelle juste avant sa fermeture. Elle a le même rôle que le destructeur d'une classe (méthode `finalize`). Elle est utilisée pour éliminer de la mémoire les objets ayant été créés pendant le fonctionnement de l'applet et principalement ceux ayant été créés au cours de l'exécution de la méthode `init`. La surcharge de cette méthode n'est absolument pas obligatoire car de toute façon le garbage collector intervient pour libérer les ressources mémoire utilisées par l'applet. Il n'est d'ailleurs même par certain que cette méthode puisse être exécutée complètement avant que le navigateur n'arrête la machine virtuelle Java.

b. Méthodes de gestion de l'aspect graphique de l'applet

Il faut considérer deux cas de figure :

Si vous utilisez des composants graphiques, tels que ceux disponibles dans les bibliothèques `awt` et `swing`, le rendu de l'interface de l'applet est assuré automatiquement. Cette possibilité provient des classes présentes dans la hiérarchie de la classe `Applet` ou `JApplet`. C'est dans ce cas la classe `Container` dont héritent entre autres les applets qui effectuent ce travail avec la méthode `paintComponents`.

Si vous assumez entièrement la représentation graphique de l'applet, vous devez surcharger la méthode `paint` pour qu'elle soit capable de gérer l'affichage de l'applet à chaque fois que celle-ci doit être redessinée dans la page web.

Pour visualiser l'ordre d'exécution de ces différentes méthodes, nous allons écrire notre première applet. Nous devons créer une classe héritant de la classe `Applet`. Dans cette classe nous déclarons une variable de type `String` pour mémoriser les passages dans les différentes méthodes. Dans la méthode `paint`, nous affichons la chaîne de caractères sur le contexte graphique de l'applet.

```
import java.applet.Applet;
import java.awt.Graphics;

public class TestApplet extends Applet
{
    private String message="";
    public void destroy()
    {
        message=message + "méthode destroy \r\n";
    }

    public void init()
    {
        message=message + "méthode init \r\n";
    }

    public void start()
```

```

{
    message=message + "méthode start \r\n";
}

public void stop()
{
    message=message + "méthode stop \r\n";
}

public void paint(Graphics g)
{
    message=message + "méthode paint \r\n";
    g.drawString(message, 10, 20);
}

}

```

Pour vérifier le bon fonctionnement de l'applet, nous devons insérer celle-ci dans une page html et visualiser cette dernière dans un navigateur ou avec l'outil **appletViewer** du jdk.

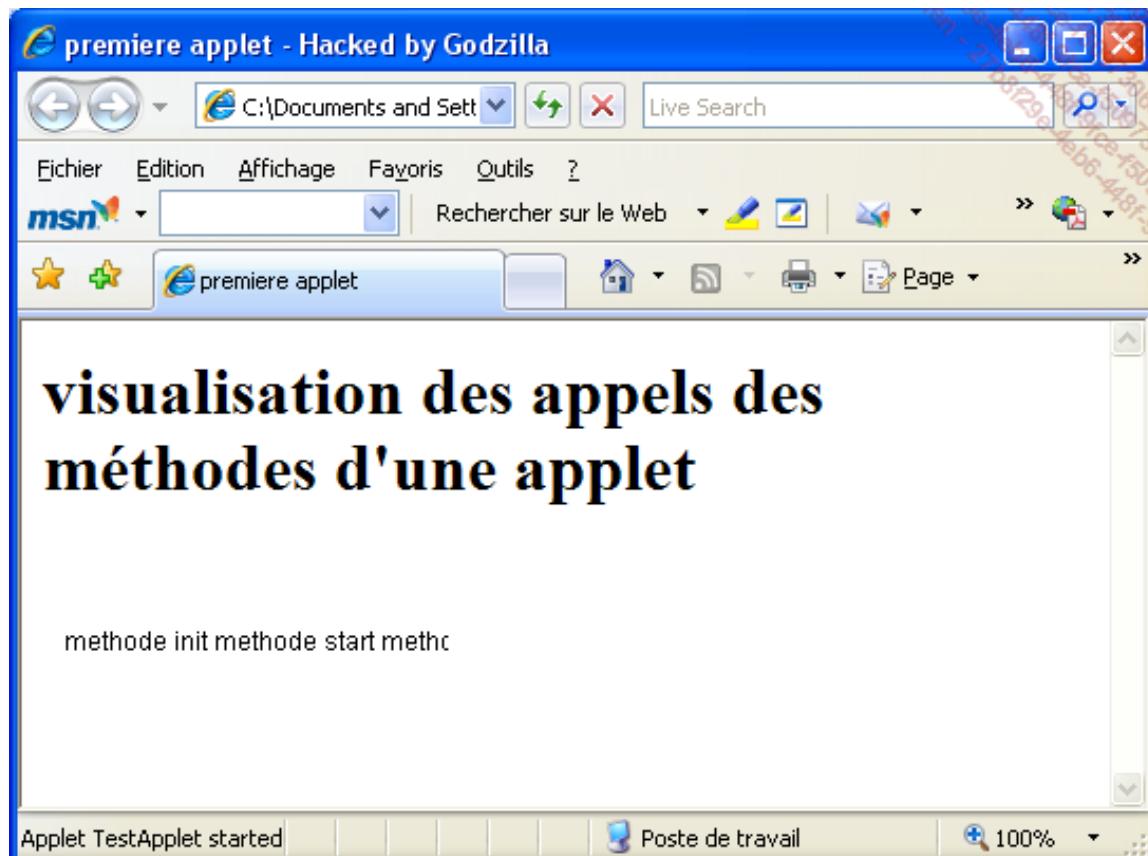
```

<html>
<head>
<title> première applet </title>
</head>
<body>
<h1>visualisation des appels des méthodes d'une applet</h1>
<applet code="TestApplet"></applet>
</body>
</html>

```

La balise `<applet>` indique au navigateur qu'il doit, lors de l'analyse de la page html, charger la classe correspondant à l'applet dont le nom est spécifié par l'attribut `code` de la balise. Cette balise dispose de nombreuses autres possibilités qui seront détaillées plus loin dans ce chapitre.

L'affichage de la page html par un navigateur est présenté ci-dessous.



Nous constatons que la méthode `init` puis la méthode `start` ont été exécutées. Il semble également

qu'au moins une autre méthode ait été exécutée mais nous ne pouvons pas visualiser dans son intégralité le texte affiché. Ce problème est lié au fait que le navigateur réserve un espace pour l'applet dans le document. L'espace nécessaire pour l'affichage de la chaîne étant supérieur à la largeur de l'applet, nous obtenons cet affichage tronqué. Nous verrons que nous pouvons agir sur les dimensions de l'espace alloué à l'applet dans la page html avec les attributs `height` et `width` de la balise mais cette solution ne fera que retarder le problème car nous aurons forcément à un moment ou un autre une chaîne de caractères nécessitant une largeur plus importante que celle dont dispose l'applet. Nous devons donc apporter plus d'attention, lors de la conception de l'applet, à sa représentation graphique.

2. Construire l'interface utilisateur d'une applet

Pour concevoir l'interface utilisateur de l'applet nous avons deux solutions. Nous pouvons utiliser une bibliothèque de composants graphiques (awt ou swing) et utiliser ses composants pour définir l'aspect visuel de l'applet. Cette solution présente l'avantage d'être rapide à mettre en œuvre et convient dans la majorité des cas. Le seul reproche que l'on puisse lui faire est justement d'utiliser des composants classiques pour l'affichage et de ne pas permettre des présentations spécifiques. Toutefois l'amélioration et l'évolution des bibliothèques graphiques procurent de plus en plus de possibilités. Si toutefois vous ne trouvez pas le composant correspondant à vos besoins, vous pouvez assumer complètement la représentation graphique de l'applet en surchargeant sa méthode `paint`. Nous allons dans un premier temps utiliser cette solution en essayant d'éliminer les problèmes rencontrés lors de notre expérimentation précédente. Nous devons veiller à ne pas dessiner au-delà des limites de l'applet. Pour cela nous allons adopter la démarche suivante :

- Créer une police de caractères pour l'affichage ;
- Obtenir les dimensions de la zone réservée à l'applet ;
- Dessiner les caractères les uns à la suite des autres en vérifiant à chaque fois que l'espace disponible est suffisant ;
- Si l'espace n'est pas suffisant sur la ligne actuelle, nous devons gérer le changement de ligne.

Essayons de résoudre les problèmes par étapes.

a. Crédation d'une police de caractères

Une police de caractères est définie par son nom et par sa taille. Il existe une multitude de polices de caractères et rien ne nous garantit que celle que nous allons choisir sera disponible sur l'ordinateur sur lequel notre applet va être exécutée. Pour pallier ce problème, Java propose des polices logiques qui sont au moment de l'exécution converties en polices disponibles sur le poste client. Les polices logiques suivantes sont disponibles :

- SansSerif
- Serif
- Monospaced
- Dialog
- DialogInput

La police réelle utilisée sera déterminée au moment de l'exécution en fonction des disponibilités. Par exemple, la police logique SansSerif sera convertie en police Arial sur un système Windows.

La création de la police est effectuée en appelant le constructeur de la classe `Font` et en lui passant comme paramètres le nom de la police, le style de la police (gras, italique...), la taille de la police.

```
Font police=null;
police =new Font("SansSerif",Font.PLAIN,14);
```

b. Obtenir les dimensions de l'applet

Les dimensions de l'applet peuvent être fixées par le navigateur ou par le concepteur de la page html lorsque celui-ci y insère l'applet. Ces dimensions peuvent être obtenues au moment de l'exécution en utilisant les méthodes `getHeight` et `getWidth`.

```
int largeur;
int hauteur;
largeur=getWidth();
hauteur=getHeight();
```

c. Dessiner les caractères

La méthode `drawString` de la classe `Graphics` permet l'affichage d'une chaîne de caractères avec la police spécifiée au préalable par la méthode `setFont`. La méthode `drawString` attend comme paramètres :

- la chaîne de caractères ;
- la position horizontale où est effectué l'affichage ;
- la position verticale où est effectué l'affichage.

Cette méthode ne gère pas de notion de curseur et il nous incombe donc l'obligation de fournir à chaque appel une position correcte pour l'affichage.

```
g.drawString(texte,positionX,positionY);
```

d. Déterminer les dimensions d'une chaîne

Pour pouvoir gérer correctement l'affichage, nous devons vérifier qu'il reste suffisamment d'espace disponible pour la chaîne à afficher. Cet espace dépend bien sûr du nombre de caractères de la chaîne et également de la police utilisée. De plus, certaines polices ayant un espacement proportionnel, il faut plus d'espace pour afficher le caractère W que pour afficher le caractère i. Il n'est donc pas envisageable de se baser sur une largeur constante pour chaque caractère. La classe `FontMetrics` nous apporte une aide précieuse pour résoudre ce problème. Un objet de `FontMetrics` peut être obtenu en utilisant la méthode `getFontMetrics` du contexte graphique de l'applet. Cette méthode attend comme paramètre la police de caractères avec laquelle elle va effectuer ses calculs. La largeur nécessaire pour afficher une chaîne de caractères donnée est ensuite obtenue en appelant la méthode `stringWidth` à laquelle il faut fournir la chaîne de caractères à "mesurer". La hauteur est elle obtenue avec la méthode `getHeight`. La hauteur étant uniquement liée à la police de caractères, il n'est pas nécessaire de fournir la chaîne de caractères pour obtenir cette information.

Maintenant que nous avons tous les éléments utiles, voici comment les utiliser pour résoudre notre problème.

```
import java.awt.Font;
import java.awt.FontMetrics;
import java.awt.Graphics;
import javax.swing.JApplet;

public class TestApplet2 extends JApplet
{
    private String message="";
    public void destroy()
    {
        message=message + "méthode destroy \r\n";
    }

    public void init()
    {
        message=message + "méthode init \r\n";
    }

    public void start()
```

```

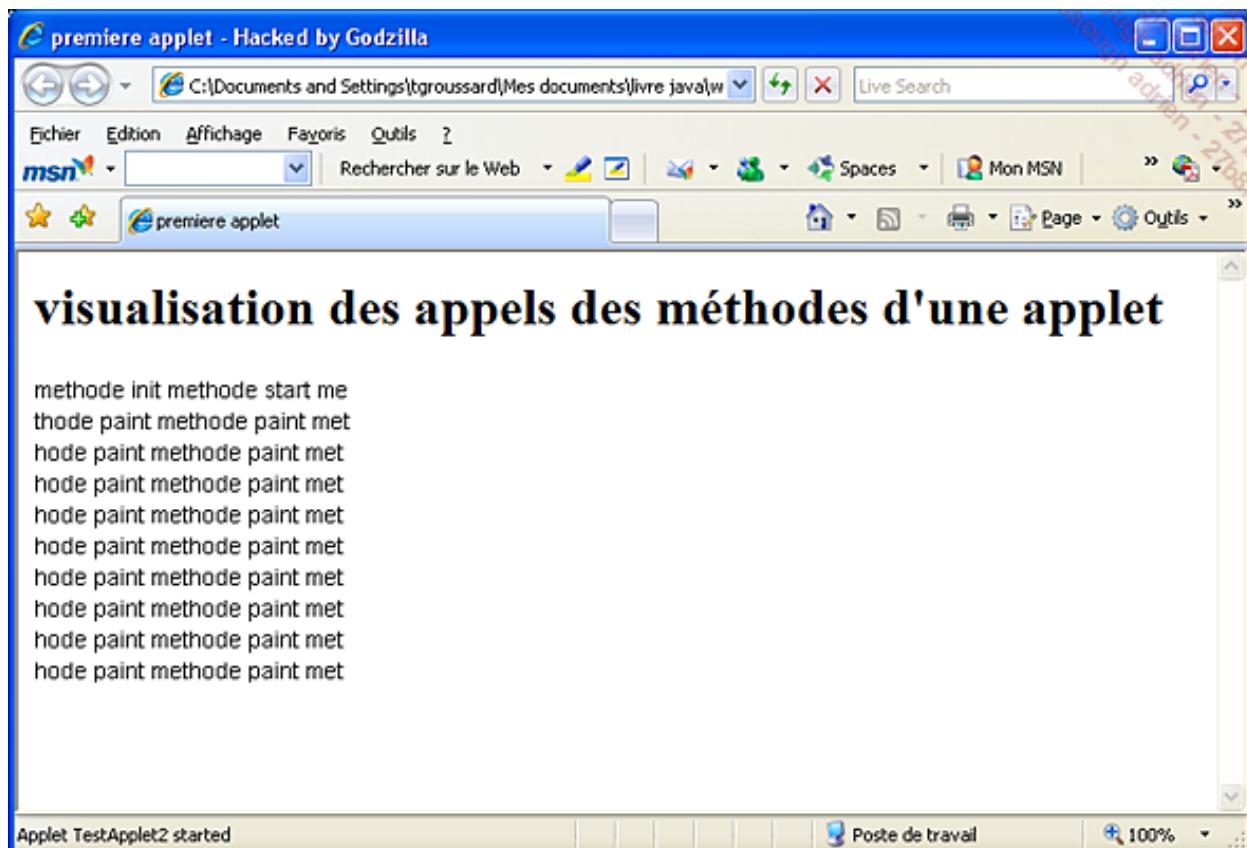
{
    message=message + "méthode start \r\n";
}

public void stop()
{
    message=message + "méthode stop \r\n";
}

public void paint(Graphics g)
{
// création de la police de caractères
    Font police=null;
    police =new Font("SansSerif",Font.PLAIN,14);
// affectation de la police au contexte graphique
    g.setFont(police);
// détermination de la largeur et de la hauteur de l'applet
    int largeurApplet;
    int hauteurApplet;
    largeurApplet=getWidth();
    hauteurApplet=getHeight();
// création d'un objet FontMetrics pour obtenir des informations
// sur la taille des caractères
    FontMetrics fm;
    fm=g.getFontMetrics(police);
    int hauteurPolice;
    int largeurCaractere;
    hauteurPolice=fm.getHeight();
// déclaration des variables pour gérer la position d'affichage
    int curseurX;
    int curseurY;
    curseurX=0;
    curseurY=hauteurPolice;
    message=message + "methode paint \r\n";
// boucle de traitement des caractères un à un
    for (int i=0;i<message.length();i++)
    {
// récupération du caractère à traiter et de sa largeur
        String caractereCourant;
        caractereCourant=message.substring(i,i+1);
        largeurCaractere=fm.stringWidth(caractereCourant);
// vérification s'il reste de la place sur la ligne
        if (curseurX+largeurCaractere>largeurApplet)
        {
            // passage au début de la ligne suivante
            curseurY=curseurY+hauteurPolice;
            curseurX=0;
        }
// affichage du caractère à la position courante
        g.drawString(caractereCourant,curseurX,curseurY);
// mise à jour de la position du curseur
        curseurX=curseurX+largeurCaractere;
    }
}
}

```

En vérifiant le fonctionnement de l'applet, nous constatons qu'il y a une nette amélioration par rapport à la version précédente.



Il y a bien encore quelques améliorations à apporter notamment pour la gestion du défilement vertical mais nous allons plutôt essayer une autre solution en confiant l'aspect graphique de l'applet à des composants spécialisés. L'affichage avec défilement de texte est une fonctionnalité courante que doit proposer une application et les concepteurs des bibliothèques graphiques de Java ont bien sûr conçus des composants adaptés à ces besoins. Les classes `TextArea` et `ScrollPane` vont nous permettre de résoudre notre problème. La mise en œuvre est extrêmement simple puisqu'il suffit de créer une instance de la classe `TextArea` en indiquant dans l'appel du constructeur les dimensions souhaitées. Cette instance doit ensuite être confiée au composant `ScrollPane` qui va assurer le défilement du texte. L'ensemble est ensuite ajouté sur l'applet. L'ajout de texte est effectué très simplement en appelant la méthode `append` de la classe `TextArea`. Notre nouvelle version d'applet prend donc la forme suivante.

```
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.ScrollPane;
import java.awt.TextArea;

public class TestApplet3 extends Applet
{
    ScrollPane defil;
    TextArea txt;
    private String message="";

    public void destroy()
    {
        message=message + "méthode destroy \r\n";
    }

    public void init()
    {
        txt=new TextArea(50,50);
        defil=new ScrollPane();
        defil.add(txt);
        add(defil);
        txt.append("méthode init \r\n");
    }
}
```

```

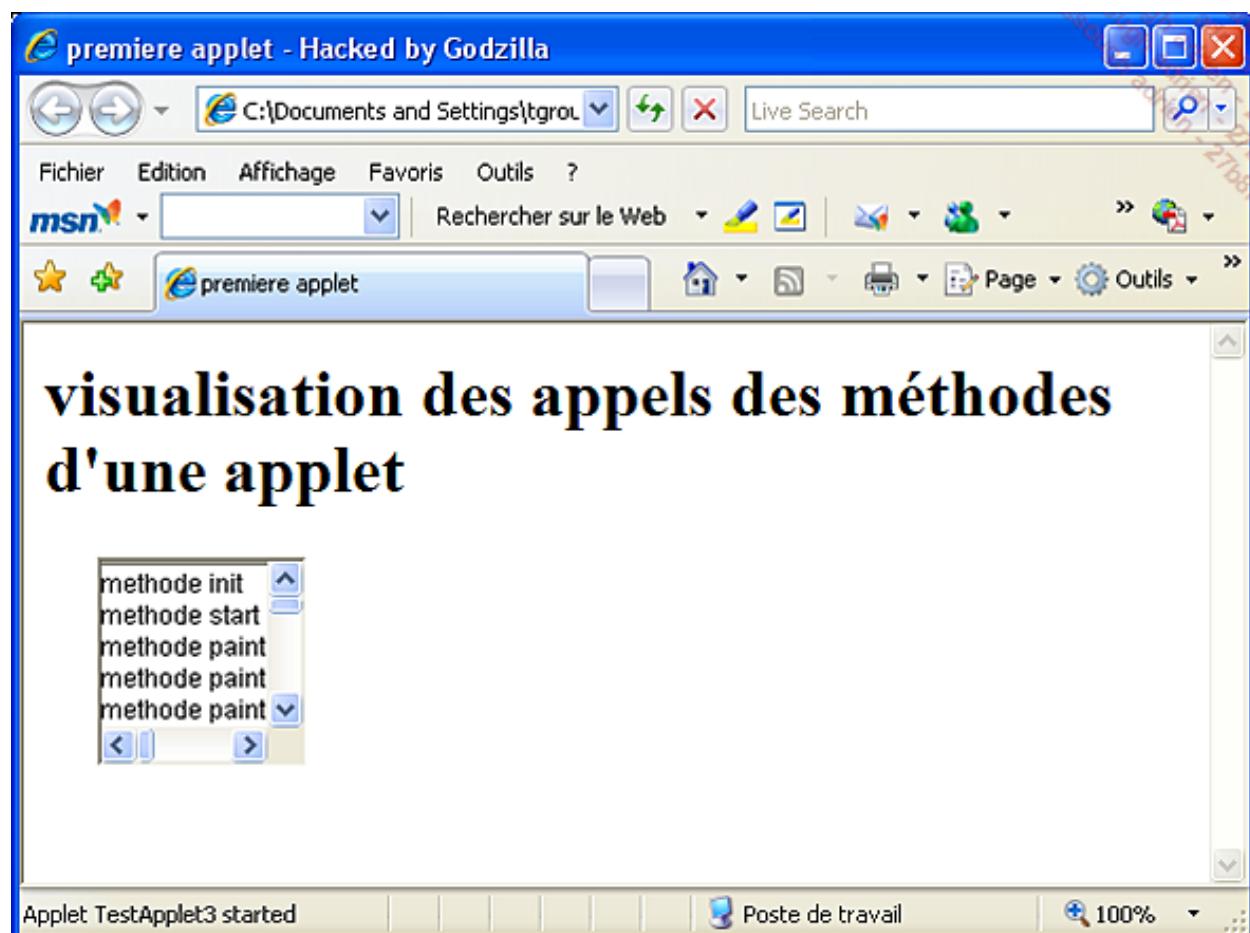
public void start()
{
    txt.append("méthode start \r\n");
}

public void stop()
{
    txt.append("méthode stop \r\n");
}

public void paint(Graphics g)
{
    txt.append("méthode paint \r\n");
}
}

```

Le code est nettement plus simple que celui de la version précédente et pourtant le résultat est beaucoup plus professionnel.



3. Les images dans les applets

L'affichage d'une image dans une applet s'effectue en trois opérations distinctes :

- le chargement de l'image à partir d'un fichier ;
- le traitement éventuel de l'image ;
- le traçage de l'image sur le contexte graphique de l'applet.

Nous allons détailler chacune de ces opérations.

a. Chargement d'une image

La classe Applet propose deux versions de la méthode getImage permettant le chargement d'une image à partir d'un fichier gif ou jpeg. La première version attendant comme paramètre l'URL complète du fichier contenant l'image. La seconde attend comme premier paramètre également une URL et comme deuxième paramètre un nom relatif à cette URL permettant l'accès au fichier contenant l'image.

Les deux solutions suivantes sont utilisables.

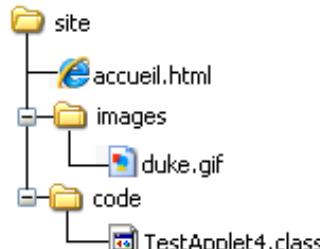
```
private Image img;  
img=getImage(new URL("http://www.eni-ecole.fr/images/cole.jpg"));  
  
private Image img;  
img=getImage(new URL("http://www.eni-ecole.fr/"), "images/cole.jpg");
```

Ces deux solutions ont pour énorme inconvénient d'utiliser des chemins absous pour les URLs. Il est très probable qu'au moment de la conception de l'applet, on ne sait pas encore sur quel serveur et sous quel nom va être déployée l'application. Il est donc impossible d'utiliser directement cette information dans le code. Par contre, nous pouvons être certains que si notre code s'exécute c'est que le navigateur l'a téléchargé. Nous pouvons donc questionner l'applet pour qu'elle nous indique l'emplacement à partir duquel le navigateur l'a téléchargé. La méthode getCodeBase nous permet d'obtenir l'URL du répertoire contenant l'applet sur le serveur. Il ne faut cependant pas oublier que si l'applet est disponible dans le navigateur c'est que celui-ci a au préalable téléchargé une page html contenant l'applet. L'URL de cette page est obtenue par la méthode getDocumentBase.

Il est donc possible d'adresser le fichier contenant l'image à partir de l'un ou l'autre de ces emplacements. C'est l'organisation de l'application qui va nous dicter la bonne méthode.

- Si l'image se trouve dans le répertoire, ou un sous-répertoire, où est placée la page html vous devez utiliser la méthode getDocumentBase.
- Si l'image se trouve dans le répertoire, ou un sous-répertoire, où est placée le code de l'applet vous devez utiliser la méthode getCodeBase.

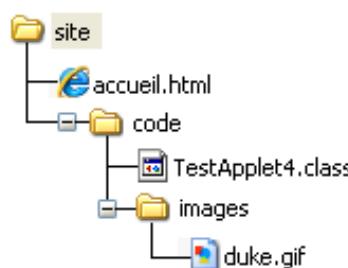
Avec l'organisation suivante :



L'image est accessible dans l'applet avec le code suivant :

```
img=getImage(getDocumentBase(),"images/duke.gif");
```

Avec cette autre organisation :



L'image est accessible avec ce code :

```
img=getImage(getCodeBase(),"images/duke.gif");
```

b. Traitement de l'image

Si des traitements spécifiques doivent être effectués avec l'image avant son affichage, vous pouvez obtenir des informations sur celle-ci par l'intermédiaire des méthodes de la classe `Image`.

Method Summary	
void	flush() Flushes all reconstructable resources being used by this Image object.
float	getAccelerationPriority() Returns the current value of the acceleration priority hint.
ImageCapabilities	getCapabilities(GraphicsConfiguration gc) Returns an ImageCapabilities object which can be inquired as to the capabilities of this Image on the specified GraphicsConfiguration.
abstract Graphics	getGraphics() Creates a graphics context for drawing to an off-screen image.
abstract int	getHeight(ImageObserver observer) Determines the height of the image.
abstract Object	getProperty(String name, ImageObserver observer) Gets a property of this image by name.
Image	getScaledInstance(int width, int height, int hints) Creates a scaled version of this image.
abstract ImageProducer	getSource() Gets the object that produces the pixels for the image.
abstract int	getWidth(ImageObserver observer) Determines the width of the image.
void	setAccelerationPriority(float priority) Sets a hint for this image about how important acceleration is.

c. Traçage de l'image

C'est en fait le contexte graphique de l'applet qui va réellement prendre en charge l'affichage de l'image sur l'applet. Pour cela, la classe `Graphics` fournit six versions différentes de la méthode `drawImage`.

Affiche l'image aux coordonnées x et y.

Affiche l'image aux coordonnées *x* et *y* et remplit les portions transparentes de l'image avec la couleur spécifiée par *bgcolor*. Le résultat est équivalent à un affichage de l'image sur un rectangle ayant pour couleur *bgcolor*.

Affiche l'image aux coordonnées `x` et `y` et redimensionne l'image avec la largeur `width` et la hauteur `height`.

```
public abstract boolean drawImage(Image img,
                                  int x,
                                  int y,
                                  int width,
                                  int height,
                                  Color bgcolor,
                                  ImageObserver observer)
```

Affiche l'image aux coordonnées `x` et `y` et redimensionne l'image avec la largeur `width` et la hauteur `height` et remplit les portions transparentes de l'image avec la couleur spécifiée par `bgcolor`.

```
public abstract boolean drawImage(Image img,
                                  int dx1,
                                  int dy1,
                                  int dx2,
                                  int dy2,
                                  int sx1,
                                  int sy1,
                                  int sx2,
                                  int sy2,
                                  ImageObserver observer)
```

Affiche la partie de l'image délimitée par le rectangle formé par `(sx1, sy1)` et `(sx2, sy2)` dans le rectangle formé par `(dx1, dy1)` et `(dx2, dy2)`. La partie de l'image à dessiner est redimensionnée automatiquement pour remplir exactement le rectangle de destination.

```
public abstract boolean drawImage(Image img,
                                  int dx1,
                                  int dy1,
                                  int dx2,
                                  int dy2,
                                  int sx1,
                                  int sy1,
                                  int sx2,
                                  int sy2,
                                  Color bgcolor,
                                  ImageObserver observer)
```

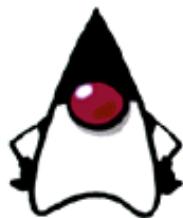
Affiche la partie de l'image délimitée par le rectangle formé par `(sx1, sy1)` et `(sx2, sy2)` dans le rectangle formé par `(dx1, dy1)` et `(dx2, dy2)`. La partie de l'image à dessiner est redimensionnée automatiquement pour remplir exactement le rectangle de destination et remplit les portions transparentes de l'image avec la couleur spécifiée par `bgcolor`.

Pour toutes ces méthodes, une instance de classe implémentant l'interface `ImageObserver` doit être fournie. Cette instance de classe est utilisée pour suivre l'évolution du chargement de l'image.

Voici ci-dessous quelques exemples de code et l'affichage correspondant.

```
img=getImage(getCodeBase(),"images/duke.gif");
g.drawImage(img,0,0,this);
```

Duke la mascotte



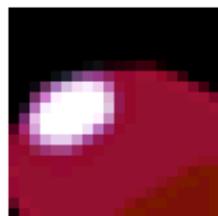
```
img=getImage(getCodeBase(),"images/duke.gif");
g.drawImage(img,0,0,20,20,this);
g.drawImage(img,30,0,50,50,this);
g.drawImage(img,90,0,100,100,this);
```

la famille Duke



```
img=getImage(getCodeBase(),"images/duke.gif");
g.drawImage(img,0,0,100,100,30,30,50,50,this);
```

gros plan sur le nez de Duke

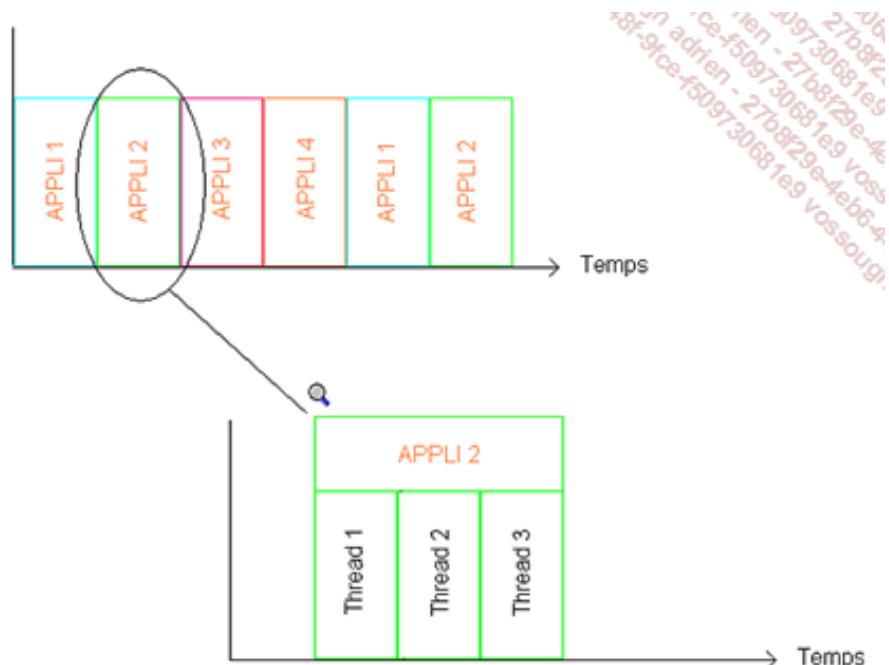


4. Les threads dans les applets

Avant d'étudier comment créer un thread, nous allons tout d'abord définir ce terme. La majorité des machines disposent d'un seul processeur pour assurer leurs fonctionnements. Ce processeur ne peut bien sûr réaliser qu'une seule opération élémentaire à la fois et pourtant avec la plupart des systèmes d'exploitation nous pouvons exécuter plusieurs applications simultanément. En fait nous avons l'impression que plusieurs applications s'exécutent simultanément mais en réalité il n'en n'est rien. Pour simuler cette simultanéité, le système d'exploitation "découpe" le temps processeur disponible en très fines tranches et distribue ces tranches aux différentes applications. La commutation étant très rapide nous avons

l'impression que toutes les applications s'exécutent simultanément. La programmation multithread applique ce même principe à une application.

Si nous prenons l'exemple d'un traitement de texte, celui-ci peut par exemple effectuer plusieurs opérations simultanément (saisie de texte, vérification de l'orthographe, mise en page, impression...). Pour pouvoir effectuer tous ces traitements, l'application réalise à nouveau un découpage de chaque tranche de temps processeur que lui fournit le système d'exploitation et affecte à chaque tâche à réaliser une portion de cette tranche.



Par défaut, une application ou une applet comporte un thread responsable de l'exécution de votre code. Celui-ci est parfois nommé *thread principal*. L'ajout d'un ou de plusieurs threads supplémentaires est parfois utile pour conserver une bonne réactivité de l'application. Dans le cas d'une applet, nous pouvons par exemple ajouter un thread pour effectuer une opération d'initialisation relativement longue. Si cette opération est effectuée par la méthode `init` de l'applet, le navigateur devra attendre la fin de cette méthode pour poursuivre le démarrage de l'applet. Par contre, si dans la méthode `init` nous démarrons un nouveau thread pour effectuer l'opération d'initialisation, la suite du démarrage de l'applet sera exécutée plus rapidement pendant que le thread continuera d'effectuer l'initialisation. Cette technique est très utile pour une applet utilisant des fichiers sons. Le fichier peut ainsi être téléchargé pendant que se poursuit l'initialisation de l'applet.

Les threads sont également très utiles lorsqu'une applet doit réaliser un traitement répétitif. Nous pouvons confier à un nouveau thread le soin d'exécuter ce traitement sans perturber le fonctionnement du reste de l'applet.

L'utilisation des threads nécessite trois étapes dans la conception de l'application :

- créer un nouveau thread ;
- définir le traitement que doit effectuer le nouveau thread ;
- gérer les démarriages et arrêts du thread.

Voici le détail de ces trois étapes.

a. Crédation d'un nouveau thread

La classe `Thread` permet la création et la gestion d'un thread. Comme pour n'importe quelle autre classe, il faut créer une instance par l'intermédiaire d'un des constructeurs disponibles. Il est à noter que la création d'une instance de la classe `Thread` ne déclenche pas le démarrage de l'exécution du thread.

b. Définir le traitement à effectuer

Lorsqu'un thread est démarré, il exécute automatiquement sa méthode `run`. Par défaut cette méthode est définie dans la classe `Thread` mais ne contient aucun code. Il est donc obligatoire de redéfinir cette méthode. Ceci peut être fait en créant une classe héritant de la classe `Thread` et en redéfinissant dans celle-ci la méthode `run`.

```
public class ThreadPerso extends Thread
{
    public void run()
    {
        // code à exécuter par le thread
    }
}
```

C'est bien sûr dans ce cas une instance de cette classe qu'il faudra créer.

Vous pouvez également obtenir le même résultat en créant une instance de classe interne anonyme.

```
Thread t;
t=new Thread(){
    public void run()
    {
        // code à exécuter par le thread
    }
};
```

La dernière solution consiste à indiquer au thread que la méthode `run` qu'il doit exécuter se trouve dans une autre classe. Il faut dans ce cas fournir une instance de cette classe lors de l'appel du constructeur de la classe `Thread`. Pour être certain que la classe utilisée contienne bien une méthode `run`, celle-ci devra implémenter l'interface `Runnable`. Ceci peut être pris en charge par la classe de l'applet.

```
public class TestApplet5 extends Applet implements Runnable
{
    Thread th;
    public void run()
    {
        // code à exécuter par le thread
    }

    public void start()
    {
        th=new Thread(this);
    }
}
```

Il reste maintenant à définir le contenu de la méthode `run`. Deux cas de figure sont possibles :

- La méthode `run` exécute un traitement unique dans ce cas elle est conçue comme une méthode classique et son exécution se terminera avec la dernière instruction de cette méthode.
- La méthode `run` exécute un traitement cyclique et dans ce cas elle doit contenir une boucle se terminant à la disparition du thread l'exécutant. On peut par exemple utiliser la syntaxe suivante pour cette boucle.

```
public void run()
{
    Thread t ;
```

```

t= Thread.currentThread() ;

while(th==t)
{
    // code à exécuter par le thread
}
}

```

La méthode statique `currentThread` permettant d'obtenir une référence sur le thread courant, cette boucle se terminera dès que la variable `th` ne référencera plus le thread courant. Ce sera par exemple le cas si l'on affecte la valeur `null` à cette variable.

Il est parfois nécessaire de contrôler la fréquence d'exécution des instructions de la boucle en y insérant un appel à la méthode statique `sleep` de la classe `Thread`. Cette méthode "endort" le thread courant pendant le nombre de millisecondes qui lui est passé comme paramètre. L'appel de cette méthode doit être protégé par un bloc `try catch`.

```

Thread t ;
t= Thread.currentThread() ;
while(th==t)
{
    // code à exécuter par le thread
    // endort le thread pendant 500 ms
    try
    {
        Thread.sleep(500);
    }
    catch (InterruptedException e){}
}

```

c. Démarrer et arrêter un thread

Le démarrage d'un thread est déclenché par un appel à sa méthode `start`. C'est cet appel qui provoque l'exécution de la méthode `run` du thread. Il ne faut surtout pas appeler directement la méthode `run` du thread car elle serait alors exécutée par le thread courant (le principal) de l'applet. Ceci réduirait à néant tous nos efforts et pourrait même provoquer le blocage de l'applet si la méthode `run` contient une boucle (le code permettant de faire évoluer la condition de sortie de boucle ne pouvant plus s'exécuter puisque dans ce cas, l'exécution de la méthode `run` monopolise le thread principal).

L'arrêt du thread est provoqué par la fin de l'exécution de sa méthode `run` soit parce que la dernière instruction qu'elle contient est terminée ou parce que la boucle qu'elle contient est terminée. La méthode `stop`, pourtant présente dans la classe `Thread`, ne doit pas être utilisée car elle présente des risques de blocage de l'application.

Pour illustrer l'utilisation des threads voici une applet permettant de faire grossir puis maigrir Duke en continu.

```

import java.applet.Applet;
import java.awt.Graphics;
import java.awt.Image;

public class TestApplet5 extends Applet implements Runnable
{
    Thread th;
    final int MAXI=100;
    final int MINI=10;
    int largeur=MINI;
    int hauteur=MINI;
    Image img;

    public void run()
    {
        boolean grossir=true;
        Thread t;
        t=Thread.currentThread();
        while (th==t)

```

```

        {
            if(largeur>MAXI & grossir)
            {
                grossir=false;
            }
            if (largeur<MINI & !grossir)
            {
                grossir=true;
            }
            if (grossir)
            {
                largeur++;
                hauteur++;
            }
            else
            {
                largeur--;
                hauteur--;
            }
            repaint();
            try
            {
                th.sleep(10);
            }
            catch (InterruptedException e)
            {}
        }
    }

    public void init()
    {
        img=getImage(getCodeBase(),"images/duke.gif");
    }

    public void start()
    {
        th=new Thread(this);
        th.start();
    }

    public void stop()
    {
        th=null;
    }

    public void paint(Graphics g)
    {
        g.drawImage(img,30,0,largeur,hauteur,this);
    }
}

```

5. Les sons dans les applets

La classe Applet propose deux méthodes permettant le chargement d'un fichier audio. La méthode `getAudioClip` n'est utilisable qu'à partir d'une applet, alors que la méthode `newAudioClip` peut être utilisée dans n'importe quel type d'application puisqu'elle est déclarée `static` dans la classe Applet et de ce fait accessible sans qu'une instance de la classe Applet existe. Ces deux méthodes acceptent comme argument un objet `URL` représentant l'emplacement du fichier audio. Une surcharge de la méthode `getAudioClip` acceptant comme deuxième paramètre une chaîne de caractères est également disponible. Ce deuxième paramètre représente le chemin relatif à l'URL passée comme premier paramètre pour obtenir le fichier audio. Cette méthode est très utile puisqu'elle permet d'employer les méthodes `getCodeBase` ou `getDocumentBase` pour obtenir l'URL de l'applet ou l'URL de la page html et de spécifier ensuite le chemin d'accès vers le fichier audio par rapport à cette URL. Ces méthodes retournent une instance de classe implémentant l'interface `AudioClip`. Cette interface définit les méthodes `play`, `loop` et `stop` permettant de jouer le fichier audio une fois, en boucle ou de l'arrêter.

Les sons dans les applets doivent être utilisés avec modération et précaution sous peine d'énerver

rapidement l'utilisateur de l'applet. Il faut par exemple prendre en compte que les fichiers audio sont parfois assez volumineux donc assez longs à télécharger. Si la méthode `getAudioClip` est utilisée dans la méthode `init` de l'applet, il faudra attendre la fin du téléchargement du fichier audio pour que l'applet puisse être utilisée. Il est préférable de démarrer, dans cette méthode `init`, un thread chargé de télécharger le fichier audio et de lancer sa restitution à la fin du téléchargement.

Il faut également garder présent à l'esprit que la machine virtuelle Java du navigateur conserve les instances des applets créées jusqu'à la fermeture du navigateur. Si une applet a exécuté la méthode `loop` sur un objet `AudioClip`, le fichier sera restitué en boucle jusqu'à la fermeture du navigateur, même si l'utilisateur navigue sur une autre page que celle contenant l'applet. Il est donc prudent d'appeler la méthode `stop` de l'objet `AudioClip` dans la méthode `stop` de l'applet.

```
import java.applet.Applet;
import java.applet.AudioClip;

public class TestAppletAudio extends Applet implements Runnable
{
    AudioClip ac;
    public void init()
    {
        Thread th;
        th=new Thread(this);
        th.start();
    }

    public void start()
    {
        if (ac!=null)
        {
            ac.loop();
        }
    }
    public void stop()
    {
        if (ac!=null)
        {
            ac.stop();
        }
    }
    public void run()
    {
        ac=getAudioClip(getCodeBase(),"son.wav");
        ac.loop();
    }
}
```

Déployer une applet

Pour que les utilisateurs puissent visualiser une applet, celle-ci doit être intégrée dans une page html. Avant ces insertion dans une page html, il faut au préalable stocker dans une archive jar l'ensemble des fichiers nécessaires à son fonctionnement. Il est également souhaitable de signer l'archive créée pour garantir l'authenticité de l'applet. Ces deux opérations sont décrites dans le chapitre consacré au déploiement.

1. La balise <applet>

Cette balise est la balise standard du html pour l'insertion d'une applet. La configuration de l'applet se fait par l'intermédiaire des nombreux attributs de la balise. Certains de ces attributs sont obligatoires.

- `code="nom de la classe de l'applet"` : cet attribut permet d'indiquer au navigateur le nom du fichier contenant la classe principale de l'applet. N'oubliez pas en indiquant le nom de ce fichier que nous sommes dans le monde Java et qu'il y a distinction entre minuscules et majuscules. Respectez donc bien la casse pour ce nom de fichier. À défaut d'autres informations, le navigateur essaie de charger ce fichier à partir du même chemin que celui d'où provient la page html.
- `width="largeur en pixels"` : cet attribut indique au navigateur la largeur de la surface qu'il doit réservé sur la page html pour l'affichage de l'applet.
- `height="hauteur en pixels"` : cet attribut indique au navigateur la hauteur de la surface qu'il doit réservé sur la page html pour l'affichage de l'applet.

La syntaxe minimale de base de la balise applet est donc la suivante :

```
<applet code= "duke.class" width="200" height="200"> </applet>
```

Les attributs suivants sont également disponibles pour la balise applet :

- `codebase="chemin d'accès"` : cet attribut indique au navigateur le chemin d'accès au fichier contenant la classe principale de l'applet si celui-ci n'est pas dans le même répertoire que la page html contenant l'applet. Si l'applet fait partie d'un package le nom de celui-ci doit être indiqué avec le nom de la classe et non avec cet attribut.
- `archive="nom d'un fichier jar"` : si l'applet nécessite plusieurs fichiers pour pouvoir fonctionner il est souvent plus efficace de les regrouper dans une archive Java. Le navigateur n'a dans ce cas que ce fichier à télécharger pour obtenir tout ce qu'il lui faut pour faire fonctionner l'applet. Sinon il doit télécharger les fichiers un à un avec la création à chaque téléchargement de fichier d'une nouvelle connexion http vers le serveur. Le fichier doit être généré avec l'utilitaire jardont le fonctionnement est détaillé dans le chapitre consacré au déploiement d'applications.
- `align="constante d'alignement"` : cet attribut indique comment l'applet est alignée par rapport à l'élément qui la suit dans la page html. Huit valeurs de constantes sont disponibles :
 - `left` : l'applet est alignée à gauche de l'élément qui la suit.
 - `right` : l'applet est alignée à droite de l'élément qui la suit.
 - `texttop` : le haut de l'applet est aligné sur le haut de l'élément texte qui suit l'applet.
 - `top` : le haut de l'applet est aligné sur le haut de l'élément qui la suit.
 - `absmiddle` : le milieu de l'applet est aligné sur le milieu de l'élément qui suit l'applet.
 - `middle` : le milieu de l'applet est aligné sur le milieu de la ligne de base du texte qui suit l'applet.
 - `bottom` : le bas de l'applet est aligné sur la ligne de base du texte qui suit l'applet.
 - `absbottom` : le bas de l'applet est aligné sur l'élément le plus bas qui suit l'applet.
- `vspace="espace vertical en pixels"` : cet attribut indique au navigateur l'espace qu'il doit

laisser libre au-dessus et au-dessous de l'applet.

- `hspace="espace horizontal"` : cet attribut indique au navigateur l'espace qu'il doit laisser libre à gauche et à droite de l'applet.

La balise `applet` peut elle-même contenir du texte. Celui-ci n'apparaîtra dans le navigateur que s'il n'est pas capable de prendre en charge l'exécution de l'applet Java.

Exemple de balise `<applet>` :

```
<applet code="TestApplet4"
        width="150"
        height="150"
        align="left"
        vspace="50"
        hspace="50"
        codebase="appletsApplication"
        archive="data.jar"
    >
Votre navigateur ne gère pas les applets
</applet>
```

2. Paramétrage d'une applet

Les applications classiques peuvent recevoir des informations au moment de leur exécution par l'intermédiaire des paramètres passés sur la ligne de commande. Ces paramètres sont utilisables dans le code de l'application grâce au tableau de chaînes de caractères reçu comme argument dans la méthode `main` de l'application. Les applets n'étant pas exécutées à partir de la ligne de commande mais par le navigateur qui analyse une page html, c'est donc à l'intérieur de celle-ci que vont être définis les paramètres. Ceux-ci sont ensuite récupérés par le code de l'applet.

a. Définir les paramètres

Les paramètres sont définis par des balises `<param>` imbriquées à l'intérieur de la balise `<applet>`. Ces balises doivent avoir obligatoirement deux attributs. Le premier, `name`, permet d'identifier le paramètre. Le second, `value`, correspond à la valeur du paramètre.

b. Récupération des paramètres dans l'applet

La valeur d'un paramètre est obtenue par la méthode `getParameter` de la classe `applet`. Celle-ci attend comme argument une chaîne de caractères représentant le nom du paramètre dont on souhaite obtenir la valeur. Cette méthode retourne toujours une chaîne de caractères. Si un paramètre représente un autre type de donnée, il devra être converti explicitement par le code de l'applet en utilisant par exemple les classes `wrapper` (`Integer`, `Float`, `Long`...). S'il n'existe pas dans la balise `<applet>` de paramètre avec le nom indiqué, la méthode `getParameter` retourne une valeur `null`.

L'exécution de l'applet ne doit bien sûr pas être perturbée par cette situation et doit fournir une valeur par défaut pour le paramètre manquant. La récupération des paramètres peut être faite dans la méthode `init` de l'applet comme dans l'exemple suivant :

```
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.Image;

public class TestApplet5 extends Applet implements Runnable
{
    Thread th;
    int MAXI=100;
    int MINI=10;
    int largeur=MINI;
    int hauteur=MINI;
    Image img;
```

```

public void init()
{
    String min;
    String max;
    min=getParameter("minimum");
    if(min!=null)
    {
        MINI=Integer.parseInt(min);
    }
    max=getParameter("maximum");
    if(max!=null)
    {
        MAXI=Integer.parseInt(max);
    }
    MAXI=Integer.parseInt(max);
    img=getImage(getCodeBase(),"images/duke.gif");

}
...
}

```

Le code suivant permet d'insérer cette applet sur une page html en spécifiant une valeur pour les deux paramètres attendus par l'applet.

```

<applet code="TestApplet5" width="250" height="250">
<param name="minimum" value="20">
<param name="maximum" value="250">
</applet>

```

3. Sécurité dans une applet

Le monde d'Internet ne dispose pas d'une réputation de sécurité absolue. C'est encore plus sensible lorsqu'il s'agit de code provenant d'Internet comme c'est le cas pour une applet. Pour garantir la sécurité du système sur lequel s'exécute une applet, les navigateurs instaurent une politique de sécurité assez stricte vis-à-vis d'une applet. Chaque navigateur implémente sa propre politique de sécurité mais en principe les restrictions suivantes s'appliquent à une applet.

- Une applet ne peut pas charger de bibliothèque ni appeler de méthode native comme par exemple une fonction du système d'exploitation. Les applets doivent donc se contenter de leur propre code et des fonctionnalités mises à leur disposition par la machine virtuelle Java qui assure leur exécution.
- Une applet ne peut pas lire un fichier existant sur la machine sur laquelle elle s'exécute. Cette limitation s'applique bien sûr aussi pour l'écriture d'un fichier et sa suppression.
- Une applet ne peut pas établir de connexion réseau avec une autre machine sauf avec la machine d'où elle provient.
- Une applet ne peut pas lancer l'exécution d'une application sur la machine sur laquelle elle-même s'exécute.
- Une applet dispose d'un accès limité aux propriétés du système.
- Lorsqu'une applet affiche une fenêtre pendant son exécution, cette fenêtre est marquée pour bien signaler à l'utilisateur qu'elle provient de l'exécution de l'applet et non d'une application locale.



4. Communication entre applets

Les applets présentes sur une page html ont la possibilité de dialoguer entre elles. Ce dialogue n'est possible qu'à certaines conditions imposées par le navigateur.

- Les applets doivent toutes provenir du même serveur.
- Elles doivent également provenir du même répertoire sur ce serveur (codebase identique).
- Elles doivent aussi être exécutées sur la même page, dans une même fenêtre de navigateur.

Pour qu'une applet puisse établir un dialogue avec une autre applet, celle-ci doit obtenir une référence vers l'applet à contacter. Cette référence peut être obtenue en utilisant la méthode `getApplet` et en fournissant comme argument à cette méthode le nom de l'applet concernée. Il faut bien sûr que l'applet soit nommée au moment de son insertion sur la page html. Un nom peut être affecté à une applet en ajoutant l'attribut `name` à sa balise ou en ajoutant un paramètre appelé `name` à sa balise. Les deux syntaxes suivantes sont identiques :

```
<applet code="TestApplet5" width="250" height="250">
<param name="minimum" value="20">
<param name="maximum" value="250">
<param name="name" value="duke">
</applet>

<applet code="TestApplet4" width="250" height="250" name="duke">
</applet>
```

La méthode `getApplet` est associée au contexte de l'applet sur lequel vous pouvez obtenir une référence par l'intermédiaire de la méthode `getAppletContext`.

Il est prudent de vérifier la valeur renvoyée par cette méthode pour s'assurer que l'applet a bien été trouvée sur la page avant d'y accéder.

```
Applet ap;
ap=getAppletContext().getApplet("duke1");
if (ap!=null)
{
    ap.setBackground(Color.CYAN);
}
```

Cette solution exige bien sûr de connaître le nom avec lequel l'applet est insérée sur la page html.

Une deuxième solution permet de s'affranchir de cette contrainte en obtenant la liste de toutes les applets présentes sur la page html. C'est dans ce cas la méthode `getApplets` qui permet d'obtenir une énumération sur la liste des applets.

```
Applet ap;
```

```

Enumeration e;
e=getAppletContext().getApplets();
while (e.hasMoreElements())
{
    ap=(Applet)e.nextElement();
    if (ap!=this)
    {
        ap.setBackground(Color.CYAN);
    }
    else
    {
        ap.setBackground(Color.PINK);
    }
}

```

Il convient d'être prudent avec cette solution car l'applet à partir de laquelle s'exécute ce code fait bien sûr partie de la liste (elle est elle aussi sur la page html !). Il faut dans ce cas effectuer un test si l'on ne veut pas que le traitement s'applique à l'applet actuelle ou si l'on souhaite effectuer un traitement différent.

5. Interaction avec le navigateur et le système

Les applets sont gérées par le navigateur qui analyse la page html sur laquelle elles sont insérées, elles peuvent donc avoir une interaction avec celui-ci. Elles ont également droit à un accès limité à certaines propriétés du système.

a. Affichage sur la console

Au cours du développement d'une applet, vous allez vous apercevoir rapidement qu'un élément fait cruellement défaut : la console. Elle nous rend en effet de grands services en phase de test d'une application en y faisant afficher des messages pour indiquer qu'une portion de code a bien été exécutée, ou encore, en y faisant afficher le contenu de certaines variables. En fait cette console est bien disponible même pour une applet mais elle est dans ce cas prise en charge par le navigateur. C'est donc par son intermédiaire que vous pouvez obtenir son affichage. Le moyen d'afficher la console est spécifique à chaque navigateur.

L'aspect de la console dépend également du navigateur. Elle peut être une simple fenêtre d'invite de commande en mode texte ou une application graphique comme avec Internet Explorer. Quel que soit son aspect, les deux flux `System.out` et `System.err` sont toujours dirigés vers cette console. Pour notre première applet, nous aurions pu utiliser le code suivant :

```

import java.applet.Applet;
import java.awt.Graphics;

public class TestApplet extends Applet
{
    public void destroy()
    {
        System.out.println("méthode destroy");
    }

    public void init()
    {
        System.out.println("méthode init");
    }

    public void start()
    {
        System.out.println("méthode start");
    }

    public void stop()
    {

```

```

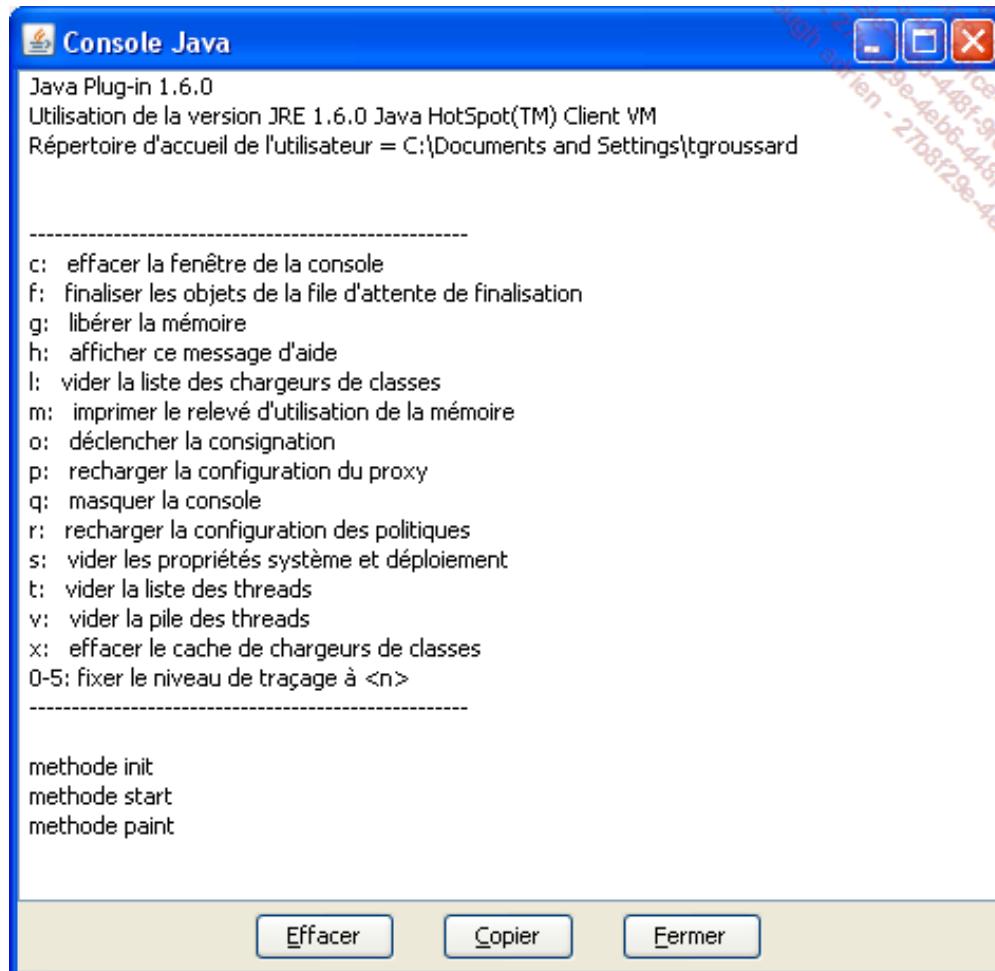
        System.out.println("méthode stop");
    }

    public void paint(Graphics g)
    {
        System.out.println("méthode paint");
    }

}

```

Et ainsi obtenir le résultat suivant sur la console :



b. Utilisation de la barre d'état du navigateur

Vous pouvez également utiliser la barre d'état du navigateur pour l'affichage de messages destinés à l'utilisateur de l'applet. L'accès à la barre d'état se fait grâce à la méthode `showStatus`. Cette méthode reçoit comme argument la chaîne de caractères à afficher. Cette barre d'état n'est parfois pas très visible et surtout son contenu peut être modifié à tout moment par une autre applet ou par le navigateur lui-même.

c. Affichage d'une page html

L'affichage d'un document html est bien sûr la spécialité d'un navigateur. C'est donc à lui que doit s'adresser l'applet si elle a besoin de visualiser un tel document. La méthode `showDocument` du contexte de l'applet permet d'effectuer cette opération. Cette méthode est disponible sous deux formes. La première forme attend comme paramètre l'URL du document à afficher. Le document est affiché à la place de la page html où se trouve l'applet.

```

Try
{

```

```

        getAppletContext().showDocument(new URL("http://www.eni.fr"));
    }
catch (MalformedURLException e)
{
    e.printStackTrace();
}

```

➤ Cette méthode n'accepte pas comme paramètre une chaîne de caractères mais une instance de la classe URL. Cette instance doit être créée à partir de la chaîne de caractères représentant l'URL de la page à afficher. L'utilisation du constructeur de la classe URL exige la présence du bloc try catch ou d'une instruction throws.

La deuxième forme attend une chaîne de caractères comme deuxième paramètre pour identifier l'emplacement où sera affichée la page. Les valeurs suivantes sont acceptées pour cette chaîne.

_blank : le document est affiché dans une nouvelle fenêtre du navigateur.

_self : le document est affiché à la place de la page où se trouve l'applet.

_parent : le document est affiché dans la fenêtre parente de celle où est affichée la page contenant l'applet.

_top : le document est affiché dans le cadre de plus haut niveau.

nomDeCadre : le document est affiché dans le cadre désigné par le nom spécifié. Si ce cadre n'existe pas, une nouvelle fenêtre est créée pour afficher le document.

d. Obtenir certaines propriétés du système

Pour que l'applet puisse s'adapter au mieux à l'environnement dans lequel elle s'exécute, elle a accès à certaines propriétés du système. Ces propriétés sont accessibles par la méthode getProperty de la classe System. Cette méthode accepte comme paramètre une chaîne de caractères indiquant le nom de la propriété dont on souhaite obtenir la valeur. Les propriétés suivantes sont accessibles depuis une applet :

Nom de la propriété	Valeur obtenue
file.separator	Caractère utilisé comme séparateur dans les chemins d'accès aux fichiers
path.separator	Caractère utilisé comme séparateur entre deux chemins d'accès (variable d'environnement path)
java.vendor	Nom du fournisseur du JRE
java.vendor.url	URL du site web du fournisseur du JRE
java.version	Version du JRE
line.separator	Caractère de séparation de lignes
os.arch	Plate-forme du système d'exploitation
os.name	Nom du système d'exploitation

L'applet suivante affiche ces différentes propriétés dans la console Java du navigateur :

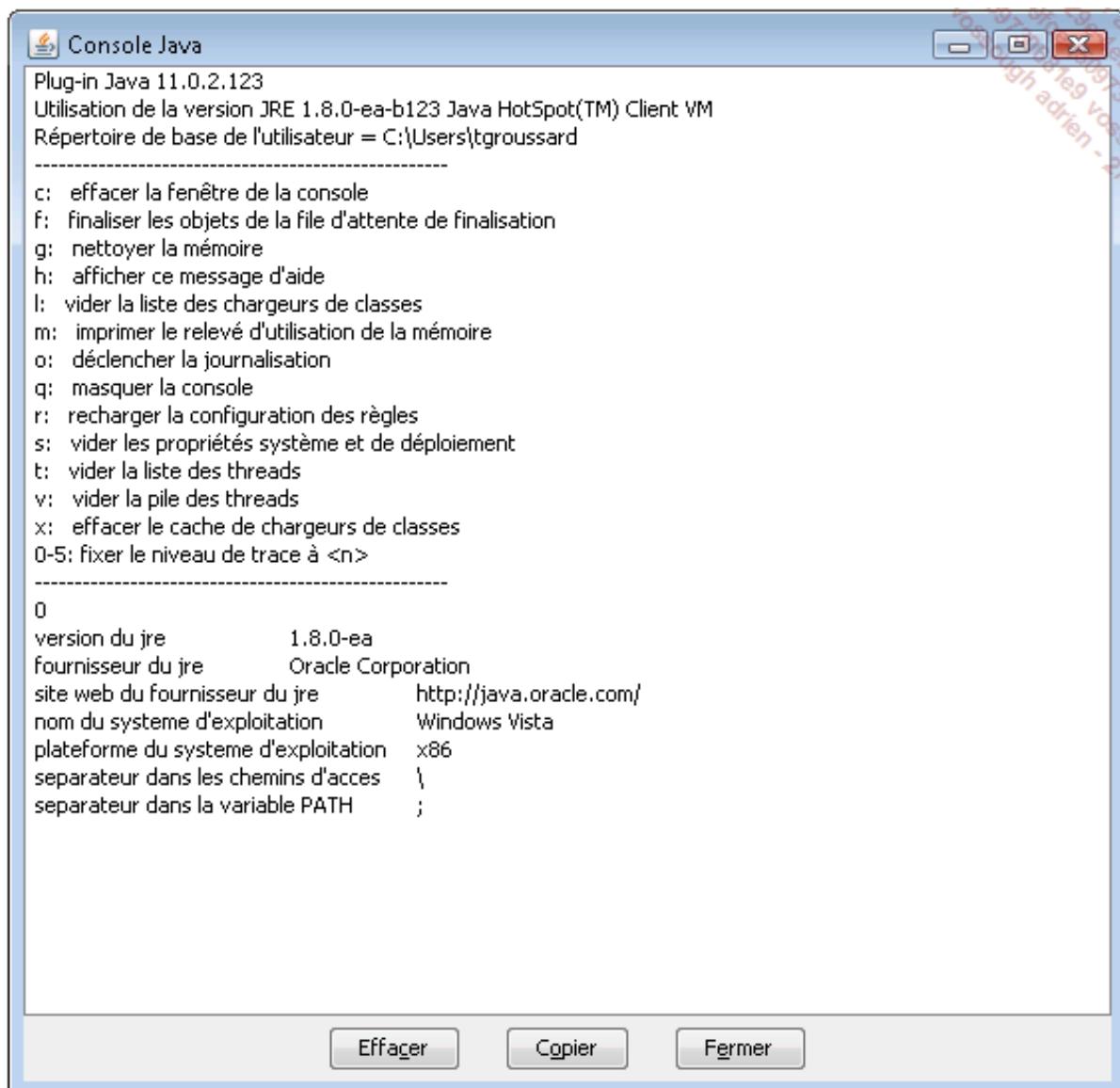
```

import java.applet.Applet;

public class Proprietes extends Applet
{
    public void start()
    {

```

```
System.out.print("version du jre \t");
System.out.println(System.getProperty("java.version"));
System.out.print("fournisseur du jre \t");
System.out.println(System.getProperty("java.vendor"));
System.out.print("site web du fournisseur du jre \t");
System.out.println(System.getProperty("java.vendor.url"));
System.out.print("nom du système d'exploitation \t");
System.out.println(System.getProperty("os.name"));
System.out.print("plateforme du système d'exploitation \t");
System.out.println(System.getProperty("os.arch"));
System.out.print(" séparateur dans les chemins d'accès \t");
System.out.println(System.getProperty("file.separator"));
System.out.print(" séparateur dans la variable PATH \t");
System.out.println(System.getProperty("path.separator"));
}
}
```



Principe de fonctionnement d'une base de données

Les bases de données sont devenues des éléments incontournables de la majorité des applications. Elles se substituent à l'utilisation de fichiers gérés par le développeur lui-même. Cet apport permet un gain de productivité important lors du développement et une amélioration significative des performances des applications. Elles facilitent également le partage d'informations entre utilisateurs. Pour pouvoir utiliser une base de données, vous devez connaître un minimum de vocabulaire lié à cette technologie.

1. Terminologie

Dans le contexte des bases de données les termes suivants sont fréquemment utilisés :

- Base de données relationnelle : une base de données relationnelle est un type de base de données qui utilise des tables pour le stockage des informations. Elles utilisent des valeurs issues de deux tables pour associer les données d'une table aux données d'une autre table. En règle générale, dans une base de données relationnelle, les informations ne sont stockées qu'une seule fois.
- Table : une table est un composant d'une base de données qui stocke les informations dans des enregistrements (lignes) et dans des champs (colonnes). Les informations sont en général regroupées par catégorie au niveau d'une table. Par exemple, nous aurons la table des Clients, des Produits, ou des Commandes.
- Enregistrement : l'enregistrement est l'ensemble des informations relatives à un élément d'une table. Les enregistrements sont les équivalents au niveau logique des lignes d'une table. Par exemple un enregistrement de la table Clients contient les caractéristiques d'un client particulier.
- Champ : un enregistrement est composé de plusieurs champs. Chaque champ d'un enregistrement contient une seule information sur l'enregistrement. Par exemple un enregistrement Client peut contenir les champs CodeClient, Nom, Prénom...
- Clé primaire : une clé primaire est utilisée pour identifier de manière unique chaque ligne d'une table. La clé primaire est un champ ou une combinaison de champs dont la valeur est unique dans la table. Par exemple, le champ CodeClient est la clé primaire de la table Client. Il ne peut pas y avoir deux clients ayant le même code.
- Clé étrangère : une clé étrangère représente un ou plusieurs champs d'une table qui font référence aux champs de la clé primaire d'une autre table. Les clés étrangères indiquent la manière dont les tables sont liées.
- Relation : une relation est une association établie entre des champs communs dans deux tables. Une relation peut être de un à un, un à plusieurs ou plusieurs à plusieurs. Grâce aux relations, les résultats de requêtes peuvent contenir des données issues de plusieurs tables. Une relation de un à plusieurs entre la table Client et la table Commande permet à une requête de renvoyer toutes les commandes correspondant à un client.

2. Le langage SQL

Avant de pouvoir écrire une application Java utilisant des données, vous devez être familiarisé avec le langage SQL (*Structured Query Language*). Ce langage permet de dialoguer avec la base de données. Il existe différentes versions du langage SQL en fonction de la base de données utilisée. Cependant SQL dispose également d'une syntaxe élémentaire normalisée indépendante de toute base de données.

a. Recherche d'informations

Le langage SQL permet de spécifier les enregistrements à extraire ainsi que l'ordre dans lequel vous souhaitez les extraire. Vous pouvez créer une instruction SQL qui extrait des informations de plusieurs tables simultanément, ou vous pouvez créer une instruction qui extrait uniquement un enregistrement spécifique. L'instruction SELECT est utilisée pour renvoyer des champs spécifiques d'une ou de plusieurs tables de la base de données.

L'instruction suivante renvoie la liste des noms et prénoms de tous les enregistrements de la table Client.

```
SELECT Nom,Prenom FROM Client
```

Vous pouvez utiliser le symbole * à la place de la liste des champs pour lesquels vous souhaitez la valeur.

```
SELECT * FROM Client
```

Vous pouvez limiter le nombre d'enregistrements sélectionnés en utilisant un ou plusieurs champs pour filtrer le résultat de la requête. Différentes clauses sont disponibles pour exécuter ce filtrage.

Clause WHERE

Cette clause permet de spécifier la liste des conditions que devront remplir les enregistrements pour faire partie des résultats retournés. L'exemple suivant permet de retrouver tous les clients habitant Nantes.

```
SELECT * FROM Client WHERE Ville='Nantes'
```

La syntaxe de la clause nécessite l'utilisation de simple cote pour la délimitation des chaînes de caractères.

Clause WHERE ... IN

Vous pouvez utiliser la clause WHERE ... IN pour renvoyer tous les enregistrements qui répondent à une liste de critères. Par exemple, vous pouvez rechercher tous les clients habitant en France ou en Espagne.

```
SELECT * FROM Client WHERE Pays IN ('France','Espagne')
```

Clause WHERE ... BETWEEN

Vous pouvez également renvoyer une sélection d'enregistrements qui se situent entre deux critères spécifiés. La requête suivante permet de récupérer la liste des commandes passées au mois de novembre 2005.

```
SELECT * from Commandes WHERE DateCommande BETWEEN '01/11/05' AND '30/11/05'
```

Clause WHERE ... LIKE

Vous pouvez utiliser la clause WHERE ... LIKE pour renvoyer tous les enregistrements pour lesquels il existe une condition particulière pour un champ donné. Par exemple, la syntaxe suivante sélectionne tous les clients dont le nom commence par un d :

```
SELECT * FROM Client WHERE Nom LIKE 'd%'
```

Dans cette instruction, le symbole % est utilisé pour remplacer une séquence de caractères quelconque.

Clause ORDER BY ...

Vous pouvez utiliser la clause ORDER BY pour renvoyer les enregistrements dans un ordre particulier. L'option ASC indique un ordre croissant, l'option DESC indique un ordre décroissant. Plusieurs champs peuvent être spécifiés comme critère de tri. Ils sont analysés de la gauche vers la droite. En cas d'égalité sur la valeur d'un champ, le champ suivant est utilisé.

```
SELECT * FROM Client ORDER BY Nom DESC,Prenom ASC
```

Cette instruction retourne les clients triés par ordre décroissant sur le nom et en cas d'égalité par ordre croissant sur le prénom.

b. Ajout d'informations

La création d'enregistrements dans une table se fait par la commande `INSERT INTO`. Vous devez indiquer la table dans laquelle vous souhaitez insérer une ligne, la liste des champs pour lesquels vous spécifiez une valeur et enfin la liste des valeurs correspondantes. La syntaxe complète est donc la suivante :

```
INSERT INTO client (codeClient,nom,prenom) VALUES (1000,'Dupond','Pierre')
```

Lors de l'ajout de ce nouveau client, seul le nom et le prénom seront renseignés dans la table. Les autres champs prendront la valeur `NULL`. Si la liste des champs n'est pas indiquée, l'instruction `INSERT` exige que vous spécifiez une valeur pour chaque champ de la table. Vous êtes donc obligé d'utiliser le mot clé `NULL` pour indiquer que pour un champ particulier il n'y a pas d'information. Si la table Client est composée de cinq champs (`codeClient,nom,prenom,adresse,pays`) l'instruction précédente peut être écrite avec la syntaxe suivante :

```
INSERT INTO client VALUES (1000,'Dupond','Pierre',NULL,NULL)
```

À noter que dans ce cas les deux mots clés `NULL` sont obligatoires pour les champs `adresse` et `pays`.

c. Mise à jour d'informations

La modification des champs pour des enregistrements existants s'effectue par l'instruction `UPDATE`. Cette instruction peut mettre à jour plusieurs champs de plusieurs enregistrements d'une table à partir des expressions qui lui sont fournies. Vous devez fournir le nom de la table à mettre à jour ainsi que la valeur à affecter aux différents champs. La liste est indiquée par le mot clé `SET` suivi de l'affectation de la nouvelle valeur aux différents champs. Si vous voulez que les modifications ne portent que sur un ensemble limité d'enregistrements, vous devez spécifier la clause `WHERE` afin de limiter la portée de la mise à jour. Si aucune clause `WHERE` n'est indiquée la modification se fera sur l'ensemble des enregistrements de la table.

Par exemple pour modifier l'adresse d'un client particulier, vous pouvez utiliser l'instruction suivante :

```
UPDATE Client SET adresse= '4 rue de Paris 44000 Nantes'  
WHERE codeClient=1000
```

Si la modification porte sur l'ensemble des enregistrements de la table, la clause `WHERE` est inutile. Par exemple si vous souhaitez augmenter le prix unitaire de tous vos articles, vous pouvez utiliser l'instruction suivante :

```
UPDATE CATALOGUE SET prixUnitaire=prixUnitaire*1.1
```

d. Suppression d'informations

L'instruction `DELETE FROM` permet de supprimer un ou plusieurs enregistrements d'une table. Vous devez au minimum fournir le nom de la table sur laquelle va se faire la suppression. Si vous n'indiquez pas plus de précisions, dans ce cas toutes les lignes de la table sont supprimées. En général une clause `WHERE` est ajoutée pour limiter l'étendue de la suppression. La commande suivante efface tous les enregistrements de la table Client :

```
DELETE FROM Client
```

La commande suivante est moins radicale et ne supprime qu'un enregistrement particulier :

```
DELETE FROM Client WHERE codeClient=1000
```

Le langage SQL est bien sûr beaucoup plus complet que cela et ne se résume pas à ces cinq instructions. Néanmoins, elles sont suffisantes pour la manipulation de données à partir de Java. Si vous souhaitez approfondir l'apprentissage du langage SQL je vous conseille de consulter un des ouvrages disponibles dans la même collection traitant de ce sujet de manière plus poussée.

Accès à une base de données à partir de Java

Lorsque l'on souhaite manipuler une base de données à partir d'un langage de programmation, deux solutions sont disponibles :

- Communiquer directement avec la base de données.
- Utiliser une couche logicielle assurant le dialogue avec la base de données.

La première solution comporte plusieurs exigences.

- Vous devez parfaitement maîtriser la programmation réseau.
- Vous devez également connaître en détail le protocole utilisé par la base de données.
- Ce type de développement est souvent très long et parsemé d'embûches. Par exemple les spécifications du protocole sont-elles accessibles ?
- Tout votre travail devra être recommandé si vous changez de type de base de données car les protocoles ne sont bien sûr pas compatibles d'une base de données à l'autre. Pire parfois même d'une version à l'autre d'une même base de données.

La deuxième solution est bien sûr préférable et c'est celle-ci que les concepteurs de Java ont choisie. Ils ont donc développé la bibliothèque jdbc pour l'accès à une base de données. Plus précisément la bibliothèque jdbc est composée de deux parties. La première partie contenue dans le package `java.sql` est essentiellement composée d'interfaces. Ces interfaces sont implémentées par le pilote jdbc. Ce pilote n'est pas développé par Sun mais en général par le concepteur de la base de données. C'est effectivement ce dernier qui maîtrise le mieux la technique pour communiquer avec la base de données. Il existe quatre types de pilotes jdbc avec des caractéristiques et des performances différentes.

- Type 1 : Pilote jdbc-odbc

Ce type de pilote n'est pas spécifique à une base de données mais il traduit simplement les instructions jdbc en instructions odbc. C'est ensuite le pilote odbc qui assure la communication avec la base de données. Cette solution n'offre que des performances médiocres. Ceci est principalement lié au nombre de couches logicielles mises en œuvre. Les fonctionnalités jdbc sont également limitées par les fonctionnalités de la couche odbc. Ce type de pilote n'est plus disponible à partir de la version 8 de Java. Oracle recommande l'utilisation du pilote spécifique à la base de données que vous souhaitez utiliser. Il est généralement disponible sur le site du concepteur de la base de données.

- Type 2 : Pilote natif

Ce type de pilote n'est pas entièrement écrit en Java. La partie de ce pilote écrite en Java effectue simplement des appels vers des fonctions du pilote natif. Ces appels sont effectués grâce à l'API JNI (*Java Native Interface*). Comme pour les pilotes de type 1, il y a donc une traduction nécessaire entre le code Java et la base de données. Cependant, ce type de pilote est tout de même plus efficace que les pilotes jdbc-odbc.

- Type 3 : pilote utilisant un serveur intermédiaire

Ce pilote développé entièrement en Java ne communique pas directement avec la base de données mais s'adresse à un serveur intermédiaire. Le dialogue entre le pilote et le serveur intermédiaire est standard quel que soit le type de base de données. C'est ensuite ce serveur intermédiaire qui transmet les requêtes en utilisant un protocole spécifique à la base de données.

- Type 4 : pilote entièrement écrit en Java

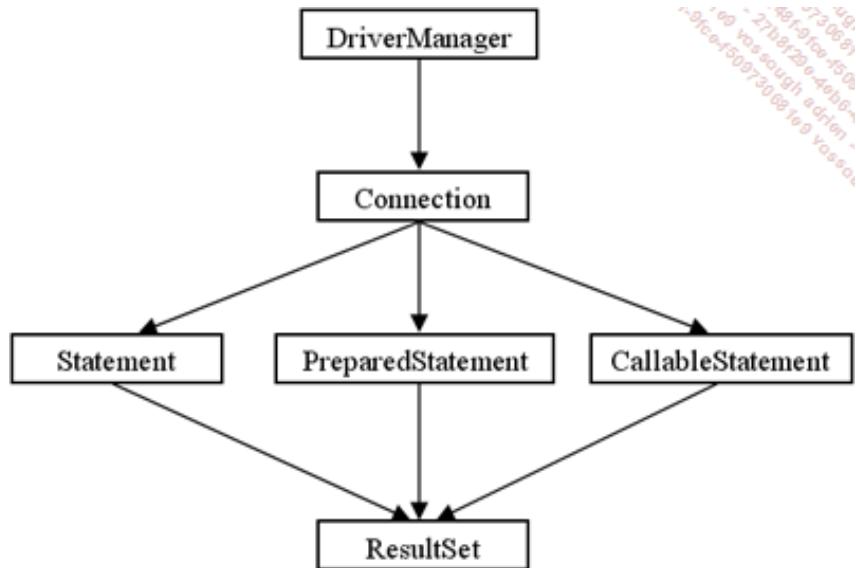
Ce type de pilote représente la solution idéale puisqu'il n'y a aucun intermédiaire. Le pilote transmet directement les requêtes à la base de données en utilisant le protocole propre à la base de données. La majorité des pilotes sont maintenant de ce type.

1. Présentation de jdbc

La bibliothèque jdbc fournit un ensemble de classes et surtout d'interfaces permettant la manipulation d'une base de données. Ces éléments représentent tout ce dont vous avez besoin pour accéder aux

données à partir d'une application Java.

Le schéma ci-dessous reprend le cheminement logique depuis le pilote jusqu'aux données.



La classe `DriverManager` est notre point de départ. C'est elle qui assure la liaison avec le pilote. C'est par son intermédiaire que nous pouvons obtenir une connexion vers la base de données. Celle-ci est représentée par une instance de classe implémentant l'interface `Connection`. Cette connexion est ensuite utilisée pour transmettre des instructions vers la base. Les requêtes simples sont exécutées grâce à l'interface `Statement`, les requêtes avec paramètres le sont avec l'interface `PreparedStatement` et les procédures stockées avec l'interface `CallableStatement`.

Les éventuels enregistrements sélectionnés par l'instruction SQL sont accessibles avec un élément `ResultSet`. Nous allons détailler ces différentes étapes dans les paragraphes suivants.

2. Chargement du pilote

La première étape indispensable est d'obtenir le pilote jdbc adapté à votre base de données. En général, ce pilote est disponible en téléchargement sur le site du concepteur de la base de données. Pour nos exemples, nous utiliserons le pilote fourni par Microsoft pour accéder à un serveur de base de données SQL Server. Il peut être téléchargé à l'adresse suivante : <http://www.microsoft.com/fr-fr/download/details.aspx?id=11774>

Après décompression du fichier, vous devez obtenir un répertoire contenant le pilote lui-même sous forme d'une archive Java (sqljdbc.jar) et de nombreux fichiers d'aide sur l'utilisation de ce pilote. Le fichier archive contient les classes développées par Microsoft qui implémentent les différentes interfaces jdbc. Ce fichier devra bien sûr être accessible au moment de la compilation et de l'exécution de l'application.

Ce pilote doit ensuite être chargé grâce à la méthode `forName` de la classe `Class`. Cette méthode attend comme paramètre une chaîne de caractères contenant le nom du pilote. Il est à ce niveau indispensable de parcourir la documentation du pilote pour obtenir le nom de la classe. Dans notre cas, cette classe porte le nom suivant :

com.microsoft.sqlserver.jdbc.SQLServerDriver

L'enregistrement du pilote se fait donc avec l'instruction suivante :

```
Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
```

Les chaînes de caractères passées comme paramètres représentent des noms de classes, elles sont donc sensibles à la casse. L'instruction `forName` doit d'ailleurs être protégée par un bloc `try catch` car elle est susceptible de déclencher une exception de type `ClassNotFoundException`.

3. Établir et manipuler la connexion

Après son chargement, le pilote est maintenant capable de nous fournir une connexion vers le serveur de base de données.

a. Établir la connexion

La méthode `getConnection` de la classe `DriverManager` se charge de ce travail. Celle-ci propose trois solutions pour établir la connexion. Chacune des versions de cette méthode attend comme paramètre une chaîne de caractères représentant une URL contenant les informations nécessaires pour établir la connexion. Le contenu de cette chaîne de caractères est spécifique à chaque pilote et il faut à nouveau consulter la documentation pour obtenir la bonne syntaxe. Le début de cette chaîne de caractères est cependant standard avec la forme suivante `jdbc:nomDuProtocole`. Le nom du protocole est propre à chaque pilote et c'est grâce à ce nom que la méthode `getConnection` est capable d'identifier le bon pilote. Le reste de la chaîne est lui complètement spécifique à chaque pilote. Il contient en général les informations permettant d'identifier le serveur et la base de données sur ce serveur vers laquelle la connexion doit être établie. Pour le pilote Microsoft, la syntaxe de base est la suivante :

```
jdbc:sqlserver://localhost;databaseName=northwind;  
user=thierry;password=secret;
```

En cas de succès, la méthode `getConnection` retourne une instance de classe implémentant l'interface `Connection`. Dans certains cas, il est préférable d'utiliser la deuxième version de la méthode `getConnection` car celle-ci permet d'indiquer le nom et le mot de passe utilisés pour établir la connexion comme paramètres séparés de l'URL de connexion.

Voici un exemple permettant d'établir une connexion vers une base de données.

```
import java.sql.Connection;  
import java.sql.DriverManager;  
import java.sql.SQLException;  
  
public class ConnexionDirecte  
{  
    public static void main(String[] args)  
    {  
        try  
        {  
            Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");  
        }  
        catch (ClassNotFoundException e)  
        {  
            System.out.println("erreur pendant le chargement  
du pilote");  
        }  
        Connection cnxDirect=null;  
        try  
        {  
            cnxDirect=DriverManager.getConnection("jdbc:sqlserver:  
//localhost;databaseName=northwind; user=thierry;password=secret;");  
        }  
        catch (SQLException e)  
        {  
            System.out.println("erreur pendant la connexion");  
        }  
    }  
}
```

b. Manipuler la connexion

Une connexion est ouverte dès sa création. Il n'y a donc pas de méthode permettant d'ouvrir une connexion. Par contre, une connexion peut être fermée en appelant la méthode `close`. Après la fermeture d'une connexion, il n'est plus possible de l'utiliser et elle doit être recréée pour pouvoir être à nouveau utilisable. La fonction `isClosed` permet de vérifier si une connexion est fermée. Si cette

fonction retourne un boolean qui est égal à `false`, on peut donc légitimement penser que la connexion est ouverte et permet donc de dialoguer avec la base de données. Ce n'est en fait pas toujours le cas. La connexion peut parfois être dans un état intermédiaire : elle n'est pas fermée mais elle ne peut pas être utilisée pour transférer des instructions vers le serveur. Pour vérifier la disponibilité de la connexion, vous pouvez utiliser la méthode `isValid`. Cette méthode teste réellement la disponibilité de la connexion en essayant d'envoyer une instruction SQL et en vérifiant qu'elle obtient bien une réponse de la part du serveur. Cette méthode n'est pas implémentée dans tous les pilotes et si elle ne l'est pas, son appel déclenche une exception du type `java.lang.UnsupportedOperationException` ou une erreur du type `java.lang.AbstractMethodError`.

Si vous effectuez uniquement des opérations de lecture sur la base de données vous pouvez optimiser la communication en précisant que la connexion est en lecture seule. La méthode `setReadOnly` permet de modifier ce paramétrage de la connexion. L'état peut ensuite être testé en utilisant la méthode `isReadOnly`. Pour certains pilotes cette fonctionnalité n'est pas implémentée et la méthode `setReadOnly` n'a aucun effet. La fonction suivante vérifie si cette fonctionnalité est disponible pour la connexion qui lui est passée comme paramètre.

```
public static void testLectureSeule(Connection cnx)
{
    boolean etat;
    try
    {
        etat = cnx.isReadOnly();
        cnx.setReadOnly(!etat);
        if (cnx.isReadOnly() != etat)
        {
            System.out
                .println("le mode lecture seule est pris en
charge par ce pilote");
        }
        else
        {
            System.out
                .println("le mode lecture seule n'est pas
pris en charge par ce pilote");
        }
        cnx.setReadOnly(etat);
    }
    catch (SQLException e)
    {
        e.printStackTrace();
    }
}
```

Lors de l'exécution d'une instruction SQL, le serveur peut détecter des problèmes et de ce fait générer des avertissements. Ces avertissements peuvent être récupérés par la méthode `getWarnings` de la connexion. Cette méthode retourne un objet `SQLWarning` représentatif du problème rencontré par le serveur. Si plusieurs problèmes sont rencontrés par le serveur celui-ci génère plusieurs objets `SQLWarning` chaînés les uns aux autres. La méthode `getNextWarning` permet d'obtenir l'élément suivant ou null si la liste est terminée. La liste peut être vidée avec la méthode `clearWarnings`. La fonction suivante affiche tous les avertissements reçus par la connexion.

```
public static void affichageWarnings(Connection cnx)
{
    SQLWarning avertissement;
    try
    {
        avertissement=cnx.getWarnings();
        if (avertissement==null)
        {
            System.out.println("il n'y a pas d'avertissement");
        }
        else
        {
            while (avertissement!=null)
```

```

        {
            System.out.println(avertissement.getMessage());
            System.out.println(avertissement.getSQLState());
            System.out.println(avertissement.getErrorCode());
            avertissement=avertissement.getNextWarning();
        }
    }
    cnx.clearWarnings();
}
catch (SQLException e)
{
    e.printStackTrace();
}
}

```

En général, lors de la création de l'url de connexion, l'un des paramètres de cette URL détermine le nom de la base vers laquelle vous souhaitez établir une connexion. La modification du nom de la base de données à laquelle vous êtes connecté se réalise avec la méthode `setCatalog` à laquelle il faut fournir le nom d'une autre base présente sur le même serveur. Il faut bien sûr que le compte avec lequel vous avez ouvert la connexion dispose des droits d'accès suffisants pour cette base de données. À noter qu'avec cette méthode, nous changeons de base de données mais la connexion concerne toujours le même serveur. Il n'y a aucun moyen de changer de serveur sans créer une nouvelle connexion.

Le code suivant :

```

System.out.println("base actuelle : " +cnxDirect.getCatalog());
System.out.println("changement de base de données");
cnxDirect.setCatalog("garage");
System.out.println("base actuelle : " +cnxDirect.getCatalog());

```

nous affiche :

```

base actuelle : northwind
changement de base de données
base actuelle : garage

```

La structure de la base peut également être obtenue avec la méthode `getMetaData`. Cette méthode retourne un objet `DatabaseMetaData` fournissant de très nombreuses informations sur la structure de la base. La fonction suivante affiche une analyse rapide de ces informations.

```

public static void infosBase(Connection cn)
{
    ResultSet rs;
    DatabaseMetaData dbmd;
    try
    {
        dbmd=cn.getMetaData();
        System.out.println("type de base : " +
dbmd.getDatabaseProductName());
        System.out.println("version: " +
dbmd.getDatabaseProductVersion());
        System.out.println("nom du pilote : " + dbmd.getDriverName());
        System.out.println("version du pilote : " +
dbmd.getDriverVersion());
        System.out.println("nom de l'utilisateur : " +
System.out.println("url de connexion : " + dbmd.getURL());
        rs=dbmd.getTables(null,null,"%",null);
        System.out.println("structure de la base");
        System.out.println("base\tschema\tnom table\ttype table");
        while(rs.next())
        {
            for (int i = 1; i <=4 ; i++)
            {
                System.out.print(rs.getString(i)+"\t");
            }
            System.out.println();
        }
    }
}

```

```

        }
        rs.close();
        rs=dbmd.getProcedures(null,null,"%");
        System.out.println("les procédures stockées");
        System.out.println("base\tschema\tnom procedure");
        while(rs.next())
        {
            for (int i = 1; i <=3 ; i++)
            {
                System.out.print(rs.getString(i)+"\t");
            }
            System.out.println();
        }
        rs.close();
    }
    catch (SQLException e)
    {
        e.printStackTrace();
    }
}

```

Toutes ces méthodes peuvent un jour ou l'autre vous rendre service mais le but principal d'une connexion est de permettre l'exécution d'instructions SQL. C'est donc elle qui va nous fournir les objets nécessaires pour l'exécution de ces instructions. Trois types d'instructions SQL peuvent être exécutés :

- Les requêtes simples
- Les requêtes précompilées
- Les procédures stockées.

À chacun de ces types correspond un objet JDBC :

- Statement pour les requêtes simples
- PreparedStatement pour les requêtes précompilées
- CallableStatement pour les procédures stockées.

Et enfin chacun des ces objets peut être obtenu par une méthode différente de la connexion :

- createStatement pour les objets Statement
- prepareStatement pour les objets PreparedStatement
- prepareCall pour les objets CallableStatement.

L'utilisation de ces méthodes et de ces objets est détaillée dans la section suivante.

4. Exécution d'instructions SQL

Avant l'exécution d'une instruction SQL vous devez choisir le type d'objet JDBC le plus approprié. Les rubriques suivantes décrivent les trois types d'objets disponibles et leur utilisation.

a. Exécution d'instructions de base avec l'objet Statement

Cet objet est obtenu par la méthode `createStatement` de la connexion. Deux versions de cette méthode sont disponibles ; la première n'attend aucun paramètre. Dans ce cas, si l'objet `Statement` est utilisé pour exécuter une instruction SQL générant un jeu d'enregistrements (`select`), celui-ci sera en lecture seule et avec un défilement en avant.

Les informations présentes dans ce jeu d'enregistrements ne pourront pas être modifiées et le parcours du jeu d'enregistrements ne pourra se faire que du premier enregistrement vers le dernier enregistrement. La deuxième version permet de choisir les caractéristiques du jeu d'enregistrements généré. Elle accepte deux paramètres. Le premier détermine le type du jeu d'enregistrements.

Les constantes suivantes sont définies :

- `ResultSet.TYPE_FORWARD_ONLY` : le jeu d'enregistrements sera à défilement en avant seulement.
- `ResultSet.TYPE_SCROLL_INSENSITIVE` : le jeu d'enregistrements pourra être parcouru dans les deux sens mais sera insensible aux changements effectués dans la base de données par d'autres utilisateurs.
- `ResultSet.TYPE_SCROLL_SENSITIVE` : le jeu d'enregistrements pourra être parcouru dans les deux sens et sera sensible aux changements effectués dans la base de données par d'autres utilisateurs.

Le second détermine les possibilités de modification des informations contenues dans le jeu d'enregistrements. Les deux constantes suivantes sont définies :

- `ResultSet.CONCUR_READ_ONLY` : les enregistrements sont en lecture seule.
- `ResultSet.CONCUR_UPDATABLE` : les enregistrements peuvent être modifiés dans le jeu d'enregistrements.

Cet objet est le plus élémentaire permettant l'exécution d'instructions SQL. Il peut prendre en charge l'exécution de n'importe quelle instruction SQL. Vous pouvez donc exécuter aussi bien des instructions DDL (*Data Definition Language*) que des instructions DML (*Data Manipulation Language*). Il faut simplement choisir dans cet objet la méthode la plus adaptée pour l'exécution du code SQL. Le choix de cette méthode est dicté par le type de résultat que doit fournir l'instruction SQL. Quatre méthodes sont disponibles :

- `public boolean execute(String sql)` : cette méthode permet l'exécution de n'importe quelle instruction SQL. Le boolean retourné par cette méthode indique si un jeu d'enregistrements a été généré (true) ou si simplement l'instruction a modifié des enregistrements dans la base de données (false). Si un jeu d'enregistrements est généré, il peut être obtenu par la méthode `getResultSet`. Si des enregistrements ont été modifiés dans la base de données, la méthode `getUpdateCount` retourne le nombre d'enregistrements modifiés. La fonction suivante illustre l'utilisation de ces méthodes.

```
public static void testExecute(Connection cnx)
{
    Statement stm;
    BufferedReader br;
    String requete;
    ResultSet rs;
    boolean resultat;
    try
    {
        stm=cnx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
        ResultSet.CONCUR_READ_ONLY);
        br=new BufferedReader(new InputStreamReader(System.in));
        System.out.println("saisir votre instruction SQL :");
        requete=br.readLine();
        resultat=stm.execute(requete);
        if (resultat)
        {
            System.out.println("votre instruction a généré
un jeu d'enregistrements");
            rs=stm.getResultSet();
            rs.last();
            System.out.println("il contient "
+ rs.getRow() + " enregistrements");
        }
        else
        {
            System.out.println("votre instruction a modifié
des enregistrements dans la base");
            System.out.println("nombre d'enregistrements
modifiés :"
+ stm.getUpdateCount());
        }
    }
}
```

```

        }
    } catch (SQLException e)
    {
        System.out.println("votre instruction n'a pas fonctionné
correctement");
    } catch (IOException e)
    {
        e.printStackTrace();
    }
}

```

public Resultset executeQuery(String requete) : cette méthode est conçue spécialement pour l'exécution d'instructions select. Le jeu d'enregistrements est disponible directement comme valeur renournée par la fonction.

- public int executeUpdate(String requête) : cette méthode est parfaitement adaptée pour l'exécution d'instructions modifiant le contenu de la base de données comme les instructions insert, update, delete. L'entier retourné par cette fonction indique le nombre d'enregistrements affectés par la modification.
- public int[] executeBatch() : cette méthode permet d'exécuter un ensemble d'instructions SQL par lot. Le lot d'instructions à exécuter doit être préparé au préalable avec les méthodes addBatch et clearBatch. La première reçoit comme paramètre une chaîne de caractères représentant une instruction SQL à ajouter au lot. La seconde permet de réinitialiser le lot d'instructions. Il n'est pas possible de supprimer une instruction particulière du lot. Vous ne devez pas ajouter au lot d'instruction SQL générant un jeu de résultats, car dans ce cas la méthode updateBatch déclenche une exception de type BatchUpdateException. Cette fonction retourne un tableau d'entiers permettant d'obtenir une information sur l'exécution de chacune des requêtes du lot. Chaque case du tableau contient un entier représentatif du résultat de l'exécution de la requête correspondante dans le lot. Une valeur supérieure ou égale à 0 indique un fonctionnement correct de l'instruction et représente le nombre d'enregistrements modifiés. Une valeur égale à la constante Statement.EXECUTE_FAILED indique que l'exécution de l'instruction a échoué. Dans ce cas, certains pilotes arrêtent l'exécution du lot alors que d'autres continuent avec l'instruction suivante du lot. Une valeur égale à la constante Statement.SUCCESS_NO_INFO indique que l'instruction a été exécutée correctement mais que le nombre d'enregistrements modifiés ne peut pas être déterminé. Cette méthode est très pratique pour exécuter des modifications sur plusieurs tables liées. Ce pourrait par exemple être le cas dans une application de gestion de commerce avec une table pour les commandes et une table pour les lignes de commande. La suppression d'une commande doit dans ce cas entraîner la suppression de toutes les lignes correspondantes. La fonction suivante vous permet de saisir plusieurs instructions SQL et de les exécuter par lot.

```

public static void testExecuteBatch(Connection cnx)
{
    Statement stm;
    BufferedReader br;
    String requete="";
    int[] resultats;
    try
    {
stm=cnx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY);
        br=new BufferedReader(new InputStreamReader(System.in));
        System.out.println("saisir vos instructions SQL puis
run pour exécuter le lot :");
        requete=br.readLine();
        while (!requete.equalsIgnoreCase("run"))
        {
            stm.addBatch(requete);
            requete=br.readLine();
        }
        System.out.println("exécution du lot d'instructions");
        resultats=stm.executeBatch();
        for (int i=0; i<resultats.length;i++)

```

```

        {
            switch (resultats[i])
            {
                case Statement.EXECUTE_FAILED:
                    System.out.println("l'\exécution de
l'instruction " + i + " a échoué");
                    break;
                case Statement.SUCCESS_NO_INFO:
                    System.out.println("l'\exécution de
l'instruction " + i + " a réussi");
                    System.out.println("le nombre
d'enregistrements modifiés est inconnu");
                    break;
                default:
                    System.out.println("l'\exécution de
l'instruction " + i + " a réussi");
                    System.out.println("elle a modifié " +
resultats[i] + " enregistrements");
                    break;
            }
        }

    }
    catch (SQLException e)
    {
        e.printStackTrace();
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
}
}

```

Plusieurs autres méthodes vont intervenir sur le comportement de l'objet Statement :

- public void setQueryTimeOut(int duree) : cette méthode indique la durée maximale allouée pour l'exécution d'une instruction SQL avant le déclenchement d'une exception.
- public void close() : lorsqu'un objet Statement n'est plus utile dans une application, il est préférable de le fermer explicitement en appelant cette méthode. Celle-ci provoque la libération de toutes les ressources utilisées par cet objet. La fermeture d'un objet statement provoque également la fermeture du jeu d'enregistrements associé.
- public void setMaxRows(int nombre) : cette méthode limite le nombre de lignes des jeux d'enregistrements générés par cet objet Statement. Si une instruction SQL génère un jeu d'enregistrements comportant plus de lignes, les lignes excédentaires sont tout simplement ignorées (sans plus d'information).
- public void setMaxFieldSize(int taille) : cette méthode limite la taille de certains types de champs dans le jeu d'enregistrements. Les types de champs concernés sont les champs d'une base de données pouvant avoir une taille variable comme par exemple les champs caractères ou binaires. Les données excédentaires sont simplement ignorées. Les champs concernés peuvent de ce fait être inutilisables dans l'application.
- public void setFetchSize(int nbLignes) : lorsqu'une instruction SQL génère un jeu d'enregistrements, les données correspondantes sont transférées du serveur de base de données vers la mémoire de l'application Java. Ce transfert est effectué par blocs en fonction de l'utilisation des données. Cette méthode indique au pilote le nombre de lignes de chaque bloc transféré de la base de données vers l'application.
- public boolean getMoreResults() : si la méthode execute est utilisée pour exécuter plusieurs instruction SQL, par exemple deux instructions select, il y a dans ce cas génération de deux jeux d'enregistrements. Le premier est obtenu par la méthode getResultSet. Le second ne sera accessible qu'après avoir appelé la méthode getMoreResults. Cette fonction permet de déplacer le pointeur sur le résultat suivant et retourne un boolean égal à true si le résultat suivant est un jeu d'enregistrements. Si c'est le cas, il est lui aussi accessible par la méthodegetResultSet. Si la fonction getMoreResults retourne un boolean false, c'est que le

résultat suivant n'est pas un jeu d'enregistrements ou qu'il n'y a simplement pas de résultat suivant. Pour lever le doute, il faut donc appeler la fonction `getUpdateCount` et vérifier la valeur renournée par cette fonction. Si cette valeur est égale à -1 c'est qu'il n'y a pas de résultat suivant sinon la valeur renournée représente le nombre d'enregistrements modifiés dans la base de données.

La fonction suivante permet l'exécution de plusieurs instructions SQL séparées par des points-virgules :

```
public static void testExecuteMultiple(Connection cnx)
{
    Statement stm;
    BufferedReader br;
    String requete;
    ResultSet rs;
    boolean resultat;
    try
    {
        stm=cnx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
        ResultSet.CONCUR_READ_ONLY);
        br=new BufferedReader(new InputStreamReader(System.in));
        System.out.println("saisir vos instruction SQL séparées
par ; :"));
        requete=br.readLine();
        resultat=stm.execute(requete);
        int i=1;
        // traitement du résultat généré par la première
        // instruction
        if (resultat)
        {
            System.out.println("votre instruction N° " + i + " a
généré un jeu d'enregistrements");
            rs=stm.getResultSet();
            rs.last();
            System.out.println("il contient " + rs.getRow() + "
enregistrements");
        }
        else
        {
            System.out.println("votre instruction N° " + i + " a
modifié des enregistrements dans la base");
            System.out.println("nombre d'enregistrements
modifiés :" + stm.getUpdateCount());
        }
        i++;
        // déplacement du pointeur sur un éventuel résultat
        // suivant
        resultat=stm.getMoreResults();
        // boucle tant qu'il y a encore un résultat de type jeu
        // d'enregistrement -> resultat==true
        // ou qu'il y a encore un résultat de type nombre
        // d'enregistrements modifiés -> getUpdateCount != -1
        while (resultat || stm.getUpdateCount() != -1)
        {
            if (resultat)
            {
                System.out.println("votre instruction N° " +
i + " a généré un jeu d'enregistrements");
                rs=stm.getResultSet();
                rs.last();
                System.out.println("il contient " + rs.getRow()
+ " enregistrements");
            }
            else
            {
                System.out.println("votre instruction N° " +
i + " a modifié des enregistrements dans la base");
                System.out.println("nombre d'enregistrements
modifiés :" + stm.getUpdateCount());
            }
        }
    }
}
```

```

        i++;
        // déplacement du pointeur sur un éventuel résultat
        // suivant
        resultat=stm.getMoreResults();
    }
}
catch (SQLException e)
{
    System.out.println("votre instruction n'a pas fonctionné
correctement");
    e.printStackTrace();
}
catch (IOException e)
{
    e.printStackTrace();
}
}

```

b. Exécution d'instructions paramétrées avec l'objet PreparedStatement

Il arrive fréquemment d'avoir à faire exécuter plusieurs fois une requête SQL avec juste une petite modification entre deux exécutions. L'exemple classique correspond à une requête de sélection avec une restriction.

```

stm=cnx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY);
resultat=stm.execute("select * from customers where customerId=
'ALFKI'");

```

La valeur sur laquelle porte la restriction est en général saisie par l'utilisateur de l'application et dans ce cas disponible dans une variable. La première solution qui vient à l'esprit consiste à construire la requête SQL par concaténation de plusieurs chaînes de caractères.

```

public static void testRequeteConcat(Connection cnx)
{
    Statement stm;
    BufferedReader br;
    String requete;
    String code;
    ResultSet rs;
    boolean resultat;
    try
    {
stm=cnx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY);
        br=new BufferedReader(new InputStreamReader(System.in));
        System.out.println("saisir le code du client recherché :");
        code=br.readLine();
        requete="select * from customers where customerID='"
+ code
+"'";
        resultat=stm.execute(requete);
    }
    catch (SQLException e)
    {
        e.printStackTrace();
    } catch (IOException e) {
        // TODO Bloc catch autogénéré
        e.printStackTrace();
    }
}

```

Cette solution présente plusieurs inconvénients :

- La concaténation de chaînes de caractères consomme beaucoup de ressources.
- Le serveur doit analyser à chaque fois une nouvelle requête.

- La syntaxe va rapidement devenir complexe si plusieurs chaînes doivent être concaténées. Il faut toujours conserver à l'esprit qu'au final nous devons obtenir une instruction SQL correcte.

Les concepteurs de JDBC ont prévu une solution efficace pour pallier ces inconvénients. L'objet `PreparedStatement` apporte une solution efficace à ce problème en nous permettant la création de requêtes avec paramètres. Dans ce type de requête, les paramètres sont remplacés par des points d'interrogation. Avant l'exécution de la requête, il faut fournir à l'objet `PreparedStatement` les valeurs qu'il doit utiliser pour remplacer les différents points d'interrogation. Un objet `PreparedStatement` peut être créé en utilisant le même principe que pour la création d'un objet `Statement`.

La méthode `prepareStatement` accessible à partir d'une connexion retourne un objet `PreparedStatement`. Cette méthode est disponible sous deux formes. La première attend comme argument une chaîne de caractères représentant la requête SQL. Dans cette requête, l'emplacement des paramètres doivent être réservé par des points d'interrogation. Si un jeu d'enregistrements est créé par l'exécution de cette requête, celui-ci sera en lecture seule et à défilement en avant seulement. La deuxième forme de la méthode `prepareStatement` attend en plus de la chaîne de caractères un argument indiquant le type de jeu d'enregistrements et un autre déterminant les possibilités de modification des informations contenues dans le jeu d'enregistrements.

Les mêmes constantes que pour la méthode `createStatement` peuvent être utilisées.

```
stm=cnx.prepareStatement("select * from customers where
customerID=?",
ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY);
```

Avant l'exécution de l'instruction SQL, vous devez fournir une valeur pour chacun des points d'interrogation représentant un paramètre. Pour cela, l'objet `PreparedStatement` dispose de nombreuses méthodes permettant l'affectation d'une valeur à un paramètre. Chacune de ces méthodes correspond au type de données SQL à insérer à la place d'un point d'interrogation. Ces méthodes sont nommées selon le même schéma : `setXXXX` où `XXXX` représente un type de données SQL. Chacune de ces méthodes attend comme premier argument un entier correspondant au rang du paramètre dans l'instruction SQL. Le premier paramètre est situé au rang 1. Le deuxième argument correspond à la valeur à transférer dans le paramètre. Le type de cet argument correspond bien sûr au type de données SQL à transférer vers le paramètre. Le pilote jdbc convertit ensuite le type Java en type SQL.

Par exemple la méthode `setInt` (`int indiceParam, int value`) effectue une conversion du type int Java en type INTEGER sql. Les valeurs stockées dans les paramètres sont conservées d'une exécution à l'autre de l'instruction SQL. La méthode `clearParameters` permet de réinitialiser l'ensemble des paramètres.

Les paramètres peuvent être utilisés dans une instruction SQL en remplacement de valeurs mais jamais pour remplacer un nom de champ, encore moins un opérateur. La syntaxe suivante est bien sûr interdite :

```
stm=cnx.prepareStatement("select * from customers where
?=?",
ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_READ_ONLY);
stm.setString(1,"customerID");
stm.setString(2,"ALFKI");
rs=stm.executeQuery();
```

Les autres méthodes disponibles avec un objet `PreparedStatement` sont parfaitement identiques à celles définies pour un objet `Statement` puisque l'interface `PreparedStatement` hérite directement de l'interface `Statement`.

```
public static void testPreparedStatement(Connection cnx)
{
    PreparedStatement stm;
    BufferedReader br;
    String code;
    ResultSet rs;
    try
    {
```

```
        stm=cnx.prepareStatement("select * from customers  
where customerID like ?",ResultSet.TYPE_SCROLL_SENSITIVE,  
ResultSet.CONCUR_READ_ONLY);  
        br=new BufferedReader(new  
InputStreamReader(System.in));  
        System.out.println("saisir le code du client  
recherché :");  
        code=br.readLine();  
        stm.setString(1,code);  
        rs=stm.executeQuery();  
        while (rs.next())  
        {  
            for (int i = 1; i  
<=rs.getMetaData().getColumnCount(); i++)  
            {  
                System.out.print(rs.getString(i)+"\t");  
            }  
            System.out.println();  
        }  
    }  
    catch (SQLException e)  
    {  
        e.printStackTrace();  
    }  
    catch (IOException e)  
    {  
        e.printStackTrace();  
    }  
}
```

c. Exécution de procédures stockées avec l'objet CallableStatement

Une procédure stockée représente du code SQL stocké sur le serveur. Cette approche procure plusieurs avantages :

- Les traitements et les temps de réponse sont améliorés.
 - Le code SQL peut être partagé entre plusieurs applications.

Par contre, les applications deviennent beaucoup plus dépendantes du serveur de base de données. Le changement de serveur vous obligera certainement à réécrire vos procédures stockées car la syntaxe d'une procédure stockée est propre à chaque serveur. Les procédures stockées sont accessibles à partir de Java grâce à l'objet `CallableStatement`. Comme pour les objets `Statement` et `PreparedStatement`, c'est encore une fois la connexion qui va nous fournir une instance de classe correcte. C'est dans ce cas la méthode `prepareCall` qui doit être utilisée. Cette méthode attend comme argument une chaîne de caractères identifiant la procédure stockée à appeler. Par contre la syntaxe de cette chaîne de caractères est un petit peu spéciale puisqu'il ne suffit pas d'indiquer le nom de la procédure stockée.

Deux cas de figures sont à prendre en compte suivant que la procédure stockée retourne ou non une valeur. Si la procédure stockée ne retourne pas de valeur, la syntaxe de cette chaîne de caractères est la suivante :

```
{call nomProcedure( ?, ?,... )}
```

Dans cette syntaxe, les points d'interrogation représentent les paramètres attendus par la procédure stockée. Comme pour l'objet `PreparedStatement` les valeurs de ces paramètres doivent être fournies par les méthodes `setXXXX` correspondantes au type du paramètre.

Si la procédure stockée retourne une valeur, il faut utiliser la syntaxe suivante qui ajoute un paramètre supplémentaire pour accueillir le retour de la procédure stockée.

```
{ ?=call nomProcedure( ?, ?, ... ) }
```

Le premier paramètre étant utilisé en sortie de la procédure stockée, c'est l'exécution de la procédure stockée qui va y stocker une valeur, vous devez en informer l'objet `CallableStatement` en appelant la méthode `registerOutParameter`. Cette méthode attend comme premier argument, l'indice du

paramètre 1 pour la valeur de retour de la procédure stockée, mais n'importe quel autre paramètre peut être utilisé en sortie, puis comme deuxième argument le type SQL du paramètre. Ce type peut être indiqué avec une des constantes définies dans l'interface `java.sql.Types`. Après exécution de la procédure stockée, la valeur des paramètres utilisés en sortie est accessible par les méthodes `getXXXX` où `XXXX` représente le type SQL du paramètre. Ces méthodes attendent comme argument l'indice du paramètre dans l'appel de la procédure stockée.

Comme pour les méthodes `createStatement` et `prepareStatement`, la méthode `prepareCall` propose une deuxième syntaxe permettant d'indiquer les caractéristiques d'un éventuel jeu d'enregistrements généré par l'exécution de la procédure stockée.

Pour illustrer l'utilisation de l'objet `CallableStatement`, nous allons utiliser les deux procédures stockées suivantes :

```
create PROCEDURE commandesParClient @code nchar(5)
AS
SELECT OrderID,
       OrderDate,
       RequiredDate,
       ShippedDate
FROM Orders
WHERE CustomerID = @code
ORDER BY OrderID

CREATE procedure nbCommandes @code nchar(5) as
declare @nb int
select @nb=count(*) from orders where customerid=@code
return @nb
```

La première retourne la liste de toutes les commandes du client dont le code est passé comme paramètre. La deuxième retourne un entier correspondant au nombre de commandes passées par le client dont le code est fourni comme paramètre.

```
public static void testProcedureStockee(Connection cnx)
{
    CallableStatement cstm1,cstm2;
    BufferedReader br;
    String code;
    ResultSet rs;
    int nbCommandes;
    try
    {
        br=new BufferedReader(new InputStreamReader(System.in));
        System.out.println("saisir le code du client recherché :");
        code=br.readLine();
        cstm1=cnx.prepareCall("{ ?=call nbCommandes ( ? )}");
        cstm1.setString(2,code);
        cstm1.registerOutParameter(1,java.sql.Types.INTEGER);
        cstm1.execute();
        nbCommandes=cstm1.getInt(1);
        System.out.println("nombre de commandes passées par le
client " + code + " : " + nbCommandes );
        cstm2=cnx.prepareCall("{ call commandesParClient ( ?
)}",ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_READ_ONLY);
        cstm2.setString(1,code);
        rs=cstm2.executeQuery();
        System.out.println("détail des commandes");
        System.out.println("numéro de commande\tdate de commande");
        while (rs.next())
        {
            System.out.print(rs.getInt("OrderID") + "\t");
            System.out.println(new
SimpleDateFormat("dd/MM/yy").format(rs.getDate("OrderDate")));
        }
    }
    catch (SQLException e)
```

```

    {
        e.printStackTrace();
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
}

```

5. Utilisation des jeux d'enregistrements avec l'interface ResultSet

Lorsqu'une instruction SQL select est exécutée par la méthode executeQuery d'un objet Statement,PreparedStatement ou CallableStatement, celle-ci retourne un objet ResultSet. C'est par l'intermédiaire de cet objet ResultSet que nous allons pouvoir intervenir sur le jeu d'enregistrements. Nos possibilités d'action sur ce jeu d'enregistrements sont déterminées par les caractéristiques de l'objetResultSet. Ces caractéristiques sont fixées au moment de la création des objets Statement,PreparedStatement ou CallableStatement en fonction des arguments passés lors de l'appel des méthodes createStatement, prepareStatement ou prepareCall.

Le premier argument détermine le type du jeu d'enregistrements. Les constantes suivantes sont définies :

- ResultSet.TYPE_FORWARD_ONLY : le jeu d'enregistrements sera à défilement en avant seulement.
- ResultSet.TYPE_SCROLL_INSENSITIVE : le jeu d'enregistrements pourra être parcouru dans les deux sens mais sera insensible aux changements effectués dans la base de données par d'autres utilisateurs.
- ResultSet.TYPE_SCROLL_SENSITIVE : le jeu d'enregistrements pourra être parcouru dans les deux sens et sera sensible aux changements effectués dans la base de données par d'autres utilisateurs.

Le second argument détermine les possibilités de modification des informations contenues dans le jeu d'enregistrements. Les deux constantes suivantes sont définies :

- ResultSet.CONCUR_READ_ONLY : les enregistrements sont en lecture seule.
- ResultSet.CONCUR_UPDATABLE : les enregistrements peuvent être modifiés dans le jeu d'enregistrements.

Il faut bien sûr que les actions exécutées sur l'objet Resultset soit compatibles avec ces caractéristiques sinon une exception sera déclenchée. Il est possible de vérifier les caractéristiques d'un objet Resultset en utilisant les méthodes getType et getConcurrency.

```

public static void infosResultSet(ResultSet rs)
{
    try {
        switch (rs.getType())
        {
            case ResultSet.TYPE_FORWARD_ONLY:
                System.out.println("le Resultset est à
défilement en avant seulement");
                break;
            case ResultSet.TYPE_SCROLL_INSENSITIVE:
                System.out.println("le Resultset peut être
parcouru dans les deux sens");
                System.out.println("il n'est pas sensible aux
modifications faites par d'autres utilisateurs");
                break;
            case ResultSet.TYPE_SCROLL_SENSITIVE:
                System.out.println("le Resultset peut être
parcouru dans les deux sens");
                System.out.println("il est sensible aux
modifications faites par d'autres utilisateurs");
                break;
        }
        switch (rs.getConcurrency())

```

```

        {
            case ResultSet.CONCUR_READ_ONLY:
                System.out.println("les données contenues dans
le ResulSet sont en lecture seule");
                break;
            case ResultSet.CONCUR_UPDATABLE:
                System.out.println("les données contenues dans
le ResulSet sont modifiables");
                break;
        }
    } catch (SQLException e)
    {
        e.printStackTrace();
    }
}

```

a. Positionnement dans un ResultSet

L'objet `ResultSet` gère un pointeur d'enregistrement déterminant sur quel enregistrement vont intervenir les méthodes exécutées sur le `ResultSet` lui-même. Cet enregistrement est parfois appelé enregistrement actif ou enregistrement courant. L'objet `ResultSet` contient toujours deux enregistrements fictifs servant de repère pour le début du `ResultSet` (BOF) et pour la fin du `ResultSet` (EOF). Le pointeur d'enregistrement peut être positionné sur l'un de ces deux enregistrements mais jamais avant l'enregistrement BOF ni après l'enregistrement EOF. Ces enregistrements ne contiennent pas de données et une opération de lecture ou d'écriture sur ces enregistrements déclenche une exception. À la création du `ResultSet` le pointeur est positionné avant le premier enregistrement (BOF). De nombreuses méthodes sont disponibles pour gérer le pointeur d'enregistrements :

- `boolean absolute(int position)` : déplace le pointeur d'enregistrement sur l'enregistrement spécifié. La numérotation des enregistrements débute à 1. Si la valeur de l'argument `position` est négative, le déplacement est effectué en partant de la fin du `ResultSet`. Si le numéro d'enregistrement n'existe pas le pointeur est positionné sur l'enregistrement BOF si la valeur est négative et inférieure au nombre d'enregistrements ou sur l'enregistrement EOF si la valeur est positive et supérieure au nombre d'enregistrements. Cette méthode retourne `true` si le pointeur est positionné sur un enregistrement valide et `false` dans le cas contraire (BOF ou EOF).
- `boolean relative(int deplacement)` : déplace le curseur du nombre d'enregistrements spécifié par l'argument `deplacement`. Si la valeur de cet argument est positive le curseur descend dans le jeu d'enregistrements et si la valeur est négative le curseur remonte dans le jeu d'enregistrements. Cette méthode retourne `true` si le pointeur est positionné sur un enregistrement valide et `false` dans le cas contraire (BOF ou EOF).
- `void beforeFirst()` : déplace le pointeur d'enregistrements avant le premier enregistrement (BOF).
- `void afterLast()` : déplace le pointeur d'enregistrements après le dernier enregistrement (EOF).
- `boolean first()` : déplace le pointeur d'enregistrements sur le premier enregistrement. Cette méthode retourne `true` s'il y un enregistrement dans le `ResultSet` et `false` dans le cas contraire.
- `boolean last()` : déplace le pointeur d'enregistrements sur le dernier enregistrement. Cette méthode retourne `true` s'il y un enregistrement dans le `ResultSet` et `false` dans le cas contraire.
- `boolean next()` : déplace le pointeur d'enregistrements sur l'enregistrement suivant l'enregistrement courant. Cette méthode retourne `true` si le pointeur est sur un enregistrement valide et `false` dans le cas contraire (EOF).
- `boolean previous()` : déplace le pointeur d'enregistrements sur l'enregistrement précédent l'enregistrement courant. Cette méthode retourne `true` si le pointeur est sur un enregistrement valide et `false` dans le cas contraire (BOF).

Pour toutes ces méthodes sauf pour la méthode `next`, il faut obligatoirement que le `ResultSet` soit de type `SCROLL_SENSITIVE` ou `SCROLL_INSENSITIVE`. Si le `ResultSet` est de type `FORWARD_ONLY`, seule la méthode `next` fonctionne et dans ce cas les autres méthodes déclenchent une exception. Les méthodes suivantes permettent de tester la position du pointeur d'enregistrements :

- `boolean isBeforeFirst()` : retourne `true` si le pointeur est placé avant le premier enregistrement (BOF).
- `boolean isAfterLast()` : retourne `true` si le pointeur est placé après le dernier enregistrement (EOF).
- `boolean isFirst()` : retourne `true` si le pointeur est placé sur le premier enregistrement.
- `boolean isLast()` : retourne `true` si le pointeur est placé sur le dernier enregistrement.
- `int getRow()` : retourne le numéro de l'enregistrement sur lequel se trouve le pointeur d'enregistrements. La valeur 0 est retournée s'il n'y a pas d'enregistrement courant (BOF ou EOF).

```
public static void positionRs(ResultSet rs)
{
    try {
        if (rs.isBeforeFirst())
        {
            System.out.println("le pointeur est avant le
premier enregistrement");
        }
        if (rs.isAfterLast())
        {
            System.out.println("le pointeur est après le
dernier enregistrement");
        }
        if (rs.isFirst())
        {
            System.out.println("le pointeur est sur le premier
enregistrement");
        }
        if (rs.isLast())
        {
            System.out.println("le pointeur est sur le dernier
enregistrement");
        }
        int position;
        position=rs.getRow();
        if (position!=0)
        {
            System.out.println("c'est l'enregistrement numéro
" + position);
        }
    } catch (SQLException e) {
        // TODO Bloc catch autogénéré
        e.printStackTrace();
    }
}
```

b. Lecture des données dans un `ResultSet`

L'objet `ResultSet` fournit de nombreuses méthodes permettant la lecture des champs d'un enregistrement. Chacune de ces méthodes est spécifique à un type de données SQL. Il faut bien sûr utiliser en priorité la méthode adaptée au type du champ dont on souhaite obtenir la valeur. Cependant certaines de ces méthodes sont relativement souples et permettent la lecture de plusieurs types de données. Le tableau ci-après reprend les principaux types de données SQL et les méthodes en permettant la lecture à partir d'un `ResultSet`. Les méthodes marquées avec le symbole `□` sont les méthodes conseillées. Les méthodes marquées avec le symbole `□` sont possibles mais avec des risques de perte d'informations.

	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP	CLOB	BLOB	ARRAY	REF	STRUCT	JAVA OBJECT
getByte	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	
getShort	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	
getInt	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	
getLong	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	
getFloat	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	
getDouble	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	
getBigDecimal	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	
getBoolean	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	
getString	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	
getBytes															⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	
getDate														⊕	⊕	⊕		⊕	⊕	⊕					
getTime														⊕	⊕	⊕		⊕	⊕	⊕					
getTimeStamp														⊕	⊕	⊕		⊕	⊕	⊕					
getAsciiStream														⊕	⊕	⊕	⊕	⊕	⊕	⊕					
getUnicodeStream														⊕	⊕	⊕	⊕	⊕	⊕	⊕					
getBinaryStream														⊕	⊕	⊕									
getClob																		⊕							
getBlob																			⊕						
getArray																				⊕					
getRef																					⊕				
getCharacterStream														⊕	⊕	⊕	⊕	⊕	⊕	⊕					
getObject	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕

Chacune de ces méthodes est disponible sous deux formes. La première accepte comme argument le numéro de la colonne dont on souhaite obtenir la valeur. La numérotation commençant à 1. La deuxième version accepte une chaîne de caractères représentant le nom de la colonne dans la base de données. Si la requête ayant été utilisée pour créer le `ResultSet` contient des alias alors les colonnes portent le nom de l'alias et non le nom du champ dans la base de données. Pour une meilleure lisibilité du code, il est bien sûr préférable d'utiliser les noms des colonnes plutôt que leurs indices. Lorsque dans la base de données un champ ne contient pas de valeur (`DNULL`), les méthodes retournent une valeur égale à 0 pour les champs numériques, une valeur `false` pour les champs booléens et une valeur `null` pour les autres types. Dans certains cas, il y a donc un doute possible sur la valeur réellement présente dans la base. Par exemple, la méthode `getInt` peut retourner une valeur égale à zéro parce qu'il y a effectivement cette valeur dans la base de données ou parce que ce champ n'est pas renseigné dans la base de données. Pour lever ce doute, la méthode `wasNull` retourne un `boolean` égal à `true` si le champ sur lequel la dernière opération de lecture dans le `ResultSet` contenait effectivement une valeur `null`.

```
public static void lectureRs(Connection cnx)
{
    Statement stm;
    String requete;
    ResultSet rs;
    try
    {
        stm=cnx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
        ResultSet.CONCUR_READ_ONLY);
        requete="select * from products ";
        rs=stm.executeQuery(requete);
        System.out.println("code produit\tdésignation\tprix
unitaire\tstock\tépuisé\tDLC");
        while(rs.next())
        {
            System.out.print(rs.getInt("ProductID")+"\t");
        }
    }
}
```

```

System.out.print(rs.getString("ProductName")+"\t");

System.out.print(rs.getDouble("UnitPrice")+"\t");
        rs.getShort("UnitsInStock");
        if (rs.wasNull())
        {
            System.out.print("inconnu\t");
        }
        else
        {

System.out.print(rs.getShort("UnitsInStock")+"\t");
        }

System.out.print(rs.getBoolean("Discontinued")+"\t");
        if (rs.getDate("DLC")!=null)
        {
            System.out.println(rs.getDate("DLC"));
        }
        else
            System.out.println("non perissable");
        }
        rs.close();
        stm.close();
    }
    catch (SQLException e)
    {
        e.printStackTrace();
    }
}

```

c. Modification des données dans un ResultSet

La modification des données est effectuée simplement en utilisant les méthodes updateXXX où XXX correspond au type de données de la colonne à mettre à jour. Comme pour les méthodes getXXX celles-ci sont disponibles en deux versions, l'une attend comme argument l'indice de la colonne à mettre à jour, la deuxième attend elle comme argument une chaîne de caractères représentant le nom de la colonne dans la base de données. Si la requête ayant été utilisée pour créer le ResultSet contient des alias, alors les colonnes portent le nom de l'alias et non le nom du champ dans la base de données. Pour une meilleure lisibilité du code, il est bien sûr préférable d'utiliser les noms des colonnes plutôt que leurs indices.

Le type du deuxième argument attendu par ces méthodes correspond bien sûr au type de données à mettre à jour dans le ResultSet. Les modifications doivent ensuite être validées par la méthode updateRow ou annulées par la méthode cancelRowUpdates. Le ResultSet doit obligatoirement être de type CONCUR_UPDATABLE pour pouvoir être modifié. Dans le cas contraire, l'exécution d'une méthode updateXXX déclenche une exception.

```

public static void modificationRs(Connection cnx)
{
    Statement stm;
    String requete;
    ResultSet rs;
    int num=0;
    BufferedReader br;
    String reponse;
    try
    {
        stm=cnx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
        ResultSet.CONCUR_UPDATABLE);
        requete="select * from products " ;
        rs=stm.executeQuery(requete);
        System.out.println("numéro de ligne\tcode
produit\tdésignation\tprix unitaire\tstock\tépuisé\tDLC");
        while(rs.next())
        {

```

```

        num++;
        System.out.print(num + "\t");
        System.out.print(rs.getInt("ProductID")+"\t");

System.out.print(rs.getString("ProductName")+"\t");

System.out.print(rs.getDouble("UnitPrice")+"\t");
        rs.getShort("UnitsInStock");
        if (rs.wasNull())
        {
            System.out.print("inconnu\t");
        }
        else
        {

System.out.print(rs.getShort("UnitsInStock")+"\t");
        }

System.out.print(rs.getBoolean("Discontinued")+"\t");
        if (rs.getDate("DLC")!=null)
        {
            System.out.println(rs.getDate("DLC"));
        }
        else
            System.out.println("non perissable");
        }
br=new BufferedReader(new InputStreamReader(System.in));
        System.out.println("quelle ligne voulez-vous modifier ? ");
reponse=br.readLine();
rs.absolute(Integer.parseInt(reponse));
        System.out.println("désignation actuelle " +
rs.getString("ProductName"));
        System.out.println("saisir la nouvelle valeur ou enter
pour conserver la valeur actuelle");
        reponse=br.readLine();
        if (!reponse.equals(""))
        {
            rs.updateString("ProductName",reponse);
        }
        System.out.println("prix unitaire actuel " +
rs.getDouble("UnitPrice"));
        System.out.println("saisir la nouvelle valeur ou enter
pour conserver la valeur actuelle");
        reponse=br.readLine();
        if (!reponse.equals(""))
        {

rs.updateDouble("UnitPrice",Double.parseDouble(reponse));
        }
        rs.getShort("UnitsInStock");
        if (rs.wasNull())
        {
            System.out.println ("quantité en stock actuelle inconnue");
        }
        else
        {
            System.out.println("quantité en stock actuelle " +
rs.getShort("UnitsInStock"));
        }
        System.out.println("saisir la nouvelle valeur ou enter
pour conserver la valeur actuelle");
        reponse=br.readLine();
        if (!reponse.equals(""))
        {

rs.updateShort("UnitsInStock",Short.parseShort(reponse));
        }
        System.out.println("voulez-vous valider ces modifications
o/n");
        reponse=br.readLine();

```

```

        if (reponse.toLowerCase().equals("o"))
        {
            rs.updateRow();
        }
        else
        {
            rs.cancelRowUpdates();
        }
        System.out.println("les valeurs actuelles ");
        System.out.print(rs.getString("ProductName")+"\t");
        System.out.print(rs.getDouble("UnitPrice")+"\t");
        rs.getShort("UnitsInStock");
        if (rs.wasNull())
        {
            System.out.print("inconnu\t");
        }
        else
        {
            System.out.print(rs.getShort("UnitsInStock")+"\t");
        }
        rs.close();
        stm.close();
    }
    catch (SQLException e)
    {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

d. Suppression de données dans un ResultSet

La suppression d'une ligne est effectuée très simplement en positionnant le pointeur sur la ligne à supprimer puis en appelant la méthode `deleteRow`. La ligne est immédiatement supprimée du `ResultSet` et dans la base de données. La position du pointeur d'enregistrement après la suppression dépend du pilote de base de données utilisé. Certains pilotes déplacent le pointeur sur l'enregistrement suivant, d'autres le déplacent sur l'enregistrement précédent et enfin quelques-uns ne modifient pas la position du pointeur. Il faut dans ce cas utiliser une des méthodes de déplacement pour positionner le pointeur sur un enregistrement utilisable. Le `ResultSet` doit obligatoirement être de type `CONCUR_UPDATABLE` pour pouvoir y supprimer des données. Dans le cas contraire, l'exécution de la méthode `deleteRow` déclenche une exception.

```

public static void suppressionRs(Connection cnx)
{
    Statement stm;
    String requete;
    ResultSet rs;
    int num=0;
    BufferedReader br;
    String reponse;
    try
    {
        stm=cnx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
        requete="select * from products ";
        rs=stm.executeQuery(requete);
        System.out.println("numéro de ligne\tcode
produit\tdésignation\tprix unitaire\tstock\tépuisé\tDLC");
        while(rs.next())
        {
            num++;
            System.out.print(num + "\t");
            System.out.print(rs.getInt("ProductID")+"\t");

            System.out.print(rs.getString("ProductName")+"\t");

            System.out.print(rs.getDouble("UnitPrice")+"\t");

```

```

        rs.getShort("UnitsInStock");
        if (rs.wasNull())
        {
            System.out.print("inconnu\t");
        }
        else
        {

System.out.print(rs.getShort("UnitsInStock")+"\t");
        }

System.out.print(rs.getBoolean("Discontinued")+"\t");
        if (rs.getDate("DLC")!=null)
        {
            System.out.println(rs.getDate("DLC"));
        }
        else
            System.out.println("non périsable");
        }
        br=new BufferedReader(new InputStreamReader(System.in));
        System.out.println("quelle ligne voulez-vous supprimer ? ");
        reponse=br.readLine();
        rs.absolute(Integer.parseInt(reponse));
        rs.deleteRow();
        System.out.println("le pointeur est maintenant sur la
ligne " + rs.getRow());
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
}

```

e. Ajout de données dans un ResultSet

Chaque objet `ResultSet` contient une ligne spéciale destinée à l'insertion de données. Le pointeur d'enregistrement doit au préalable être positionné sur cette ligne spéciale avec l'instruction `moveToInsertRow`. Une fois que le pointeur est positionné sur cette ligne, celle-ci peut être mise à jour avec les méthodes `updateXXX`. L'insertion de la ligne doit ensuite être validée avec la méthode `insertRow`. Cette méthode provoque la mise à jour de la base de données. La ligne d'insertion devient à ce moment une ligne normale du `ResultSet` et le pointeur d'enregistrement est positionné sur cette ligne. Vous pouvez revenir sur la ligne sur laquelle vous étiez avant l'insertion grâce à la méthode `moveToCurrentRow`. Si vous ne fournissez pas de valeur pour toutes les colonnes, des valeurs `null` seront insérées dans la base de données pour les colonnes non renseignées. La base de données doit accepter les valeurs nulles pour ces champs sinon une exception est déclenchée. Le `ResultSet` doit obligatoirement être de type `CONCUR_UPDATABLE` pour pouvoir y insérer des données. Dans le cas contraire, l'exécution de la méthode `moveToInsertRow` déclenche une exception.

```

public static void ajoutRs(Connection cnx)
{
    Statement stm;
    String requete;
    ResultSet rs;
    int num=0;
    BufferedReader br;
    String reponse;
    try
    {
        stm=cnx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
        ResultSet.CONCUR_UPDATABLE);
        requete="select * from products ";
        rs=stm.executeQuery(requete);
        br=new BufferedReader(new InputStreamReader(System.in));
        System.out.println("saisir les valeurs de la nouvelle ligne");
        rs.moveToInsertRow();
        System.out.print("code produit : ");
        reponse=br.readLine();

```

```

        rs.updateInt ("ProductID",Integer.parseInt(reponse));
        System.out.print("Désignation : ");
        reponse=br.readLine();
        rs.updateString ("ProductName",reponse);
        System.out.print("Prix unitaire : ");
        reponse=br.readLine();
        rs.updateDouble("UnitPrice",Double.parseDouble(reponse));
        System.out.print("Quantité en stock : ");
        reponse=br.readLine();
        rs.updateDouble("UnitsInStock",Short.parseShort(reponse));
        rs.insertRow();
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
}

```

6. Gestion des transactions

Les transactions vont permettre de garantir qu'un ensemble d'instructions SQL sera exécuté avec succès pour chacune d'elles ou qu'aucune d'entre elles ne sera exécutée. Le transfert d'une somme d'argent entre deux comptes bancaires représente l'exemple classique où une transaction est nécessaire. Imaginez la situation suivante : notre banque doit effectuer l'encaissement d'un chèque de 1000 € à débiter sur le compte numéro 12345 et à créditer sur le compte 67890. Par mesure de sécurité après chaque opération effectuée sur un compte (débit ou crédit), un rapport est édité. Voici ci-dessous un extrait du code pouvant effectuer ces opérations.

```

public static void mouvement(String compteDebit,String compteCredit,double somme)
{
    try
    {

        Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
        Connection cnx=null;

        cnx=DriverManager.getConnection("jdbc:sqlserver://localhost;database
Name=banque; user=sa;password=");
        PreparedStatement stm;
        stm=cnx.prepareStatement("update comptes set solde=solde
+ ? where numero=?");
        stm.setDouble(1,somme * -1);
        stm.setString(2,compteDebit);
        stm.executeUpdate();
        impressionRapport(compteDebit, somme);
        stm.setDouble(1,somme);
        stm.setString(2,compteCredit);
        stm.executeUpdate();
        impressionRapport(compteCredit, somme);
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
}

```

Pour 99,9999 % des mouvements effectués avec ce code, il n'y a aucun problème, sauf qu'un beau jour l'imprimante chargée des éditions se bloque et ce blocage déclenche une exception dans la méthode `impressionRapport`. A priori cette exception ne pose pas de problème puisque l'appel de cette méthode est placé dans un bloc `try` et qu'un bloc `catch` traite l'exception. Il ne faut cependant pas oublier que si une exception est déclenchée et qu'un bloc `catch` est exécuté, l'exécution se poursuit par l'instruction suivant le bloc `catch`. Dans ce cas de figure, les instructions placées entre celle ayant déclenché l'exception et la fin du bloc `try` ne sont tout simplement pas exécutées. Dans notre cas, ceci peut être très problématique si cette exception est déclenchée dans la méthode `impressionRapport` exécutée aussitôt après l'opération de débit. L'opération de crédit n'a tout

simplement pas lieu. La somme est donc perdue. Une solution consiste à annuler l'effet des instructions SQL ayant déjà été exécutées en effectuant le traitement inverse, dans notre cas en créditant le compte.

C'est en fait ce mécanisme qui est mis en œuvre dans une transaction mais bien sûr de manière automatique. C'est au niveau de la connexion vers le serveur de base de données que sont gérées les transactions.

a. Mise en œuvre des transactions

Jusqu'à présent, nous ne nous sommes pas préoccupés des transactions et pourtant nous en réalisons depuis notre première instruction SQL exécutée via JDBC. Le fonctionnement par défaut de jdbc consiste effectivement à inclure chaque instruction exécutée dans une transaction puis valider la transaction si l'instruction a été exécutée correctement (`commit`) ou à annuler l'instruction dans le cas contraire (`rollback`). Ce mode de fonctionnement est appelé mode `autoCommit`. Si vous souhaitez gérer vous-même la fin d'une transaction en validant ou en annulant toutes les instructions qu'elle contient, vous devez désactiver le mode `autoCommit` en appelant la méthode `setAutoCommit(false)` sur l'objet `Connection`. Vous êtes maintenant responsables de la fin des transactions. Les méthodes `commit` et `rollback` de l'objet `Connection` permettent de valider ou d'annuler les instructions exécutées depuis le début de la transaction. Une nouvelle transaction débute automatiquement dès la fin de la précédente ou dès l'ouverture de la connexion. Le code de notre méthode permettant de transférer un montant entre deux comptes doit donc prendre la forme suivante.

```
public static void mouvement1(String compteDebit, String compteCredit, double somme)
{
    Connection cnx=null;
    try
    {

        Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
        cnx=DriverManager.getConnection("jdbc:sqlserver://localhost;database
Name=banque; user=sa;password=");
        cnx.setAutoCommit(false);
        PreparedStatement stm;
        stm=cnx.prepareStatement("update comptes set solde=solde
+ ? where numero=?");
        stm.setDouble(1,somme * -1);
        stm.setString(2,compteDebit);
        stm.executeUpdate();
        impressionRapport(compteDebit, somme);
        stm.setDouble(1,somme);
        stm.setString(2,compteCredit);
        stm.executeUpdate();
        impressionRapport(compteCredit, somme);
        cnx.commit();
    }
    catch (Exception e)
    {
        try {
            cnx.rollback();
        }
        catch (SQLException e1)
        {
            e1.printStackTrace();
        }
        e.printStackTrace();
    }
}
}
```

b. Points de sauvegarde

Lors de l'appel de la méthode `rollback`, l'ensemble des instructions SQL exécutées depuis le début de la transaction sont annulées. Cette méthode propose une deuxième version acceptant comme paramètre un objet `SavePoint`. Cet objet représente un repère dans l'exécution des instructions SQL. Il est créé par la méthode `setSavePoint`. L'appel de la méthode `rollback` avec comme argument un

objet SavePoint provoque l'annulation de toutes les instructions exécutées jusqu'à ce point de sauvegarde.

c. Niveaux d'isolement

Pendant qu'une transaction est active, les données modifiées par les instructions exécutées au sein de la transaction peuvent être verrouillées par la base de données pour éviter les conflits et les incohérences. Différents types de verrous sont disponibles pour une transaction. Pour bien comprendre leurs effets, il faut au préalable identifier les types de problèmes pouvant être rencontrés lorsqu'une transaction est active.

- **Lecture erronée** : cette anomalie se produit lorsqu'une application accède à des données qui sont en train d'être modifiées par une transaction qui n'a pas encore été validée.
- **Lecture non reproductible** : cette anomalie se produit lorsque les exécutions successives d'une même instruction select ne produisent pas le même résultat. C'est le cas si les données que vous lisez sont en cours de modification par une autre transaction.
- **Lecture fantôme** : cette anomalie se produit si des exécutions successives d'une même requête renvoient des données en plus ou en moins. Cela peut être le cas si une autre transaction est en train de supprimer ou d'ajouter des données à la table.

JDBC prévoit plusieurs niveaux d'isolement. Ces niveaux d'isolation déterminent la manière dont sont verrouillées les données durant la transaction. Ce verrouillage peut être placé en lecture, en écriture ou encore en lecture et en écriture sur les données accédées par les instructions de la transaction.

La méthode setTransactionIsolationLevel permet de fixer les types de problèmes pouvant être évités. Cette méthode accepte comme argument une des constantes présentées dans le tableau suivant. Ce tableau présente également l'effet de chaque niveau d'isolement sur les données.

Constante	Lecture erronée	Lecture non reproductible	Lecture fantôme
TRANSACTION_READ_UNCOMMITTED	possible	possible	possible
TRANSACTION_READ_COMMITTED	impossible	possible	possible
TRANSACTION_REPEATABLE_READ	impossible	impossible	possible
TRANSACTION_SERIALIZABLE	impossible	impossible	impossible

Le fait de spécifier un niveau d'isolement pour une transaction peut parfois avoir un effet sur les autres applications accédant aux données manipulées dans la transaction puisque celles-ci peuvent être verrouillées et donc être inaccessibles aux autres applications.

Archives Java

1. Présentation

Une archive Java est un format de fichier particulier permettant de regrouper dans un seul fichier (l'archive) plusieurs autres fichiers. En général, l'on regroupe ainsi tous les fichiers nécessaires au fonctionnement d'une application. Ceci comprend bien sûr les fichiers `.class` mais également toutes les autres ressources indispensables au bon fonctionnement de l'application. Cette possibilité de regroupement procure de nombreux avantages pour le déploiement d'applications.

- Le premier et certainement le plus appréciable réside dans le fait que le déploiement d'une application se résume à la recopie d'un seul et unique fichier sur le poste client. Même si l'application exige pour son fonctionnement plusieurs ressources organisées sous forme d'une arborescence bien précise. Cette arborescence est créée à l'intérieur du fichier archive et n'a pas besoin d'être reproduite sur le poste client.
- Les fichiers archive peuvent être compressés afin d'optimiser leur stockage et leur transfert sur un réseau.
- Cet avantage est encore plus appréciable pour les applets qui peuvent être ainsi récupérées par le navigateur avec une seule requête http.
- La sécurité est également améliorée par signature et le scellement de l'archive.
- Le format des archives étant standard, il n'y a aucune restriction vis-à-vis d'un système spécifique.

2. Manipulation d'une archive

La manipulation d'une archive Java (fichier `jar`) reprend les mêmes principes que la manipulation d'une archive dans le monde unix avec la commande `tar`. Les options de la commande `jar` permettant de manipuler une archive Java sont d'ailleurs étrangement ressemblantes à celles de la commande `tar` d'unix. Le format utilisé en interne par les fichiers archive est également bien connu puisqu'il s'agit du format ZIP. Les fichiers archive Java peuvent d'ailleurs être manipulés avec des outils dédiés à la manipulation de fichiers ZIP.

La commande standard de manipulation d'archive est la commande `jar`. Elle fait partie des outils fournis avec le jdk.

a. Crédation d'une archive

La syntaxe de base de création d'une archive Java est la suivante :

```
jar cf nomDuFichier listeFichier
```

Le paramètre `c` est bien sûr destiné à indiquer à la commande `jar` que l'on souhaite créer une archive. Le paramètre `f` indiquant, quant à lui, que la commande doit générer un fichier. Le nom du fichier est indiqué par le troisième paramètre de cette commande. Par convention, l'extension de ce fichier sera `.jar`. Le dernier élément représente le ou les fichiers à inclure dans l'archive. Si plusieurs fichiers sont à inclure dans l'archive, leurs noms doivent être séparés par un espace. Le caractère joker `*` est également autorisé dans la liste. Si un nom de répertoire est présent dans la liste, son contenu entier est ajouté à l'archive.

L'archive est générée dans le répertoire courant. Un fichier manifest par défaut est également ajouté à l'archive. Les options suivantes sont également disponibles.

- `v` affiche le nom des fichiers au fur et à mesure de l'ajout dans l'archive.
- `0` désactive la compression du fichier archive.
- `M` désactive la génération du manifest.

- **m** ajoute le manifest indiqué à l'archive.
- **-C** supprime le nom de répertoire dans l'archive.

b. Visualisation du contenu

Le contenu d'une archive peut être visualisé avec la commande suivante :

```
jar tf ardoise.jar
```

La commande affiche sur la console le contenu de l'archive.

```
META-INF/MANIFEST.MF
.classpath
.project
Client$1.class
Client$2.class
Client$3.class
Client$4.class
Client$5.class
Client$6.class
Client.class
ClientArdoiseMagique.class
PanelDessin.class
ThreadEntree.class
```

Des informations supplémentaires peuvent être obtenues en ajoutant l'option **v** à la commande. La date de modification et la taille des fichiers sont ajoutées au résultat de la commande.

```
jar tvf ardoise.jar

 59 Mon Feb 14 11:34:38 CET 2005 META-INF/MANIFEST.MF
 247 Tue Feb 08 19:07:22 CET 2005 .classpath
 568 Tue Feb 08 19:07:22 CET 2005 .project
1050 Mon Feb 14 08:35:58 CET 2005 Client$1.class
1527 Mon Feb 14 08:35:58 CET 2005 Client$2.class
3091 Mon Feb 14 08:35:58 CET 2005 Client$3.class
1023 Mon Feb 14 08:35:58 CET 2005 Client$4.class
1077 Mon Feb 14 08:35:58 CET 2005 Client$5.class
1182 Mon Feb 14 08:35:58 CET 2005 Client$6.class
6731 Mon Feb 14 08:35:58 CET 2005 Client.class
 530 Mon Feb 14 08:34:18 CET 2005 ClientArdoiseMagique.class
5585 Mon Feb 14 08:36:10 CET 2005 PanelDessin.class
3146 Mon Feb 14 08:39:36 CET 2005 ThreadEntree.class
```

Les chemins d'accès aux fichiers sont affichés avec le caractère / comme séparateur et sont relatifs à la racine de l'archive. Le contenu de l'archive n'est bien sûr pas modifié par l'exécution de cette commande.

c. Extraction

Les fichiers peuvent être extraits de l'archive avec la commande suivante :

```
jar xvf ardoise.jar
```

Les fichiers présents dans l'archive sont recréés sur le disque dans le répertoire courant. Si l'archive contient une portion d'arborescence, celle-ci est recréée dans le répertoire courant. Les éventuels fichiers et répertoires existants sont écrasés par ceux présents dans l'archive. L'extraction d'une archive peut être sélective en indiquant comme paramètre supplémentaire la liste des fichiers à extraire de l'archive en séparant les noms de ces fichiers par un espace. La commande suivante permet d'extraire de l'archive uniquement le fichier ClientArdoiseMagique.class.

```
jar xvf ardoise.jar ClientArdoiseMagique.class
```

Le contenu de l'archive n'est pas modifié par cette commande.

d. Mise à jour

Le contenu d'une archive peut être mis à jour par l'ajout de fichiers après sa création. Il faut dans ce cas utiliser la commande suivante :

```
jar uf ardoise.jar connect.gif
```

Le dernier paramètre de cette commande représente la liste des fichiers à mettre à jour dans l'archive. Si ces fichiers n'existent pas dans l'archive, ils sont ajoutés sinon ils sont remplacés par la nouvelle version. Si l'archive contient des répertoires, le chemin d'accès complet doit être spécifié dans la liste des fichiers à ajouter.

e. Exécution

Une application présente dans une archive Java peut être exécutée directement depuis l'archive sans avoir besoin d'extraire les fichiers. Vous devez indiquer à la machine virtuelle Java qu'elle doit elle-même extraire les fichiers de l'archive en utilisant l'option `-jar` lors du lancement de l'application. Le nom du fichier archive doit être spécifié à la suite de cette option.

```
java -jar ardoise.jar
```

La machine virtuelle Java a cependant besoin d'une information supplémentaire pour déterminer quel fichier de l'archive contient la méthode `main` par laquelle elle doit débuter l'exécution de l'application. Pour cela, elle recherche le fichier manifest de l'archive qui doit bien sûr contenir cette information pour que l'application puisse être exécutée à partir de l'archive.

3. Le manifest

Les fichiers archives Java sont bien plus que des simples fichiers compressés puisqu'ils proposent une multitude de fonctionnalités complémentaires :

- Exécution directe depuis l'archive.
- Signature du contenu de l'archive.
- Scellement de parties de l'archive.
- Gestion des versions.

Toutes ces fonctionnalités sont disponibles grâce au fichier manifest inclus dans l'archive.

a. Présentation

Le fichier manifest est un simple fichier texte contenant des couples nom de paramètre et valeur de paramètre. Ces deux informations sont séparés par le caractère `:`.

Ce fichier est toujours nommé `MANIFEST.MF` et se trouve dans le répertoire `META-INF` de l'archive.

b. Création

À la création d'un fichier archive, un fichier manifest est créé par défaut. Il contient les informations suivantes :

```
Manifest-Version: 1.0
Created-By: 1.8.0-ea (Oracle Corporation)
```

La première ligne indique la version du fichier manifest, la seconde indique la version du jdk avec laquelle

le fichier archive a été généré.

Pour ajouter d'autres informations au fichier manifest, vous devez procéder en deux étapes. Vous devez dans un premier temps préparer un fichier texte contenant les informations que vous souhaitez inclure dans le manifest de l'archive. La dernière ligne de ce fichier doit obligatoirement se terminer par un caractère retour chariot ou saut de ligne (ou les deux). Vous devez ensuite fusionner ce fichier texte avec le manifest par défaut de l'archive en utilisant l'option m de la commande jar. La syntaxe de la commande est donc la suivante :

```
jar cfm ardoise.jar infos.txt *
```

Cette commande génère un fichier archive nommé **ardoise.jar** contenant tous les fichiers du répertoire courant et ajoute au manifest par défaut les informations contenues dans le fichier **infos.txt**. Ce fichier peut par exemple contenir une information permettant d'indiquer le nom de la classe contenant la méthode main par laquelle doit débuter l'exécution de l'application. Le fichier **infos.txt** contient dans notre cas la ligne suivante :

```
Main-Class: ClientArdoiseMagique
```

Ne pas oublier le retour chariot à la fin de la ligne et l'espace après le caractère :.

L'archive est générée avec le fichier manifest suivant :

```
Manifest-Version: 1.0  
Created-By: 1.8.0-ea (Oracle Corporation)  
Main-Class: ClientArdoiseMagique
```

La version de la commande jar fournie avec le jdk 8 propose également l'option e permettant d'indiquer le point d'entrée dans l'application sans avoir à créer de fichier intermédiaire. La syntaxe peut donc être la suivante :

```
jar cvfe ardoise.jar ClientArdoiseMagique.class *
```

4. Scellement et signature d'une archive

Ces deux opérations vont permettre de renforcer la sécurité d'une application. Le scellement va garantir que toutes les classes d'un package proviennent d'un seul et unique fichier jar.

La signature du fichier permet de garantir l'origine du fichier jar.

a. Scellement

Lors de nos premiers pas en programmation objet, nous avons vu qu'il est possible de restreindre la visibilité des membres d'une classe. Le mot clé `protected` permet de limiter la visibilité d'un élément à la classe où il est défini, aux sous-classes de celle-ci et aux autres classes faisant partie du même package.

Dans l'exemple ci-dessous, la méthode `calculAge` est déclarée `protected` et donc accessible depuis les autres classes du package `fr.eni.pk1`. Elle est par contre invisible des autres packages comme par exemple le package `fr.pirate.eni`.

```
package fr.eni.pk1;  
  
import java.time.LocalDate;  
import java.time.temporal.ChronoUnit;  
  
public class Personne  
{  
    private String nom;  
    private String prenom;
```

```

private LocalDate date_nais=LocalDate.of(1963,11,29);

public Personne()
{
}

public Personne(String n,String p,LocalDate d)
{
    this.nom=n;
    this.prenom=p;
    this.date_nais=d;
}

public String getNom()
{
    return nom;
}

public void setNom(String nom)
{
    this.nom = nom;
}

public String getPrenom()
{
    return prenom;
}

public void setPrenom(String prenom)
{
    this.prenom = prenom;
}

public LocalDate getDate_nais()
{
    return date_nais;
}

public void setDate_nais(LocalDate date_nais)
{
    this.date_nais = date_nais;
}

protected long calculAge()
{
    return date_nais.until(LocalDate.now(),ChronoUnit.YEARS);
}

```

La compilation de la classe suivante, définie dans un autre package, provoque une erreur.

```

package fr.pirate.eni;
import fr.eni.pk1.*;

public class Principale
{
    public static void main(String[] args)
    {
        Personne p;
        p=new Personne ();
        p.calculAge();
    }
}

```

```
Administrator : invite de commandes
D:\java8\code\tests\src\fr\pirate\eni>javac -cp pk1.jar Principale.java
Principale.java:10: error: calculAge() has protected access in Personne
    p.calculAge();
          ^
1 error
D:\java8\code\tests\src\fr\pirate\eni>_
```

Par contre, si nous utilisons le même nom de package que celui utilisé pour la classe Personne, il n'y a plus de problème.

Un utilisateur ayant à sa disposition le fichier jar peut donc facilement avoir accès aux éléments déclarés protected des classes. Pour éviter ce problème, il est possible de sceller le fichier jar contenant la classe Personne. En fait, on ne scelle pas le fichier jar mais les packages qu'il contient. En effectuant cette opération, nous indiquons à la machine virtuelle Java qui va utiliser l'archive que toutes les classes du package scellé sont dans l'archive. Si la machine virtuelle rencontre dans une autre archive une classe faisant partie du même package, elle déclenche une exception. Pour l'illustration ci-dessous, le fichier d'archive pk1.jar contient la classe fr.eni.pk1.Personne et le fichier pk2.jar contient la classe fr.eni.pk1.Principale. Il y a donc deux classes appartenant au même package dans deux fichiers jar. Le package fr.eni.pk1 a été scellé dans l'archive pk1.jar.

```
Administrator : invite de commandes
D:\java8\code\tests\src>java -cp pk1.jar;pk2.jar fr.eni.pk1.Principale
Exception in thread "main" java.lang.SecurityException: sealing violation
        at java.net.URLClassLoader.getAndVerifyPackage(Unknown Source)
        at java.net.URLClassLoader.defineClass(Unknown Source)
        at java.net.URLClassLoader.access$100(Unknown Source)
        at java.net.URLClassLoader$1.run(Unknown Source)
        at java.net.URLClassLoader$1.run(Unknown Source)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(Unknown Source)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        at fr.eni.pk1.Principale.main(Principale.java:21)
D:\java8\code\tests\src>_
```

Pour sceller un package, il faut ajouter au fichier Manifest de l'archive les informations suivantes.

```
Name: nom du package à sceller
Sealed: true
```

Ces informations doivent être ajoutées au moment de la création de l'archive. Il faut pour cela créer un fichier texte contenant les informations à ajouter au manifest de l'archive.

Exemple de fichier (protection.txt) :

```
Name: fr/eni/pk1/
Sealed: true
```

Vous devez utiliser le caractère / pour séparer les termes du nom de package et ne pas oublier le / final.

Ce fichier est ensuite utilisé dans la ligne de commande de création de l'archive.

```
jar -cvfm pk1.jar ./protection.txt fr/eni/pk1
```

b. Signature

Lorsque vous transmettez un document papier à l'un de vos correspondants, généralement vous apposez votre signature sur ce document pour garantir que vous êtes bien l'auteur du document. Lorsque le destinataire reçoit le document, il peut grâce à votre signature vérifier que vous en êtes bien l'auteur. La signature d'une archive jar utilise le même principe, le but étant toujours de permettre au destinataire la vérification de l'identité de l'auteur du document. Pour que ce mécanisme soit efficace, il faut que vous fassiez confiance au signataire du document. Imaginez qu'une administration vous demande une copie certifiée conforme de votre carte d'identité. Vous effectuez une photocopie du document et vous ajoutez la mention « conforme à l'original » puis votre signature. Il y a de fortes chances pour que l'administration refuse ce document puisque vous en êtes l'auteur et que vous avez vous-même certifié sa conformité. Par contre, si après avoir effectué une copie du document, vous vous rendez dans un organisme officiel (mairie, gendarmerie, commissariat de police) et que c'est cet organisme qui valide l'authenticité du document en y apposant un cachet, le destinataire ne peut plus avoir de doute. Dans la signature d'une archive jar, ce mécanisme est mis en œuvre par l'intermédiaire d'un certificat. Celui-ci va jouer le même rôle que le cachet de la mairie ou de la gendarmerie qui valide votre document papier.

Il faut également garantir que le document ne puisse pas être modifié après son authentification ou, s'il l'est, que cette modification soit détectable. Généralement vous rédigez ces documents avec un stylo et non avec un crayon pour éviter qu'un coup de gomme permette la modification discrète du document. Dans une archive jar signée, une empreinte SHA de chaque fichier est calculée puis stockée dans le manifest de l'archive. L'utilisateur de l'archive effectuera le même calcul d'empreinte et vérifiera que le résultat est conforme à ce qui est indiqué dans le manifest.

Pour pouvoir signer une archive jar, il vous faut le "matériel" adapté :

- l'outil qui va permettre de placer les sceaux sur une archive. C'est l'utilitaire `jarsigner` du jdk qui est utilisé.
- le bâton de cire pour créer les sceaux. Deux solutions sont disponibles pour se procurer cet élément. L'acheter dans un "magasin spécialisé" ou le fabriquer nous-mêmes. Les magasins s'appellent des autorités de certification. Ils peuvent vous fournir tous les éléments nécessaires à la signature d'une archive jar. Vous aurez à votre disposition une clé privée qui va servir à signer réellement l'archive, une clé publique qui va servir à vérifier l'authenticité de la signature. Cette clé publique est certifiée par l'autorité de certification pour garantir qu'elle vous est bien attribuée.

Si vous ne souhaitez pas acheter ces éléments, vous pouvez les générer vous-même grâce à l'utilitaire `keytool` du jdk.

La ligne de commande ci-dessous lance cet utilitaire pour la création de ces éléments.

```
keytool -genkey -alias java8 -keystore certifs
```

Le paramètre `-alias` indique le nom associé à la clé. Le paramètre `-keystore` indique le nom du fichier où sont enregistrés les éléments générés. Des questions supplémentaires vous seront posées par cet utilitaire afin qu'il recueille les informations dont il a besoin.

```
Entrez le mot de passe du fichier de clés :  
Ressaisissez le nouveau mot de passe :  
Quels sont vos nom et prénom ?  
[Unknown]: groussard thierry  
Quel est le nom de votre unité organisationnelle ?  
[Unknown]: eni  
Quel est le nom de votre entreprise ?  
[Unknown]: editions eni  
Quel est le nom de votre ville de résidence ?  
[Unknown]: saint herblain  
Quel est le nom de votre état ou province ?  
[Unknown]: bzh  
Quel est le code pays à deux lettres pour cette unité ?  
[Unknown]: fr  
Est-ce CN=groussard thierry, OU=eni, O=editions eni,  
L=saint herblain, ST=bzh, C=fr ?  
[non]: oui
```

```
Entrez le mot de passe de la clé pour <java8>
    (appuyez sur Entrée s'il s'agit du mot de passe du fichier
de clés) :
Ressaisissez le nouveau mot de passe :
```

Maintenant que nous avons les éléments nécessaires, nous pouvons signer notre archive. Il faut bien qu'elle ait été générée au préalable.

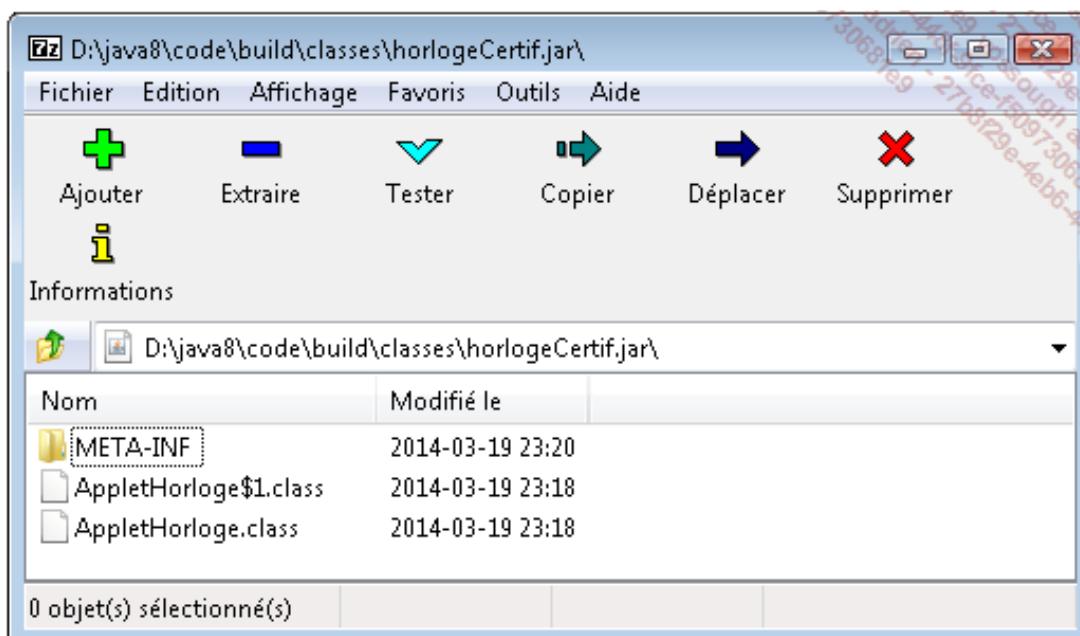
La ligne de commande permettant de lancer l'utilitaire `jarsigner` est relativement longue. La voici ci-après (pour des raisons de lisibilité, les paramètres sont ici placés sur plusieurs lignes mais ils doivent être saisis sur une seule ligne en les séparant par des espaces).

```
jarsigner -keystore certifs
    -storepass password
    -keypass password
    -tsa http://tsa.safecreative.org
    -signedjar horlogeCertif.jar
    horloge.jar
    java8
```

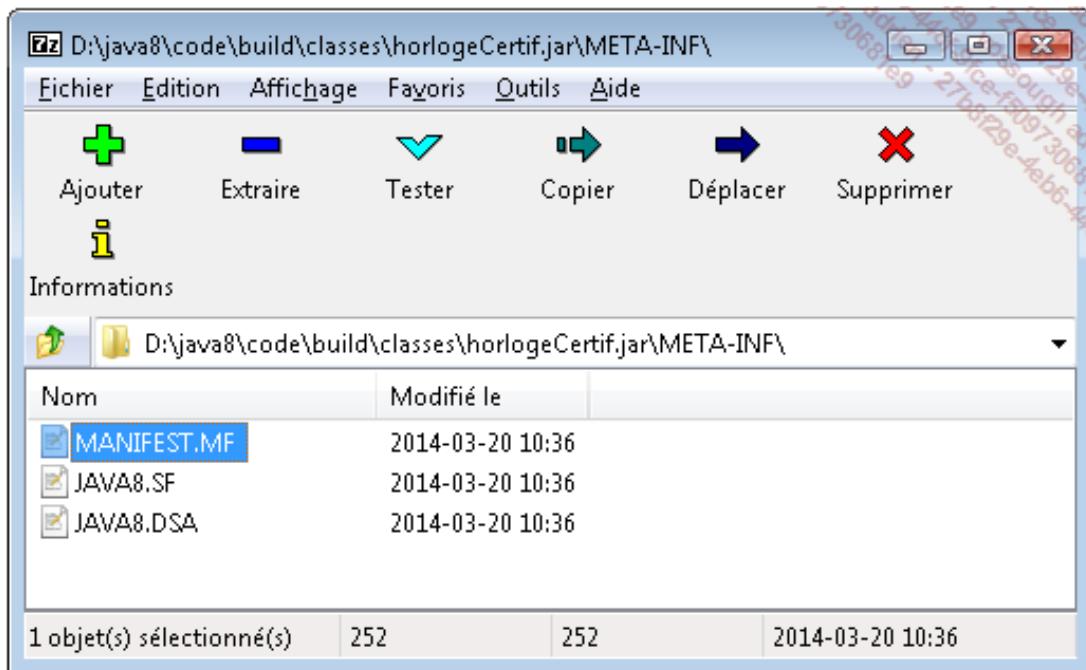
Voici le rôle de chacun de ces paramètres :

- `-keystore` : indique le nom du fichier contenant les clés à utiliser pour la signature.
- `-storepass` : mot de passe pour accéder au contenu du fichier.
- `-keypass` : mot de passe associé aux clés.
- `-tsa` : URL d'un service fournissant un Time Stamp utilisé pour dater la signature de l'archive.
- `-signedjar` : nom du fichier jar signé.
- le nom du fichier à signer.
- le nom du signataire.

Nous pouvons ensuite vérifier le contenu du fichier jar généré avec un utilitaire 7Zip par exemple :



Nous avons nos fichiers compilés et le dossier `META-INF` dans lequel se trouve le manifest ainsi qu'un fichier `.SF` et un fichier `.DSA`.



Le fichier .DSA contient la signature de l'archive générée avec la clé privée du signataire ainsi que le certificat contenant la clé publique utilisable pour la vérification de l'archive. Le contenu de ce fichier est bien sûr incompréhensible pour un humain.

Le fichier .SF contient l'empreinte SHA des différents fichiers de l'archive. Ces informations sont également présentes dans le manifest.

```
Manifest-Version: 1.0
Created-By: 1.8.0-ea (Oracle Corporation)

Name: AppletHorloge$1.class
SHA-256-Digest: X4tyEOAePgqTqGxA9nnvWBmwmS8Iyyg0whulzEYWzS0=

Name: AppletHorloge.class
SHA-256-Digest: 3517G6dfPZ3SYq4aAVP6UmxlDuyhtP0pQLfpwq+0qYM=
```

Maintenant que l'archive est signée, nous pouvons vérifier la réaction d'un navigateur lorsqu'il lance l'exécution d'une applet contenue dans l'archive. Nous avons une boîte de dialogue similaire à celle-ci qui s'affiche.

Avertissement de sécurité

Voulez-vous exécuter l'application ?



Nom : AppletHorloge
Editeur : INCONNU
Emplacement : file://

L'exécution d'applications par des éditeurs inconnus sera bloquée dans une version ultérieure car elle peut s'avérer dangereuse et représenter un risque de sécurité.

Risque : cette application sera exécutée sans restriction d'accès, ce qui peut représenter un risque pour votre ordinateur et vos informations personnelles. Les informations fournies ne sont pas fiables ou sont inconnues. Il est donc recommandé de ne pas exécuter cette application à moins que vous n'en connaissiez la source.

Cette application sera bloquée lors d'une prochaine mise à jour de sécurité Java car le manifeste du fichier JAR ne contient pas l'attribut de droit d'accès. Pour plus d'informations, contactez l'éditeur. [Plus d'informations](#)

Cochez la case ci-dessous, puis cliquez sur Exécuter pour démarrer l'application

J'accepte le risque et je souhaite exécuter l'application.

Exécuter

Annuler

Le navigateur détecte toujours un risque potentiel avec l'applet. Il détecte bien que l'applet est signée mais il est incapable de valider le nom de l'éditeur de l'application. Le problème vient en fait du certificat utilisé pour signer l'archive. Celui-ci n'étant pas généré par une autorité de certification reconnue, le navigateur ne lui accorde aucune confiance. Pour éliminer cet avertissement, il faut absolument utiliser un certificat valide.

Java Web Start

1. Présentation

La technologie Java Web Start permet de démarrer l'exécution d'une application par un simple clic sur un lien sans aucune installation de l'application au préalable. Le lien pointe sur un fichier JNLP (*Java Network Launching Protocol*) contenant les informations pour que Java Web Start télécharge et exécute l'application.

L'application peut être déployée sur pratiquement n'importe quelle plate-forme et bien sûr exécutée sur une toute autre plate-forme. À la première exécution, l'application est stockée dans un cache local du poste client. Cette technique améliore la rapidité des lancements ultérieurs et permet également l'exécution de l'application même si aucune connexion réseau n'est disponible. Si une application nécessite une version particulière de la plate-forme Java et que celle-ci n'est pas disponible sur le poste, Java Web Start télécharge et installe automatiquement la bonne version. La sécurité est également prise en compte puisque Java Web Start n'autorise que des accès limités aux ressources disques et réseau utilisables par l'application.

2. Exécution d'une application

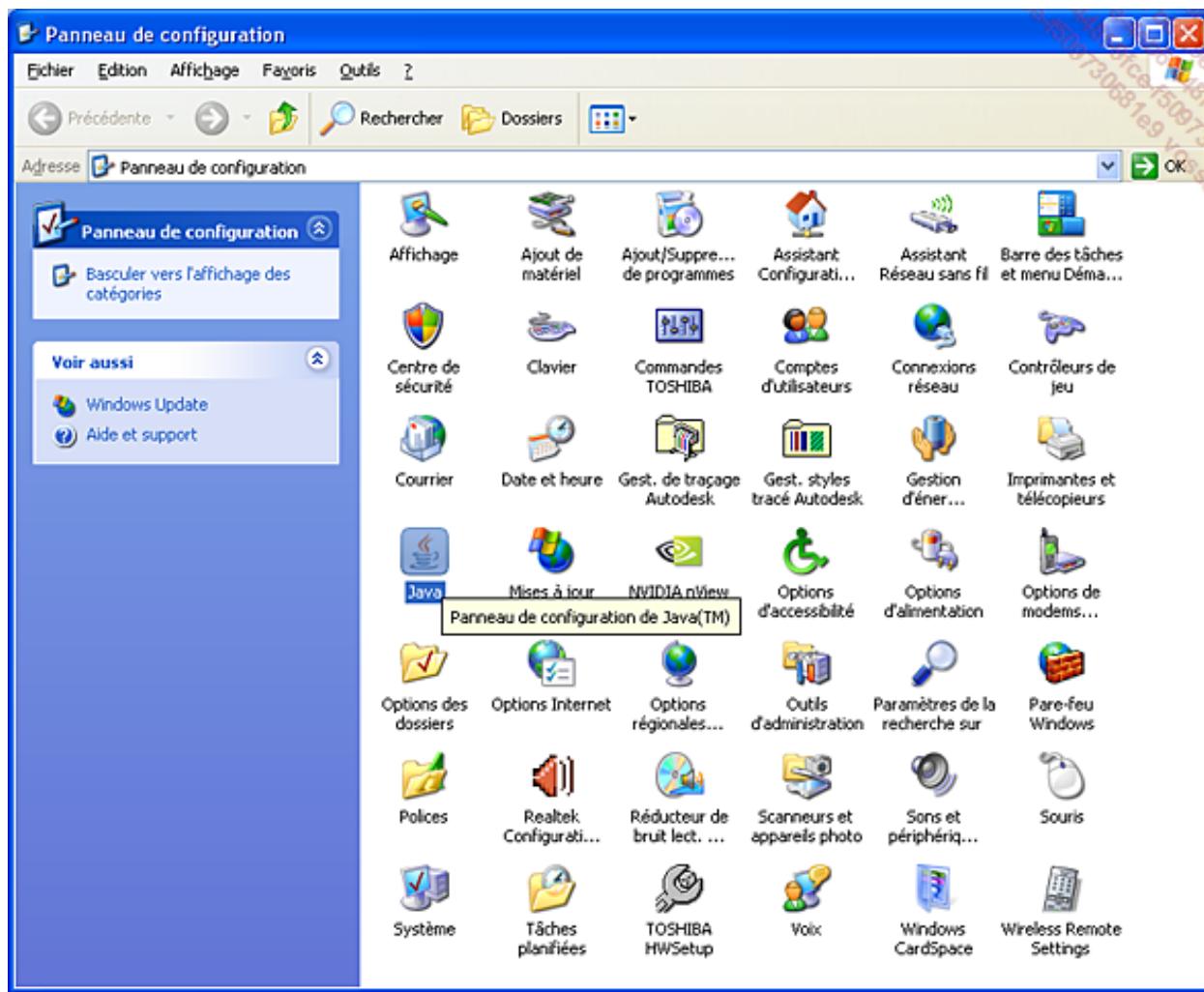
Deux solutions sont possibles pour l'exécution d'une application avec Java Web Start.

a. À partir d'un navigateur

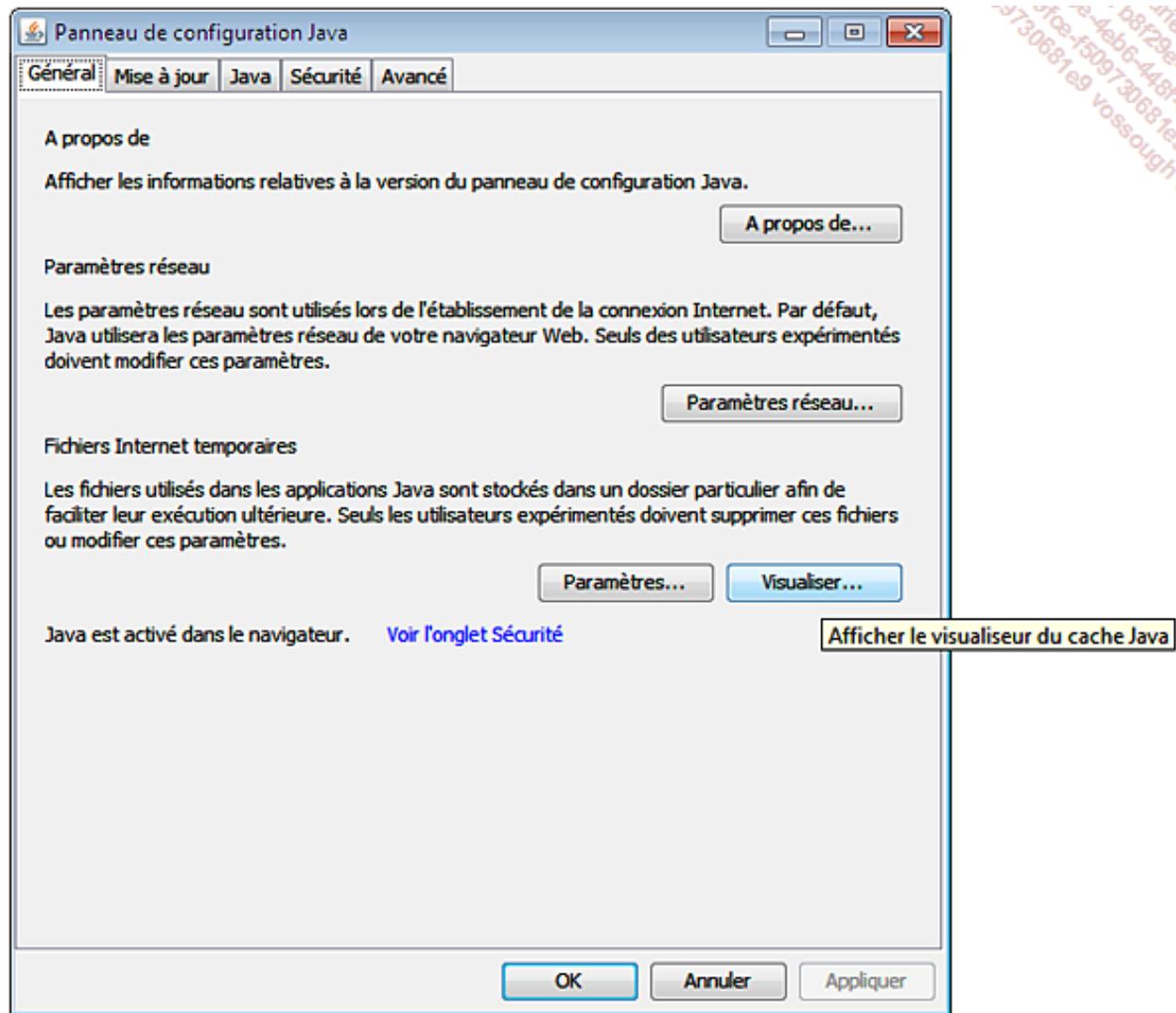
Le lancement de l'application s'effectue simplement en cliquant sur un lien pointant vers le fichier JNLP de l'application. Le fichier est téléchargé puis Java Web Start en extrait toutes les informations nécessaires au fonctionnement de l'application.

b. À partir du cache local

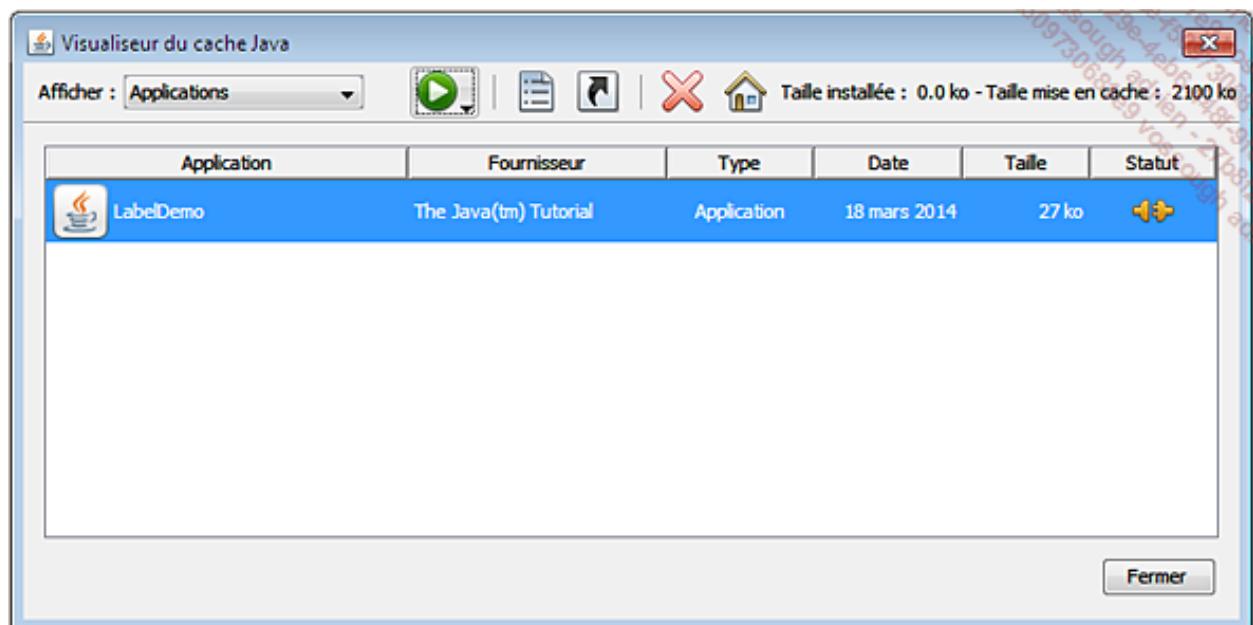
Une application peut également être exécutée à partir des fichiers mis en cache lors d'une exécution précédente. Vous devez utiliser le visionneur de cache Java pour afficher la liste des applications ayant déjà été exécutées. Le visionneur de cache est disponible via l'icône Java du Panneau de configuration.



- Sélectionnez ensuite l'option **Afficher** de la rubrique **Fichiers Internet temporaires** de l'onglet**Général**.



Le visionneur du cache Java affiche alors la liste des applications ayant déjà été utilisées.



Diverses options sont alors disponibles via les boutons suivants de la barre d'outils :



Exécute à nouveau l'application sélectionnée soit à partir du serveur (en ligne), soit à partir du cache Java (hors ligne).



Affiche le fichier JNLP de l'application.



Ajoute un raccourci sur le bureau permettant de lancer l'application comme une application classique.



Supprime l'application de la liste.



Affiche la page d'accueil de l'application dans le navigateur.

3. Déploiement d'une application

Le déploiement d'une application avec Java Web Start se décompose en quatre opérations :

- Configurer le serveur Web.
- Créer le fichier JNLP.
- Placer l'application sur le serveur Web.
- Créer la page Web d'accueil.

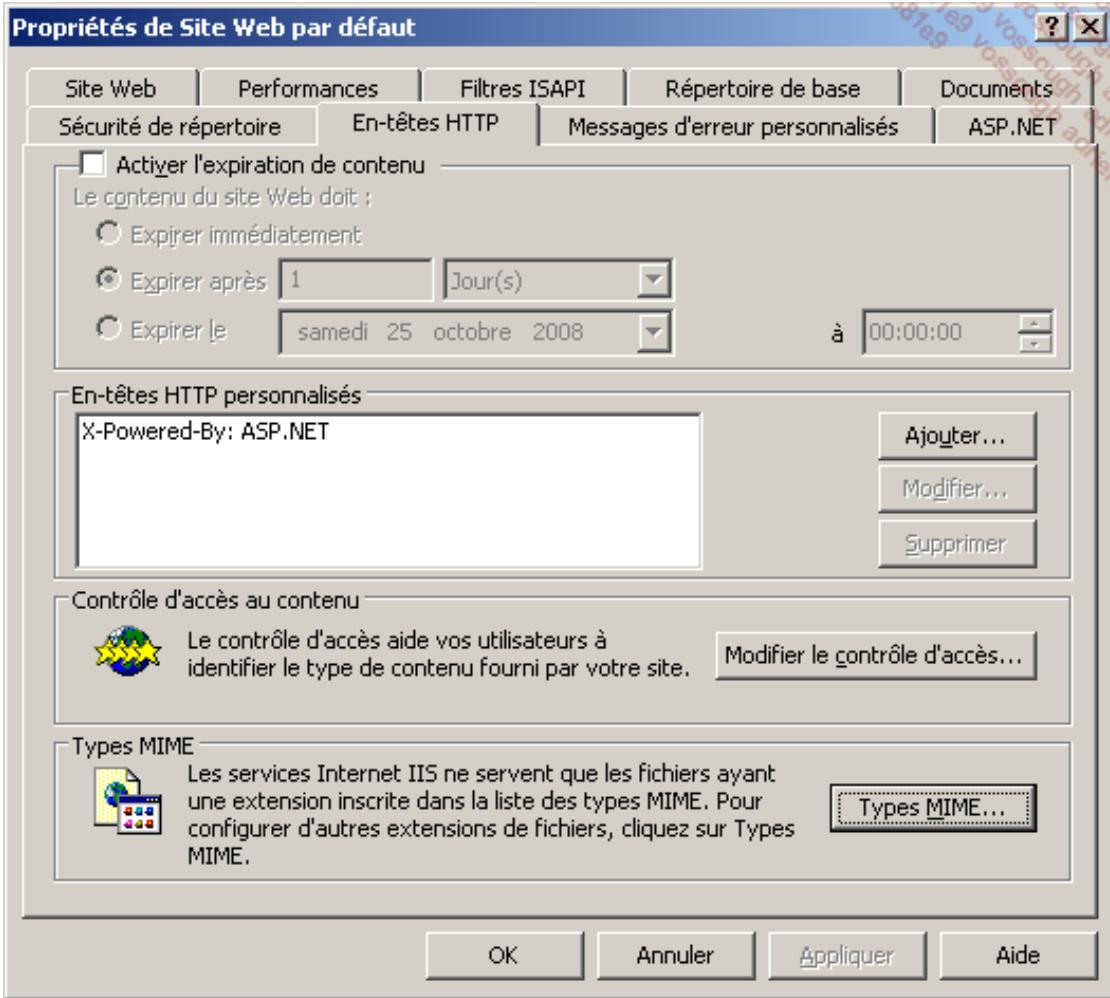
Nous allons donc détailler chacune de ces étapes.

a. Configuration du serveur Web

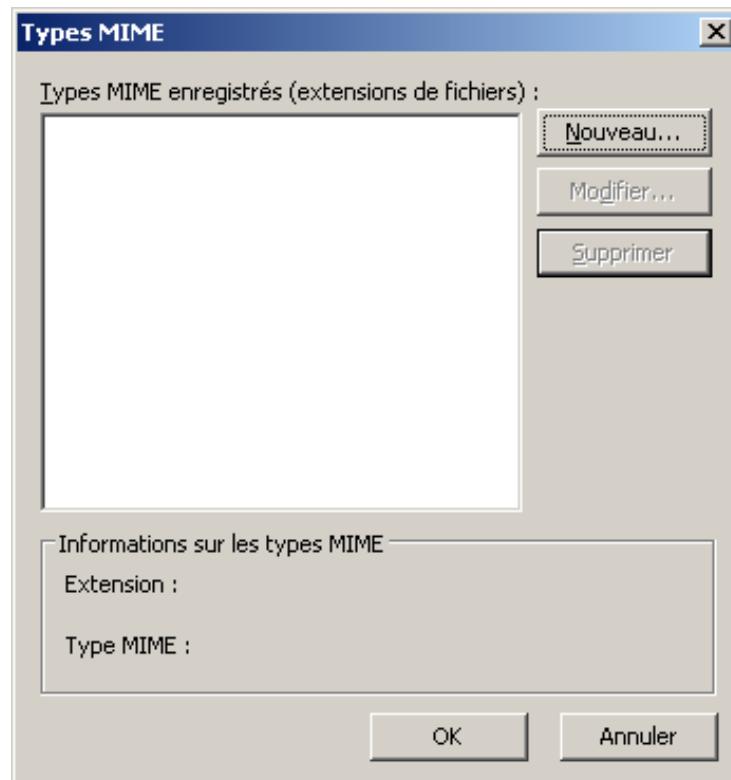
La seule modification nécessaire sur le serveur Web consiste à configurer le type MIME associé à l'extension de fichier .jnlp. Cette configuration est bien sûr propre à chaque type serveur. Pour un serveur Apache, il suffit simplement d'ajouter au fichier `mime.types` la ligne suivante :

```
application/x-java-jnlp-file JNLP
```

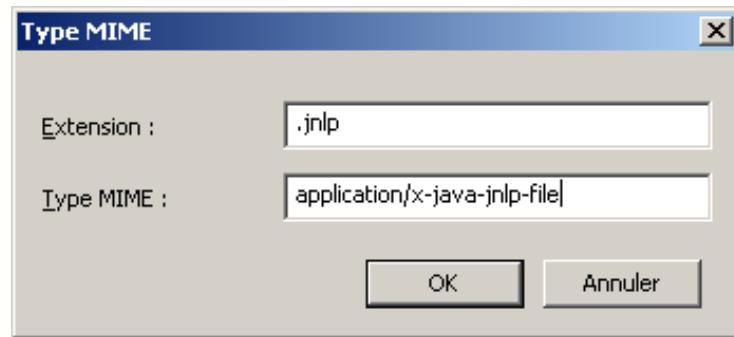
Pour un serveur IIS, il faut utiliser l'onglet **En-têtes HTTP** de la page de propriétés du serveur Web.



Le bouton **Types MIME** permet d'accéder à la boîte de dialogue de gestion des types MIME reconnus par le serveur Web.



Le bouton **Nouveau** affiche une boîte de dialogue permettant la saisie des informations concernant le type MIME à ajouter au serveur.

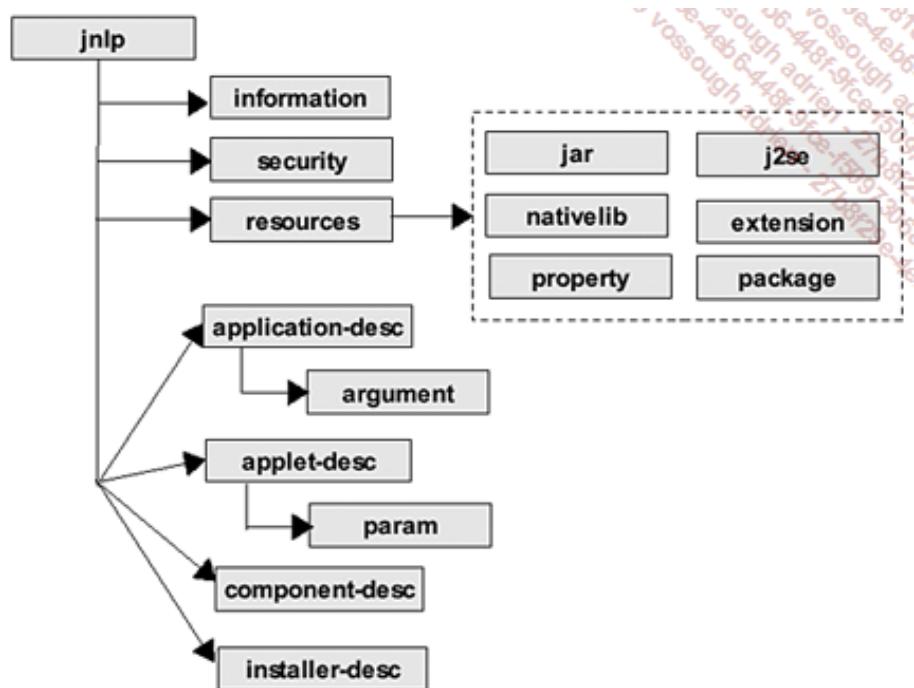


Il est souvent souhaitable de redémarrer le serveur pour qu'il prenne en compte les modifications de configuration.

b. Création du fichier JNLP

Le fichier JNLP est l'élément principal du déploiement avec Java Web Start. Ce fichier au format xml contient toutes les informations nécessaires pour l'exécution de l'application. Le format de ce fichier doit respecter les JSR-56 (*Java Specification Requests*).

Le diagramme ci-dessous représente le format attendu pour ce fichier.



L'élément **jnlp** est l'élément racine du fichier. Ses attributs décrivent les propriétés du fichier jnlp.

L'élément **information** est utilisé pour fournir les informations concernant l'application. Elles seront utilisées pendant l'installation de l'application.

L'élément **security** est utilisé pour obtenir un environnement sécurisé lors l'exécution de l'application.

L'élément **resources** indique quelles sont les ressources faisant partie de l'application.

Le fichier **jnlp** se termine par un élément **application-desc**, **applet-desc**, **component-desc** ou **installer-desc** en fonction du type d'application à déployer. Un seul de ces éléments doit être présent dans le fichier.

Voici ci-dessous un exemple de fichier jnlp :

```
<?xml version="1.0" encoding="utf-8"?>
```

```

<!-- fichier de déploiement pour l'ardoise magique -->

<jnlp spec="1.0+"
      codebase="http://thierry.eni.fr/ardoiseMagique"
      href="ardoise.jnlp">
  <information>
    <title>ardoise magique</title>
    <vendor>thierry groussard</vendor>
    <homepage
      href="http://thierry.eni.fr/ardoiseMagique/install.html"/>
      <description kind="short">cette application permet de
      partager un espace de dessin entre plusieurs utilisateurs
    </description>
    <offline-allowed/>
  </information>
  <resources>
    <jar href="ardoise.jar"/>
    <j2se version="1.6+"
          href="http://java.sun.com/products/autodl/j2se"/>
  </resources>
  <application-desc main-class="ClientArdoiseMagique"/>
</jnlp>

```

Nous allons détailler chacune des informations présentes dans ce fichier.

```
<?xml version="1.0" encoding="utf-8"?> :
```

Cette ligne indique qu'il s'agit d'un document conforme au standard xml 1.0 et que l'encodage des caractères utilisé est utf-8.

```
<!-- fichier de déploiement pour l'ardoise magique --> :
```

Ligne de commentaires dans un document xml.

```

<jnlp spec="1.0+"
      codebase="http://thierry.eni.fr/ardoiseMagique"
      href="ardoise.jnlp"> :

```

Balise racine du document jnlp.

L'attribut `spec` indique la version du protocole jnlp que doit accepter le client pour que l'installation soit possible. Dans notre cas, il faut que le client accepte la version 1.0 ou ultérieure. L'application pourra donc être installée par n'importe quel client.

L'attribut `codebase` indique l'emplacement racine de tous les autres documents référencés dans le fichier jnlp par des attributs `href`.

L'attribut `href` spécifie l'url relative du fichier jnlp. Cette information est combinée avec la valeur de l'attribut `codebase` pour obtenir une url absolue.

```
<title>ardoise magique</title>
```

Titre de l'application utilisé pour l'identifier dans le visionneur du cache Java.

```
<vendor>thierry groussard</vendor>
```

Nom du fournisseur de l'application affiché dans le visionneur du cache Java.

```
<homepage href="http://thierry.eni.fr/ardoiseMagique/install.html"/>
```

URL de la page d'accueil de l'application. C'est cette page qui doit contenir un lien vers le fichier jnlp.

```
<description kind="short">cette application permet de partager un espace de dessin entre plusieurs utilisateurs</description>
```

Texte de description rapide de l'application.

```
<offline-allowed/>
```

Indique que l'application peut être exécutée même si aucune connexion réseau n'est disponible. C'est dans ce cas la version mise en cache qui est exécutée. Si une connexion réseau est disponible, Java Web Start vérifie si une version plus récente de l'application est disponible sur le serveur. Si c'est le cas, c'est alors cette nouvelle version qui est exécutée. Sinon c'est la version mise en cache qui est exécutée.

```
<jar href="ardoise.jar"/>
```

Nom du fichier archive contenant l'application.

```
<j2se version="1.6+" href="http://java.sun.com/products/autodl/j2se"/>
```

Version du jre nécessaire pour le bon fonctionnement de l'application. Le signe + après le numéro de version, indique qu'il s'agit d'une version minimale nécessaire. Si une version ultérieure est disponible sur le poste client, l'application pourra être exécutée. Si le signe + n'est pas indiqué, Java Web Start exigera la version exacte. Si elle n'est pas disponible sur le poste client, l'attribut href indique où elle peut être téléchargée. Java Web Start propose alors à l'utilisateur d'effectuer ce téléchargement.

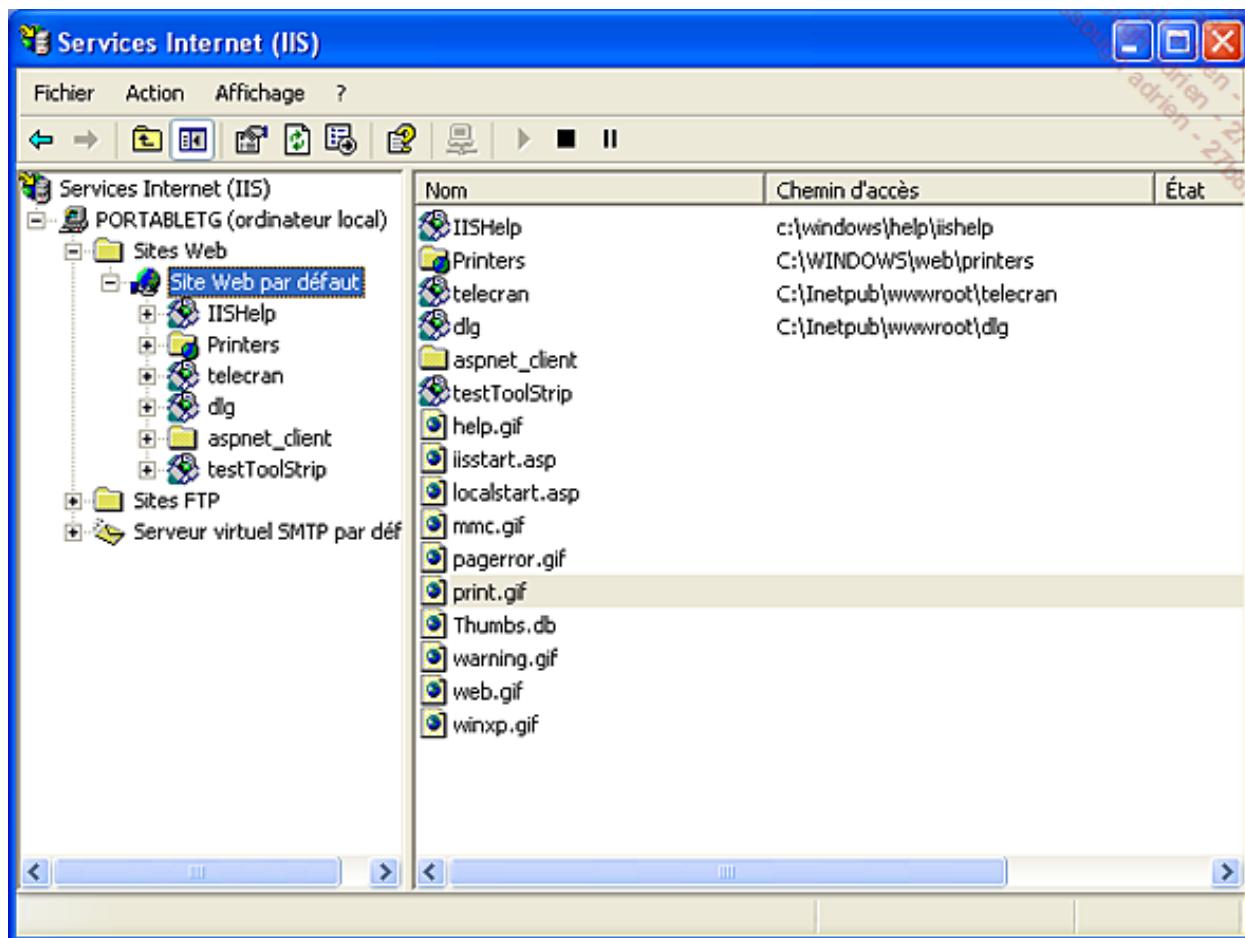
```
<application-desc main-class="ClientArdoiseMagique"/>
```

Indique que l'application à exécuter est une application Java autonome et non une applet. L'attribut main-class indique le nom de la classe contenant la méthode main permettant le démarrage de l'application. Cet attribut est optionnel si le fichier archive dispose d'un manifest contenant déjà cette information.

c. Déployer l'application sur le serveur

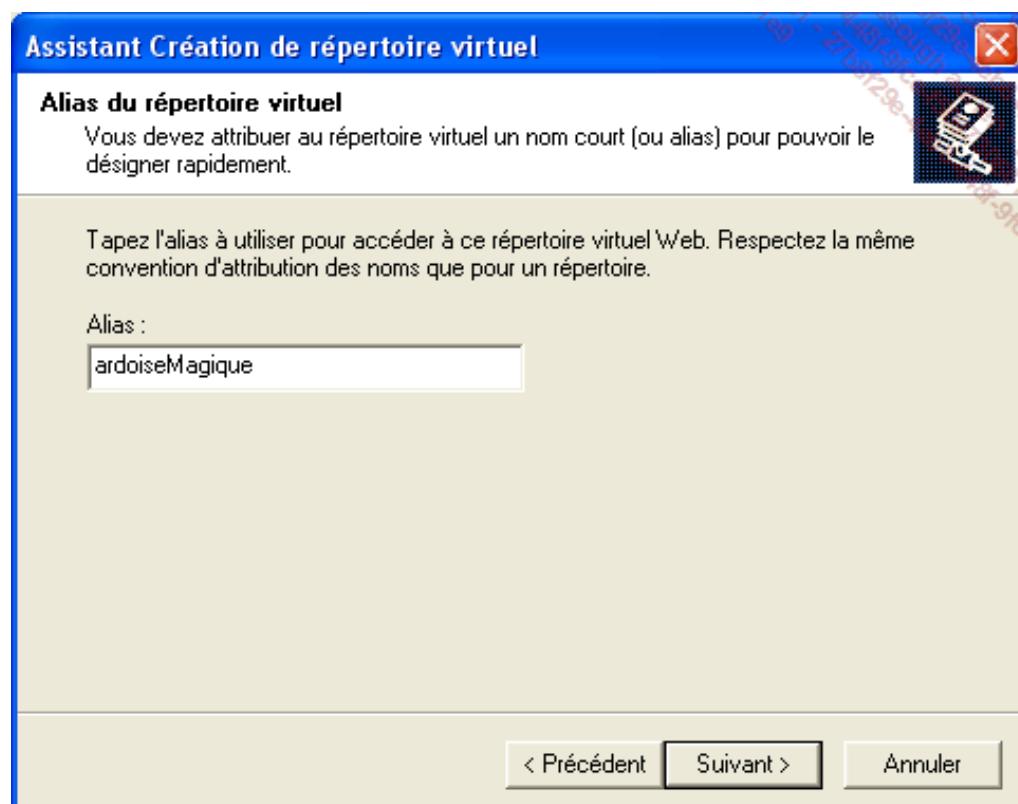
Cette étape est bien sûr spécifique à chaque serveur Web. En cas de doute, il est conseillé de contacter l'administrateur du serveur. Voici à titre d'exemple, les opérations à effectuer pour déployer l'application sur un serveur Web IIS de Windows.

- ➔ Ouvrez le Gestionnaire des services Internet par l'option **Outils d'administration** du Panneau de configuration. Accédez ensuite au site Web par défaut.



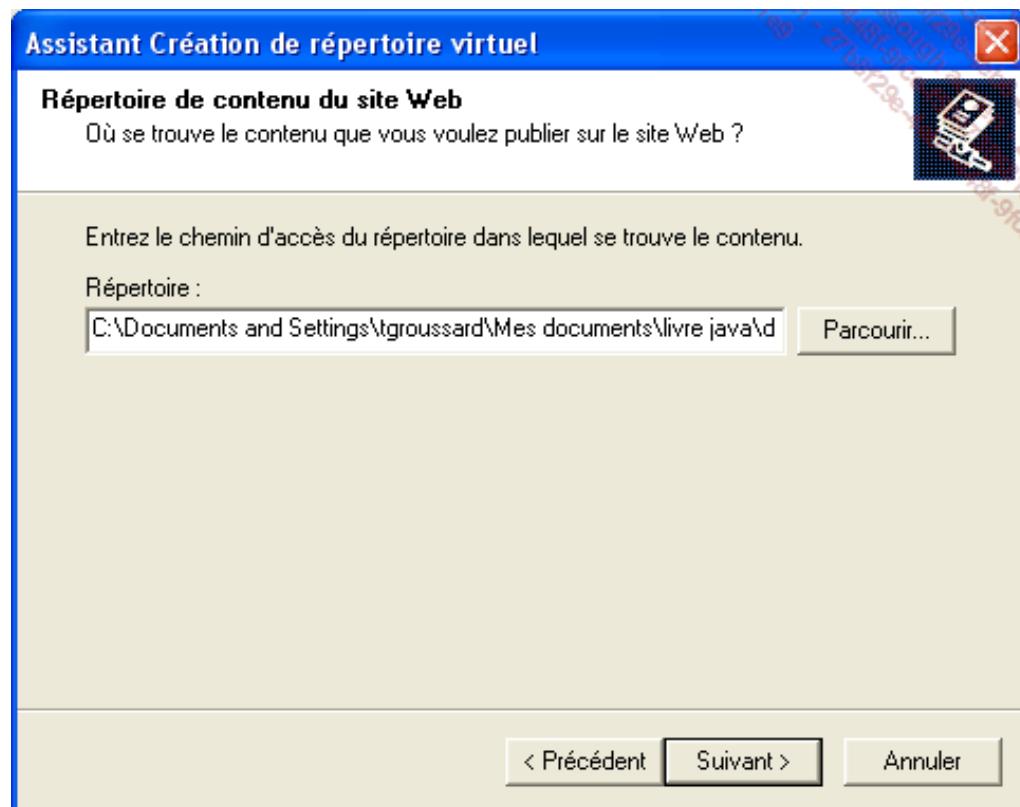
- Ajoutez ensuite un répertoire virtuel par le menu contextuel du site Web par défaut. Un assistant vous guide pour la création du répertoire virtuel.

Vous devez à la première étape fournir l'alias du répertoire virtuel. Il s'agit du nom utilisé dans l'URL pour atteindre cet emplacement.

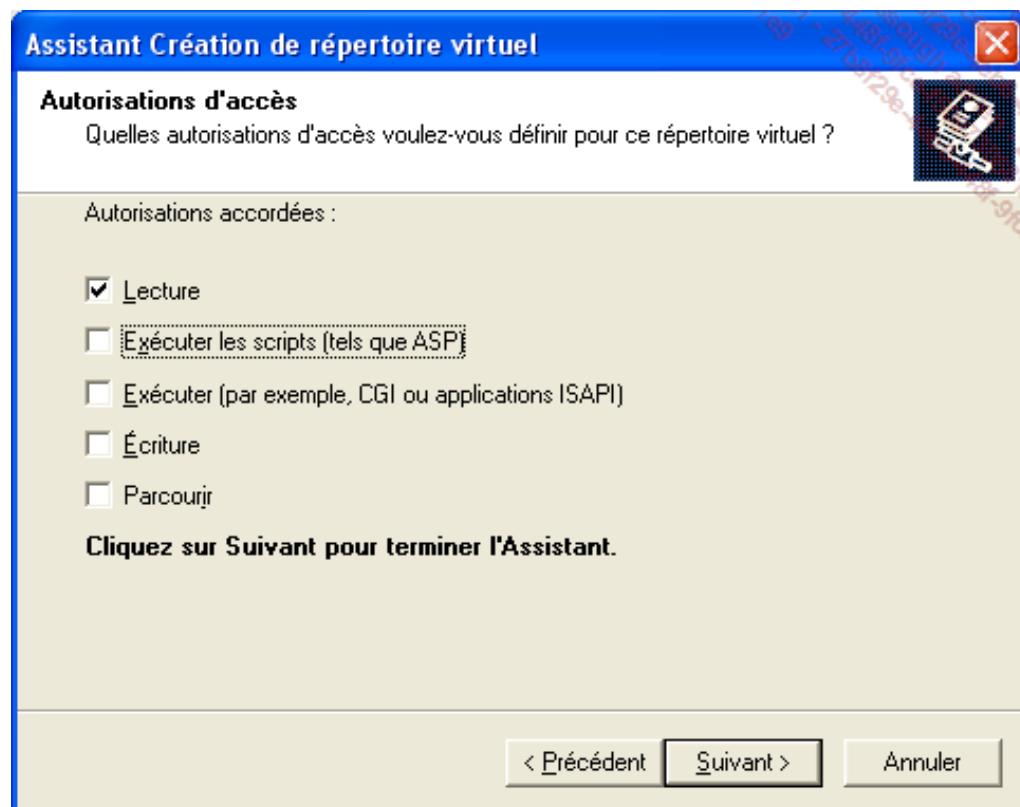


La deuxième étape vous demande de fournir l'emplacement où se trouvent les fichiers à publier. C'est en

général un répertoire d'un des disques de la machine, mais cela peut être également un partage réseau.



La dernière étape configure les autorisations d'accès accordées sur ce répertoire virtuel. Seule l'autorisation de lecture est obligatoire.



Après la création du répertoire virtuel, il est recommandé d'arrêter et de redémarrer le serveur Web.

d. Cr éation de la page Web d'accueil

Pendant cette ultime étape, vous pouvez laisser s'exprimer vos talents artistiques pour concevoir la page

d'accueil. Les seules contraintes sont d'avoir sur cette page un lien vers le fichier jnlp et de respecter le nom de la page d'accueil tel qu'il est mentionné dans le fichier jnlp. L'ajout du lien se fait avec la balise html suivante :

```
<a href="ardoise.jnlp">installation de l'application ardoise magique </a>
```