Java pas à pas

Introduction à la programmation et au langage Java

Robert Godin, Daniel Lemire

Seconde édition, juillet 2024

Préface

Cet ouvrage présente les concepts de base de la programmation et du langage Java. Le livre s'adresse à un auditoire très large, aussi bien un débutant qui désire apprendre la programmation pour le plaisir qu'à un étudiant qui entreprend une carrière d'informaticien. L'approche proposée introduit graduellement les concepts de base de la programmation et leur incarnation dans le langage Java à l'aide d'une série d'exemples et d'exercices. Les exercices apparaissent avec leurs solutions mais il est important de prendre le temps de les faire pour en tirer les pleins bénéfices.

Un aspect qui motive l'emploi de Java comme premier langage est sa popularité. Sa conception par James Gosling, alors qu'il travaillait pour la compagnie Sun Microsystems, a été motivée par la possibilité d'écrire un code qui soit le moins dépendant possible de la plate-forme d'exploitation visée (*Write Once Run Anywherre* - WORA). Java est devenu un langage de développement très répandu pour une grande variété d'applications dans toutes sortes de contextes d'exploitation. La plupart des applications roulant sur les téléphones intelligents Android sont écrites en Java ou dans une variante du Java (comme Kotlin). Les systèmes de mégadonnées comme Apache Spark ou Elasticsearch sont souvent écrits en Java.

Afin de promouvoir l'aspect ludique de la programmation, le développement d'applications graphiques en deux dimensions et d'un jeu interactif simple est proposé pour illustrer des concepts de programmation importants, tels que la programmation objet et la modularisation du code, dans un contexte non trivial.

La programmation est une activité très gratifiante parce que les efforts fournis sont récompensés par des créations concrètes et utiles. Un grand sentiment d'accomplissement peut être éprouvé au fur et à mesure du développement d'un logiciel. Le plaisir est d'autant plus grand que le projet est d'envergure. Cependant, l'initiation à la programmation comporte aussi un grand potentiel de frustrations, surtout dans les premiers temps. En particulier, un certain nombre de détails au sujet des systèmes de développements de programmes doivent être maîtrisés dans les premières phases d'apprentissage afin d'atteindre les activités plus intéressantes qui sont au cœur de la programmation. Nous vous encourageons à faire preuve de

persévérance et de patience pour surmonter cette première difficulté car les efforts seront grandement récompensés par la suite.

Il faut toujours garder à l'esprit que la seule lecture d'un livre sur la programmation ne suffit jamais pour apprendre à programmer. Il vous faut programmer pour apprendre. Vous devrez faire les exercises, tester les exemples, les modifier, etc. Il faut être curieux pour apprendre.

1. Concepts de base

" L'art de douter est le meilleur secret pour apprendre ", Marcel Prévost

Un ordinateur doit être programmé pour accomplir une tâche.

***Programme*, *programmation*, *langage de programmation*, *logiciel* (*software*), *matériel* (*hardware*)**

Un *programme* est un ensemble organisé d'instructions que l'ordinateur exécute afin d'accomplir une tâche. La *programmation* est le processus de production d'un programme. Un programme est exprimé à l'aide d'un *langage de programmation*. Il existe une grande variété de langages de programmation qui ont été développés tels que Java, Python, C, C^{++}, C#, JavaScript, Swift, PHP, Scala, Go, R, etc.

Le terme logiciel (*software*) désigne l'ensemble des programmes, par opposition au matériel (*hardware*) de l'ordinateur qui correspond à l'ensemble des composantes physiques.

Avant d'aborder la programmation et Java, il est utile d'introduire certaines notions de base et situer la programmation dans son contexte.

1.1 Composantes matérielles d'un ordinateur (*hardware*)

La description d'un ordinateur faite dans cette section est une introduction à un sujet complexe. Parfois, certains aspects sont simplifiés ou omis afin de s'en tenir à l'essentiel pour la compréhension des concepts de base de la programmation. Un ordinateur est constitué de plusieurs *composantes matérielles* qui collaborent entre elles afin de produire un traitement. L'*architecture matérielle* d'un ordinateur est la description de ses composantes matérielles et de leurs interrelations. Les composantes matérielles d'un ordinateur typique sont illustrées à la Figure 1 : processeur central, mémoire centrale, bus et unités périphériques.

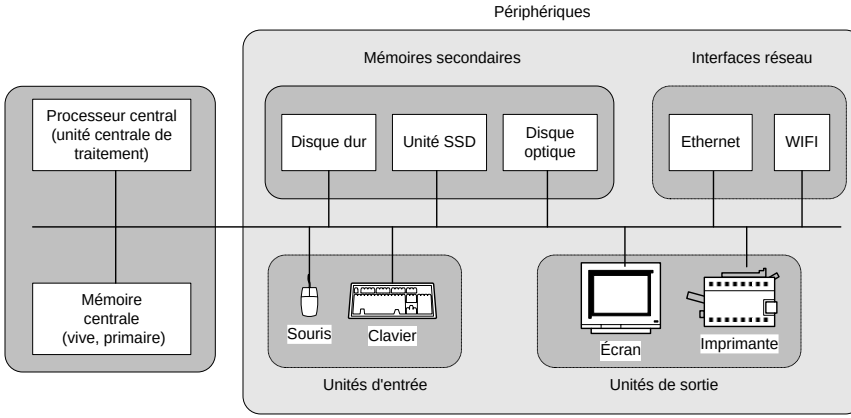


Figure 1. Composantes matérielles (*hardware*) d'un ordinateur typique.

Bus

Le bus permet la communication entre les composantes de l'ordinateur. Une composante branchée au bus peut envoyer des informations à une autre composante ou recevoir des informations d'une autre composante branchée au même bus.

Pour permettre les échanges ordonnés entre les différentes composantes, il doit y avoir un mécanisme qui empêche les conflits entre plusieurs composantes qui veulent utiliser le bus en même temps. Les composantes doivent respecter des règles précises pour établir la communication. L'ensemble des règles est appelé le *protocole de communication du bus*.

1.1.1 Processeur et mémoire

Le cœur d'un ordinateur est composé d'un *processeur central* et d'une *mémoire centrale*. Avant qu'un programme ne puisse être exécuté, celui-ci doit être placé dans la mémoire centrale de l'ordinateur.

Mémoire centrale* (*main memory*, *primary storage*), *principale*, *vive*, *primaire* ou *volatile

La mémoire centrale contient temporairement les *instructions* et *données* d'un programme en cours de traitement. L'ordinateur exécute des instructions placées en mémoire centrale. Ces instructions manipulent des données qui

doivent aussi résider en mémoire centrale. Un programme est habituellement chargé en mémoire centrale à partir d'une unité périphérique (souvent une mémoire secondaire tel que le disque dur) avant d'être exécuté.

Adresse-mémoire

La mémoire centrale est constituée d'une séquence de *cases* (*cellules*, *mots*) de taille fixe. Une case de la mémoire centrale est identifiée par une *adresse* (*adresse-mémoire*). Dans le cas le plus simple, l'adresse est un entier dans un intervalle de 0 à *n*-1[1], où *n* est la taille de la mémoire centrale. La taille d'une case peut varier selon le processeur utilisé.

Les cases de la mémoire centrale peuvent contenir des instructions ou des données. Le contenu des cases peut être modifié par les instructions des programmes.

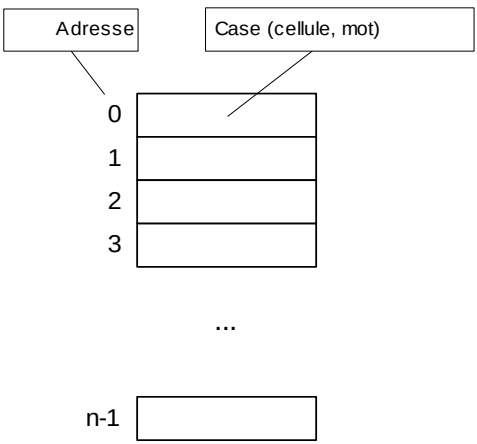


Figure 2. Mémoire centrale.

La mémoire centrale est caractérisée par sa rapidité d'accès d'une part et sa volatilité (non-permanence) d'autre part. La vitesse est importante pour que le processeur puisse accéder rapidement aux instructions et aux données en mémoire centrale lors de l'exécution d'un programme. On désigne aussi la mémoire centrale par le terme *Random Access Memory* (RAM) à cause de cette capacité à pouvoir accéder rapidement à n'importe quelle case à tout

[1] Par exemple, si la taille de la mémoire est *n*=16, les cases seront numérotées de 0 à 15. En réalité, le schéma d'adressage peut être plus compliqué...

moment. Sauf pour une petite partie appelée le ROM (*Read Only Memory*), le contenu de la mémoire centrale n'est pas permanent. Il est perdu lorsque le courant électrique qui alimente l'ordinateur est interrompu. Le ROM et les mémoires secondaires permanentes (tel que le disque) peuvent être utilisées pour conserver l'information en permanence au-delà des interruptions de courant. Il faut comprendre que les interruptions de courant ne sont pas toujours volontaires et peuvent provenir, par exemple, d'une panne d'électricité. Il est donc important de placer, en mémoire secondaire ou en ROM, les éléments qui doivent être conservés de manière *persistante*, i.e. survivre aux programmes ou aux anomalies de fonctionnement.

En pratique, le programmeur n'accède pas directement aux adresses en mémoire correspondant à du stockage physique. Le système d'exploitation offre au logiciel un accès à une mémoire virtuelle. L'adresse virtuelle est convertie par le système d'exploitation, au besoin, en adresse physique. Le programme reçoit sa mémoire en bloc appelée pages. Les pages peuvent avoir différentes tailles selon le système d'exploitation et l'architecture matérielle, mais occupent généralement au moins 4 kilo-octets. Par ailleurs, au sein d'un langage de programmation comme Java, mêmes les adresses virtuelles ne sont pas accessibles au programmeur.

***Processeur central* (*Central Processing Unit* - CPU), ou *unité centrale de traitement* (UCT)**

Le *processeur central* est la composante qui coordonne l'exécution d'un programme. Il effectue inlassablement le traitement suivant :

Chercher la prochaine instruction en mémoire centrale[2]
Exécuter l'instruction
Chercher la prochaine instruction en mémoire centrale
Exécuter l'instruction
Etc.

Chacune des instructions d'un programme produit un traitement relativement simple. Par exemple, une instruction peut additionner le contenu de deux cases de la mémoire centrale dont les adresses sont x et y, et placer le résultat dans une troisième case dont l'adresse est z.

[2] La réalité est un peu plus complexe. L'accès à la mémoire centrale peut être accéléré par l'intermédiaire d'une antémémoire (*cache memory*).

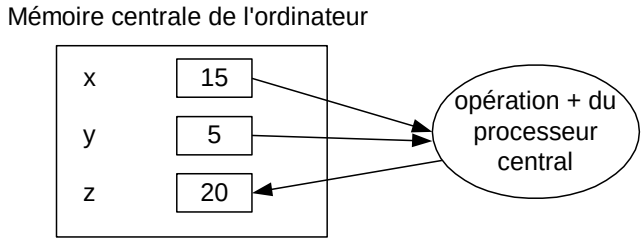


Figure 3. Instruction effectuée par le processeur central.

Le processeur central est lui-même typiquement constitué de deux composantes : l'Unité de Contrôle (UC) et l'Unité Arithmétique et Logique (UAL). L'UC est responsable de déterminer la séquence d'exécution des instructions. L'UAL exécute les instructions telles que l'addition illustrée à la Figure 3.

Les processeurs effectuent généralement leurs opérations sur des valeurs qui sont stockées au sein de registres : les registres sont des unités de mémoire particulières faisant généralement 64 bits et permettant de représenter différentes valeurs (nombres, adresses, etc.). Les processeurs n'ont que quelques dizaines de registres au plus: les registres doivent constamment être déchargés en mémoire ou rechargés à partir de la mémoire.

La puissance d'un ordinateur vient de sa capacité à exécuter un très grand nombre d'opérations simples à une vitesse extrême. Un processeur typique peut exécuter des milliards d'instructions par seconde.

Les processeurs exécutent leurs instructions à une cadence fixe, souvent précisée en gigahertz (GHz). À chaque seconde, le processeur bénéficie d'un certain nombre de cycles. Les instructions démarrent et se terminent toujours lors d'un cycle. Toute chose étant égale, un processeur ayant une cadence plus rapide (plus de cycles par seconde) sera plus rapide ; cependant il utilisera aussi davantage d'énergie. Les processeurs ajustent donc parfois leur vitesse en fonction de la charge de calcul.

La plupart de nos processeurs sont superscalaires : ils peuvent exécuter plusieurs instructions par cycle. Ainsi un processeur pourrait exécuter plusieurs additions en simultanée. Certains processeurs peuvent faire 4 ou 6 additions par cycle. Par ailleurs, la plupart des processeurs bénéficient

d'instructions spécialisées appelées SIMD (single instruction multiple data) qui fonctionnent sur des registres plus volumineux (faisant 128 bits, 256 bits ou plus) capables de représenter plusieurs valeurs à la fois (par exemple, 4 entiers). La plupart du temps, le programmeur n'a pas à se soucier de la manière dont sont exécutées les instructions : si on fait exception de la performance, il importe peu de savoir combien d'opérations sont traités par cycle.

Un aspect important du fonctionnement de l'ordinateur est la manière de déterminer la *prochaine instruction* à exécuter. Il y a trois mécanismes fondamentaux à cet effet.

1. *Séquence.* L'adresse en mémoire centrale de la *prochaine instruction* est normalement celle qui suit l'adresse de l'instruction précédente. Donc, par défaut, les instructions sont exécutées en séquence. Cependant, si c'était toujours le cas, l'unité centrale de traitement exécuterait les instructions jusqu'à ce qu'elle aboutisse à un cul de sac, soit la dernière adresse de la mémoire centrale et ne pourrait continuer !

2. *La boucle* Des instructions peuvent spécifier l'adresse de la prochaine instruction à exécuter. Par exemple, une instruction peut provoquer le saut à une adresse précédente ce qui permet de répéter un ensemble d'instructions en revenant constamment au début de la séquence.

3. *Décision* (*sélection, embranchement* ou *choix*). Certaines instructions peuvent choisir l'adresse de la prochaine instruction en fonction d'une condition. Par exemple, si le contenu de la case d'adresse *x* est 0, sauter à l'adresse *y* sinon continuer normalement en séquence. C'est ce genre d'instruction qui permet à l'ordinateur de « prendre des décisions » et de modifier son comportement au besoin.

Ces trois manières d'organiser l'exécution des instructions, la séquence, la boucle et la décision, sont des mécanismes de base de la plupart des langages de programmation.

En pratique, la plupart des CPU adoptent une architecture multi-cœur où chaque cœur est un processeur capable d'exécuter ses propres instructions. Il peut y avoir 2, 4, 6 ou même 64 cœurs (ou processeurs) dans un même CPU. Ces processeurs coexistent dans la même unité et ils ont accès aux

mêmes ressources matérielles. L'unité centrale doit coordonner les processeurs.

Les processeurs sont souvent beaucoup plus rapides que la mémoire et l'accès à la mémoire souffre d'un temps d'accès (latence). En général, plusieurs dizaines de nanosecondes sont nécessaires pour qu'un processeur puisse avoir accès à une valeur en mémoire. Un tel délai peut représenter des dizaines de cycles pendant lesquels le processeur ne pourrait rien faire. Pour améliorer la performance, les processeurs ont donc leur propre mémoire appelée mémoire tampon. Cette mémoire sert à la fois à stocker des données et des instructions. Il y a plusieurs couches de mémoire tampon allant de la plus petite et la plus rapide (L1) à la plus lente et la plus volumineuse (L3 sur les processeurs x64). Le temps d'accès à la mémoire stockée en L1 n'est que de quelques cycles, mais ne permet que de stocker quelques kilo-octets. Il serait complexe pour le processeur de gérer la mémoire avec une granularité très fine (par exemple, octet par octet). Ainsi la mémoire tampon gère la mémoire en blocs de mémoire appelée lignes. Les lignes font souvent entre 64 octets et 128 octets. Le processeur ne peut donc pas demander la valeur d'un octet spécifique : il faut accéder à une ligne entière. Les lignes sont alignées avec les adresses de la mémoire virtuelle : les premiers 64 octets forment une ligne et ainsi de suite. Dans une architecture multi-cœur, les différents processeurs ont des mémoires tampons distinctes, mais aussi des mémoires tampons partagées. Cette mémoire tampon garde une copie des données récemment traitées. Il faut néanmoins souvent tout de même avoir accès à la mémoire centrale, mais le processeur passe généralement par la mémoire tampon. La plupart des processeurs peuvent faire plusieurs requêtes d'accès à la mémoire en même temps.

1.1.2 Unités périphériques

Les unités périphériques permettent d'échanger de l'information entre la mémoire centrale de l'ordinateur et le monde extérieur.

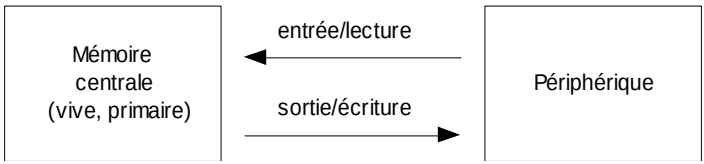


Figure 4. Opération d'entrée/lecture et de sortie/écriture

***Entrée, lecture* (*input*)**

Du point de vue de la mémoire centrale, une opération de transfert d'une information d'une unité périphérique vers la mémoire centrale est appelée une *opération d'entrée* ou de *lecture*.

***Sortie, écriture* (*output*)**

Un transfert inverse de la mémoire centrale vers une unité périphérique est une *opération de sortie* ou d'*écriture*. Des instructions d'entrées/sorties sont prévues pour déclencher les opérations d'entrée/sortie.

Périphérique d'entrée

Les *périphériques d'entrée* permettent de transmettre de l'information du monde extérieur à la mémoire centrale. Par analogie avec l'humain, les périphériques d'entrée sont les sens de l'ordinateur. En particulier, certains périphériques permettent aux humains de communiquer avec l'ordinateur. La souris, l'écran tactile, la caméra, le microphone et le clavier sont des unités d'entrée bien connus. La *manette de jeu* et le *bâton joyeux* (*joystick*) sont employés pour les jeux. Au-delà de ces périphériques d'entrée usuels, toutes sortes de capteurs existent pour saisir des données de diverses natures.

Périphérique de sortie

Les *périphériques de sortie* permettent à l'ordinateur de transmettre de l'information au monde extérieur et en particulier à l'humain. L'*écran* et l'*imprimante* sont des périphériques de sortie typiques. Les *haut-parleurs* ou les *écouteurs* sont aussi fréquemment utilisés.

Divers périphériques d'entrée/sorties plus adaptés à des applications particulières sont de plus en plus répandus. Les *interfaces réseau* (*modem*, *carte réseau*, ...) et les *mémoires secondaires* (*disque dur*, *mémoire* SSD, *disque optique*, ...)

sont aussi des périphériques d'entrée/sortie. Plusieurs de ces périphériques permettent à la fois les entrées et les sorties. Par exemple, il est possible de lire et d'écrire des informations sur le disque dur. Il est possible d'envoyer et de recevoir des informations par un réseau. Cependant, il est impossible d'écrire sur certains disques optiques.

Mémoire secondaire* (*secondary storage*), *de masse*, *auxiliaire*, *permanente*, *externe*, *stable*, *non volatile* ou *persistante

Une *mémoire secondaire* est une mémoire habituellement plus lente que la mémoire centrale mais qui a la caractéristique d'être permanente. Son contenu ne disparaît pas lorsque le courant électrique est interrompu. Les informations (données et programmes) qui doivent être conservées en permanence sont donc placées en mémoire secondaire. Ce type de mémoire est aussi appelé mémoire *de masse*, *auxiliaire*, *permanente, externe*, *stable*, *non volatile* ou *persistante*.

Il est possible de transférer des informations (données ou programmes) entre une mémoire secondaire et la mémoire centrale par des instructions d'entrées/sorties prévues à cet effet. Historiquement, la mémoire secondaire la plus répandue était l'*unité de disque magnétique* (*disque dur*, *disque rigide* ou tout simplement le *disque*). La *mémoire* SSD (*Solid State Drive*), ou *disque* SSD est maintenant plus populaire à cause de sa rapidité en lecture par rapport au disque. Les autres types de mémoires secondaires moins rapides ont d'autres caractéristiques qui les rendent utiles (coût, capacité de stockage, etc.).

Dans certains cas, il est possible d'utiliser une mémoire secondaire distante qui n'est pas à proximité physiquement du processeur. On accède à cette mémoire par l'entremise d'un réseau avec ou sans fil.

Interface réseau

Une interface réseau permet à l'ordinateur de communiquer avec d'autres ordinateurs par l'intermédiaire d'un réseau d'ordinateurs. Il y a différents types de réseaux d'ordinateurs et la manière de brancher un ordinateur à un réseau varie d'un type de réseau à un autre.

Réseau local* (*Local Area Network* - LAN), *domestique*, *d'entreprise*, *Internet

Un réseau dit *local* permet la communication entre un ensemble limité d'ordinateurs situés à proximité[3] les uns des autres. Un *réseau local domestique* permet la communication entre ordinateurs d'une résidence. Un *réseau local d'entreprise* permet la communication entre les ordinateurs à l'intérieur d'une entreprise. Le réseau *Internet* est un réseau planétaire qui permet la communication avec des centaines de millions d'ordinateurs répartis autour de la planète. Pour être plus précis, le réseau Internet permet la communication entre (*inter*) les réseaux locaux (*net*). C'est donc un réseau de réseaux ! Ainsi, un ordinateur branché à un réseau local peut accéder à Internet lorsque le réseau local est lui-même branché à Internet.

Deux interfaces réseau populaires sont l'interface avec fil *Ethernet* ou l'interface sans fils (WIFI) à un réseau local qui est lui-même relié au réseau Internet. Les appareils mobiles peuvent passer par le réseau cellulaire pour l'accès à Internet. Le branchement entre les périphériques et le bus suit des conventions basées des normes bien établies tel que ISA, PCI, PCMCIA, SCSI, etc. La pièce maîtresse de l'ordinateur sur laquelle sont installées les différentes composantes de l'ordinateur est la *carte mère* (*mother board*).

1.2 Le logiciel

Un ordinateur fonctionne en exécutant des programmes. On utilise souvent le terme *logiciel* (*software*) pour désigner les programmes. Lorsqu'on démarre un ordinateur[4], il y a un premier programme qui est automatiquement exécuté, appelé le *programme de démarrage* (*boot program*). Ce premier programme est toujours dans la mémoire centrale à une adresse fixe, connue à l'avance. À cet effet, il y a une petite partie de la mémoire centrale qui est permanente, le ROM (*Read Only Memory*), et qui contient le programme de démarrage. Le programme de démarrage a pour rôle essentiel de placer en

[3] La distance possible entre les ordinateurs d'un réseau local varie en fonction du matériel utilisé.

[4] On allume l'ordinateur en enfonçant le bouton ON qui est parfois bien caché pour que les non-initiés éprouvent un sentiment d'humiliation la première fois qu'ils essaient de le faire fonctionner. Curieusement, il faut parfois enfoncer le bouton ON pour éteindre certains ordinateurs…

mémoire centrale un plus gros programme appelé le *système d'exploitation*[5], à partir d'une mémoire secondaire, habituellement le disque dur[6].

***Système d'exploitation* (*operating system* - OS)**

Le système d'exploitation est un programme dont le rôle est de faciliter l'utilisation de l'ordinateur et en particulier des périphériques. Windows, Unix et macOS sont des exemples de systèmes d'exploitation. Sans système d'exploitation, un ordinateur est aussi utile qu'une aiguille dans une botte de foin.

Le système d'exploitation permet de faciliter l'utilisation de l'ordinateur et d'organiser l'exécution d'autres programmes appelés *programmes d'application* ou plus simplement *applications*. Après que le système d'exploitation est chargé en mémoire centrale, celui-ci se met en état d'attente d'une commande de l'utilisateur. À ce moment, l'utilisateur va souvent commander l'exécution d'un programme d'application.

***Interface système* (*shell*), *interface en ligne de commande* (*Command Line Interface*- CLI), *interface à l'utilisateur graphique* (Graphical User Interface - GUI)**

Les systèmes d'exploitation offrent habituellement deux types d'interface permettant d'invoquer leurs fonctions. Les interface en ligne de commande (*Command Line Interface* – CLI) ou les *interfaces à l'utilisateur graphiques* conviviales (*Graphical User Interface* - GUI) basées sur l'utilisation des fenêtres, de menus, de boutons, etc. Dans le cas de Windows, l'interface en ligne de commande est aussi appelée *fenêtre de commande Windows* (*Command Prompt*) et Windows est l'interface graphique. Dans les systèmes d'exploitation UNIX, par analogie avec une noix, il est d'usage d'employer le terme *shell* (coquille) pour désigner une interface système qui permet de faire exécuter des commandes du *noyau* (*kernel*) du système d'exploitation.

[5] Pour être plus précis, le programme de démarrage inclut déjà certaines parties du système d'exploitation nécessaires pour accéder aux unités périphériques. Dans les ordinateurs PC compatibles, cette portion du système d'exploitation est appelée le *BIOS* (*Basic Input Output System*).

[6] Il est aussi possible de charger le système d'exploitation à partir d'une autre mémoire secondaire (clé USB, cédérom, etc.) mais ceci est habituellement effectué dans des circonstances spéciales, par exemple, lorsqu'un problème survient avec le disque dur.

***Programme d'application* (ou simplement *application*)**

Les programmes d'applications accomplissent des tâches utiles. Par exemple, parmi les applications d'usage courant, il y a les *programmes de traitement de textes* (e.g. Bloc-notes, Wordpad de Windows, Word de Microsoft Office, Emacs de UNIX), les *chiffriers électroniques* (e.g. Excel de Microsoft Office), les *fureteurs WEB* (e.g. Google Chrome, Safari, Microsoft Edge), les *systèmes de courriel* (e.g. Outlook de Microsoft, Google GMail), les *réseaux sociaux* (e.g. Facebook, Twitter, Instagram).

Cet ouvrage traite du développement de programmes d'application en Java. Concrètement, avant qu'il ne soit chargé en mémoire centrale pour être exécuté, un programme d'application se trouve habituellement en mémoire secondaire, par exemple, sur disque dur. Un des rôles importants du système d'exploitation est de permettre d'organiser les programmes et les données placées en mémoire secondaire afin de pouvoir les retrouver.

***Fichier* (*file*), *système de gestion de fichiers* (*file system*)**

La partie du système d'exploitation qui s'occupe de l'organisation des données et des programmes en mémoire secondaire est le *système de gestion de fichiers* (*file system*). Un *fichier* peut contenir un ou une partie d'un programme ou encore des données pour un programme.

On utilise souvent le terme *document* pour désigner un fichier de données. En effet, un fichier peut par exemple contenir un texte produit avec un programme de traitement de texte. Le fichier est alors un document contenant des données du point de vue de l'application de traitement de texte. Le système de gestion de fichier permet de regrouper des fichiers dans un *dossier* (aussi appelé *répertoire*). Un dossier peut lui-même regrouper d'autres dossiers, ainsi de suite, résultant en une *hiérarchie de dossiers*.

Pour identifier un dossier ou un fichier, on peut employer son *chemin* (*path*). Le chemin indique la séquence des dossiers qui mènent à l'élément visé.

Sous Windows, le chemin commence tout d'abord par le nom de *volume* dans lequel si situe le fichier ou dossier que l'on recherche, puis vient ensuite l'ordre des dossiers à parcourir (les noms des dossiers sont séparés par des «\»). Le nom de volume identifie une unité périphérique d'entrée/sortie telle

qu'un disque dur, un disque SSD, etc. Ainsi, dans l'exemple de chemin suivant, il y a un disque dur nommé C. Pour arriver au fichier *HelloWorld.java*, il faut donc partir du disque C, aller dans le dossier *Users*, puis ensuite, dans *Robert*, et ensuite dans *Documents* pour enfin arriver au fichier *HelloWorld.java*. Ce fichier contient un programme en Java comme nous le verrons par la suite.

C:\Users\Robert\Documents\HelloWorld.java

Il est à noter qu'après le nom de volume, il faut mettre un « : » avant le « \ ».

D'autres systèmes d'exploitation comme Linux, FreeBSD ou macOS utilisent des chemins qui débutant toujours par la racine («/») et omettant le nom du volume.

/users/lemire/Documents/HelloWord.java

Les systèmes de gestion de fichier modernes permettent le stockage dans des serveurs distants en donnant l'impression d'une unité de stockage locale. De plus en plus, le stockage est effectué dans le *nuage informatique* (*infonuagique*, *cloud computing*). Les grandes entreprises offrent des services de stockage dans le nuage qui incorporent des mécanismes de sécurité, de fiabilité et de partage plus élaborés que le stockage sur des unités locales.

Parallélisme simulé : pseudo-parallélisme

Les ordinateurs permettent aussi de faire exécuter plusieurs programmes en même temps. Mais comment ceci est-il possible même quand il n'y a qu'un seul processeur ! En réalité, dans un système monoprocesseur (à un seul processeur), il n'y a effectivement qu'un seul programme à la fois qui est exécuté. Le système d'exploitation fournit l'illusion d'une exécution simultanée de plusieurs programmes en allouant à chacun des programmes à tour de rôle une petite partie du temps du processeur central. Mais ceci se fait tellement rapidement que l'utilisateur ne peut le percevoir même s'il est un « petit vite ». Le terme *pseudo-parallélisme* désigne cette illusion. Au-delà du pseudo-parallélisme, les ordinateurs modernes sont souvent pourvus de plusieurs processeurs qui permettent donc l'exécution en parallélisme réel de plusieurs traitements.

Les unités périphériques sont aussi souvent pourvues de processeurs simples qui peuvent exécuter des opérations d'entrée/sortie en parallèle avec le traitement du processeur central. Pour coordonner les différents traitements entre tous ces processeurs, des mécanismes d'interruption sont prévus afin qu'un processeur puisse signaler à un autre processeur qu'une tâche est terminée, et que ce dernier puisse réagir à cet évènement. Ceci entraîne une interruption du cours normal d'exécution de la boucle de traitement de base. La synchronisation et coordination des différentes tâches parallèles d'un ordinateur peut devenir très complexe et c'est le rôle du système d'exploitation de fournir des services à cet effet.

1.2.1 Le binaire, le langage machine et la compilation

Les instructions qui composent un programme en mémoire centrale sont exprimées sous forme d'un code appelé le *langage machine*. Ce langage varie en fonction du type de processeur visé. Une instruction en langage machine est composée d'une suite de symboles binaires. Un symbole binaire est soit un « 0 » ou un « 1 ». On désigne par *langage binaire*, un langage qui se limite à l'utilisation de deux symboles. Chacun des symboles d'une séquence de symboles binaires (0 ou 1) est appelé un *bit*. Une séquence de 8 bits est appelée un *octet*[7] (*byte*). Chacune des cases de la mémoire peut contenir un nombre fixe de bits, habituellement, 8 bits (1 octet). La taille d'une instruction est un multiple de la taille d'une case mémoire. Il peut même y avoir des tailles d'instruction variables pour un même processeur.[8]

Les données manipulées par un programme sont aussi codées en mémoire sous forme binaire. Par exemple, un nombre entier est souvent représenté par une séquence de 32 bits (4 octets). Le nombre entier est codé selon un système de numération binaire (base 2). Par exemple, l'entier 25 en décimal est représenté sur huit bits (un octet) par :

$$00011001_2 = 0 \times 2^7 + 0 \times 2^6 + 0 \times 2^5$$

$$+1 \times 2^4 + 1 \times 2^3$$

$$+0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

[7] En anglais, le terme octet existe aussi, mais on lui préfère le terme *byte*. Généralement, un *byte* est constitué de 8 bits.

[8] C'est le cas pour les processeurs x86/x64 commercialisés par Intel et AMD.

$$= 25_{10}$$

Chacun des bits correspond à un exposant en base 2.

Les mémoires modernes ont des tailles impressionnantes. Les unités de mesure suivantes sont employées pour les tailles des mémoires et données. Elles sont basées sur le système international qui repose sur la base 10.

Nom	**Symbole**	**Nombre d'octets**
Kilo-octet	Ko	10^3
mégaoctet	Mo	10^6
gigaoctet	Go	10^9
téraoctet	To	10^{12}
pétaoctet	Po	10^{15}
exaoctet	Eo	10^{18}
zettaoctet	Zo	10^{21}
yottaoctet	Yo	10^{24}

En anglais, on emploi les acronymes KB, MB, GB, TB, PB où B correspond à *Byte*.[9] À cause de l'emploi des nombres binaires en informatique, on a traditionnellement employé le préfixe *kilo* pour désigner 2^{10}, *méga* pour 2^{20} et *giga* pour 2^{30}. Depuis 1998, la norme CEI 60027-2 propose plutôt les préfixes du tableau suivant pour les nombres en base 2. Cependant, ces conventions sont plus ou moins respectées en pratique.

Nom	**Symbole**	**Nombre d'octets**
kibioctet	Kio	$2^{10} = 1\ 024$
mébioctet	Mio	$2^{20} = 1\ 048\ 576$
gibioctet	Gio	$2^{30} = 1\ 073\ 741\ 824$
tébioctet	Tio	$2^{40} = 1\ 099\ 511\ 627\ 776$
pébioctet	Pio	$2^{50} = 1\ 125\ 899\ 906\ 842\ 624$
exbioctet	Ei	$2^{60} = 1\ 152\ 921\ 504\ 606\ 846\ 976$
zébioctet	Zi	$2^{70} = 1\ 180\ 591\ 620\ 717\ 411\ 303\ 424$

[9] En anglais, le B majuscule désigne un *byte*. Tant en français qu'en anglais, le b minuscule désigne le bit. Il ne faut donc pas confondre KB et Kb.

yobioctet	Yi	2^{80} = 1 208 925 819 614 629 174 706 176

Une chaîne de caractère est typiquement représentée en utilisant une séquence d'octets. Chacun des caractères est codé en suivant une norme de codage. Le code ASCII est une des premières normes employées pour représenter les caractères par des codes de la taille d'un octet. L'exemple suivant montre le codage pour une suite d'octets. La première ligne montre le code binaire. La deuxième ligne indique les caractères correspondants.

```
01100001 01100010 01100011 00001101 00001010 00110001 00110010
00001101 00001010

a        b        c        \r       \n       1        2        \r
\n
```

Sous Windows, la fin de ligne est représentée par la séquence des caractères spéciaux ASCII, *retour de chariot* ('\r') et *saut de ligne* ('\n')[10]. Le code ASCII ne permet pas de traiter les caractères de toutes les langues. La norme *Unicode* (www.unicode.org) est une norme plus générale qui permet d'encoder les caractères d'un grand nombre de langues. Sur Internet, on représente souvent Unicode en utilisant le code UTF-8. Le code ASCII est un cas particulier du code UTF-8. Le code UTF-8 permet de représenter une gamme plus riche de caractères (appelés parfois graphèmes) en utilisant des séquences de 2, 3 ou 4 octets. On distingue les octets ASCII des autres au sein d'un séquence UTF-8 par le fait que leur bit de poids fort est 0. Unicode permet aussi de combiner plusieurs graphèmes pour faire un même caractère. Le Java utilise le code UTF-16 qui représente chaque graphème en utilisant 2 ou 4 octets ; le code UTF-16 est incompatible avec ASCII. Le code UTF-8 est plus concis que le code UTF-16, sauf lors que le texte comporte beaucoup de caractères asiatiques. En pratique, il est souvent nécessaire de faire une conversion entre UTF-16 et UTF-8 lorsqu'on programme en Java, en particulier quand on doit communiquer avec le web.

Une image est représentée par un quadrillage de pixels (*picture element*). Pour une image en noir et blanc, chacun des pixels est représenté par un bit qui correspond à la couleur noir ou blanc (0 ou 1). Pour une image en couleur, le système RVB (RGB) encode chacun des pixels par trois entiers entre 0 et

[10] La manière de représenter les fins de ligne peut différer en fonction de la plate-forme. Sous macOS et Linux, par exemple, on emploie uniquement le *saut de ligne* ('\n').

255 qui correspondent aux trois couleurs, rouge (*Red*), vert (*Green*) et bleu (*Bleu*).

Dans le cas des premiers ordinateurs, il fallait les programmer directement en langage machine, ce qui était très fastidieux et peu productif. Aujourd'hui, les humains programment à l'aide de langages de programmation dits évolués, tel que Java. Les langages évolués cachent le codage binaire employé pour les données. Les données sont représentées dans une forme interprétable par l'humain. Par exemple, les nombres entiers sont représentés en décimal dans les programmes Java.

Cependant, comme l'ordinateur ne comprend pas directement le Java, il faut un moyen intermédiaire qui permet d'exécuter le programme Java sur une machine. Deux approches sont possibles, la compilation et l'interprétation. Le processus de compilation et d'exécution d'un programme Java est illustré à la Figure 5.

Compilateur

Un *compilateur* est un programme qui traduit un programme, écrit en un langage évolué, en un programme en langage machine. Le programme en langage évolué est appelé le *programme source* ou encore *code source*. Et le programme en *langage machine* résultant de la traduction est appelé le *programme objet* ou encore le *code objet*. Les langages C, C++, Rust, Swift et Fortran s'exécutent normalement par l'entremise d'un compilateur.[11]

Code-octet Java, machine virtuelle Java[12]

Le cas de Java est un peu plus compliqué parce que le compilateur Java traduit le programme source en un programme objet intermédiaire, correspondant au langage machine de la *machine virtuelle Java* (*Java Virtual Machine* -JVM). Le langage machine de la machine virtuelle Java est appelé *code-octet* (*byte-code*) *Java* (ou simplement *code-octet*). La machine virtuelle Java est une machine théorique qui n'existe pas vraiment, du moins pour l'instant. Il manque donc quelque chose pour faire exécuter le programme objet en *code-octet* sur une machine réelle ! Cependant, comme le langage code-octet est

[11] En pratique, plusieurs compilateurs vont utiliser des langages intermédiaires au lieu de compiler directement en langage machine.
[12] Le modèle en code-octet de Java n'est pas unique. La plateforme .NET de Microsoft et son langage C# sont similaires.

proche du langage machine des machines réelles, le processus de traduction restant est assez facile à réaliser. Il y a trois manières d'exécuter les programmes en *code-octet Java* sur la machine réelle visée :

- *Interprète de code-octet*. Un petit programme spécial, appelé *interprète de code-octet*, simule une machine Java. Comme la machine n'existe pas réellement mais qu'elle est simulée, elle est dite *virtuelle*. En d'autres mots ce petit programme donne l'illusion d'une machine dont le langage est le *code-octet*. L'interprète de code-octet est un programme en langage machine de la machine réelle.

- *Compilateur de code-octet*. Un second niveau de compilation traduit les instructions code-octet en instructions du langage machine de la machine réelle. Un compilateur JIT (*Just-In-Time*) effectue la traduction au moment d'exécuter le code-octet.

- *Processeur Java*. La troisième approche consisterait à construire un processeur dont le langage est le *code-octet* (*code-octet*) Java.

L'avantage d'utiliser un langage intermédiaire, le code-octet, dans le processus de traduction est l'indépendance du code objet vis-à-vis la machine réelle utilisée. Le même programme objet en code-octet Java peut être exécuté sur n'importe quelle machine pourvu qu'une machine virtuelle Java soit disponible sur celle-ci. L'indépendance de la plate-forme d'exécution est une considération importante à l'origine de Java qui vise à être utilisé dans des contextes variés. D'autres langages de programmation comme Kotlin peuvent utiliser le même code-octet que Java.

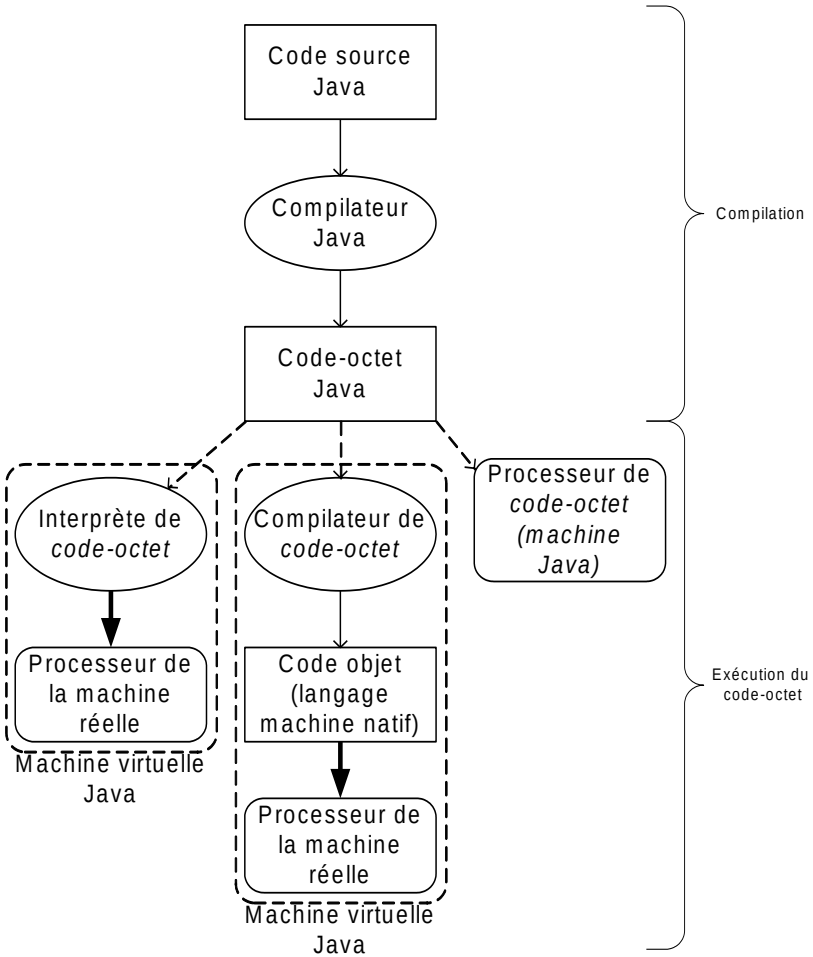


Figure 5. Compilation et exécution d'un programme Java.

Interprète

Un interprète est un programme qui exécute les instructions du langage une à la fois au fur et à mesure qu'elles sont lues. Cette approche permet de simplifier le processus d'exécution d'un programme en évitant l'étape intermédiaire de compilation. L'inconvénient est une exécution moins rapide que dans l'approche avec compilation qui permet d'optimiser l'exécution du code d'une manière plus globale. Certains langages de programmation sont plus appropriés à l'interprétation que la compilation, tel que Python,

Javascript, etc. Certains langages normalement interprétés (Python, JavaScript) bénéficient aussi d'un compilateur qui permet d'accélérer une partie du code. Dans la pratique, il y a donc souvent des modèles hybrides, utilisant à la fois des interprètes et des compilateurs.

1.2.2 Étapes de création et d'exécution d'un programme Java avec l'environnement JSE sous Windows

Pour illustrer les concepts précédents de manière concrète, cette section montre comment écrire, compiler et faire exécuter un petit programme Java très simple sous le système d'exploitation Windows. À cet effet, les outils de base offerts de Java (le *Java™ Platform Standard Edition (JSE).*) offerts gratuitement sur le site AdoptOpenJDK sont installés.[13] Le lien actuel est à l'adresse suivante :

https://adoptium.net/

Pour une démonstration d'installation, vous pouvez vous rendre sur le site YouTube suivant :

https://www.youtube.com/watch?v=Tk6u3Wm___s

Quand vous consultez une vidéo, prenez en compte que l'apparence du site, du système et d'autres éléments peuvent varier dans le temps. Une vidéo n'est pas un guide mais une illustration.

Il est aussi parfaitement possible de passer par le site d'Oracle :

http://www.oracle.com/technetwork/java/javase/downloads/index.html

Bien que le logiciel d'Oracle soit un excellent choix, plusieurs trouveront préférable de passer par AdoptOpenJDK.

Voici un exemple de scénario d'installation de la version de JSE pour le système d'exploitation Windows 10 et macOS, en utilisant le logiciel d'Oracle. Si vous employez un autre système d'exploitation ou une autre version du JSE, les détails des manipulations peuvent varier. Par exemple,

[13] Attention à ne pas installer le *Java Runtime Environment* (JRE). Le JRE permet d'exécuter des programmes Java, mais il ne permet pas de compiler de nouveaux programmes.

sous Linux on peut installer le JSE avec une commande telle que `apt-get install default-jdk` (ubuntu). Les instructions d'installation sont parfois un peu mystérieuses pour un débutant. Si c'est votre cas, il peut être nécessaire de recourir aux services d'un informaticien expérimenté.

- **Exemple d'installation de JSE avec Windows**

1. Télécharger le programme d'installation Oracle

2. Exécuter le programme d'installation. Une fenêtre indique que le programme d'installation s'exécute.

3. Répondre aux questions du programme d'installation. Vous pouvez accepter les options définies par défaut en cliquant sur le bouton *Next* à chacun des dialogues proposés.

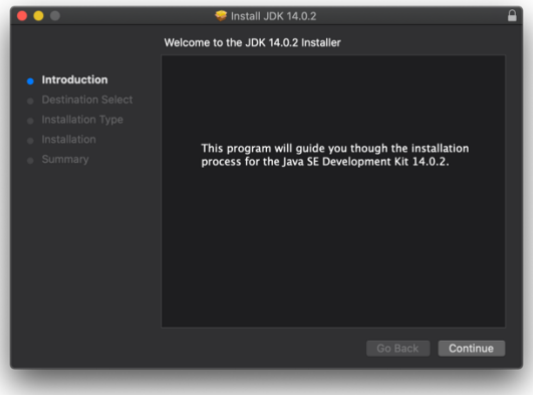
Le dialogue suivant permet de spécifier le dossier de base de l'installation du JSE. Dans cet exemple, la valeur de défaut a été acceptée. Il est habituellement préférable de ne pas la changer parce que les programmes qui utilisent le JSE peuvent la retrouver plus facilement.

La fenêtre de dialogue suivante indique que le processus s'est déroulé correctement.

- **Exemple d'installation de JSE avec macOS**

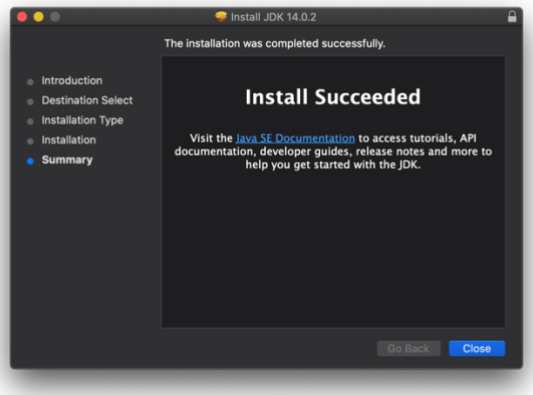
4. Télécharger le programme d'installation

5. Exécuter le programme d'installation. Une fenêtre indique que le programme d'installation s'exécute.



6. Répondre aux questions du programme d'installation. Vous pouvez accepter les options définies par défaut en cliquant sur le bouton *Continue* à chacun des dialogues proposés.

La fenêtre de dialogue suivante indique que le processus s'est déroulé correctement.



Parmi les outils inclus dans JSE, il y a un *compilateur Java* et un environnement d'exécution Java (*Java Run-time Environment* – JRE) incluant une *machine virtuelle Java* (*Java Virtual Machine* – JVM) qui peut exécuter le code-octet

produit par le compilateur Java. Le JRE contient aussi d'autres éléments nécessaires à l'exécution des programmes Java dont des *librairies de classes réutilisables* (cette notion sera expliquée par la suite). Cet ensemble d'outil est rudimentaire d'un point de vue convivialité et il existe des environnements de développement Java intégrés (*Integrated Development Environment* - IDE) plus élaborés tel que *Eclipse* (www.eclipse.org), *Netbeans* (www.netbeans.org), *JDeveloppper* d'Oracle (www.oracle.com), etc. Ces environnements facilitent la tâche du programmeur en simplifiant l'utilisation des outils du JSE. Ces IDE facilitent le processus d'édition, de compilation, de déploiement et d'exécution des programmes Java.

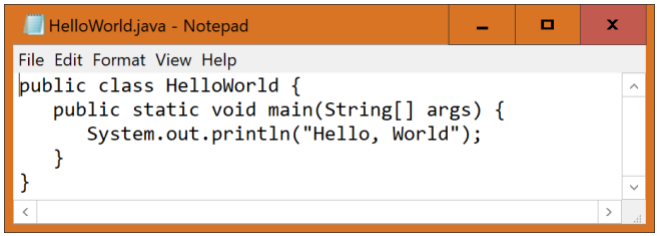
Le scénario qui suit est un exemple du processus de création et d'exécution d'un petit programme Java à l'aide des outils de base de JSE (le compilateur Java et la JVM Java) sous Windows. Il peut être utile d'invoquer ces outils directement au moins une fois afin de se familiariser avec le mécanisme. Une approche similaire est possible sous macOS et Linux.

- **Étape 1 : édition du texte du programme source Java avec Bloc-notes (Windows)**

Le texte d'un programme Java est produit avec un éditeur de texte. Voici le texte du programme de notre exemple :

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

Ce genre de programme est une tradition typique dans l'apprentissage d'un nouveau langage de programmation. Il ne fait qu'afficher la phrase « Hello, World » sur l'unité périphérique de sortie standard qui est habituellement une fenêtre à l'écran de l'ordinateur. Dans la figure suivante, l'éditeur de texte *Bloc-notes* de Windows est utilisé pour éditer le texte.



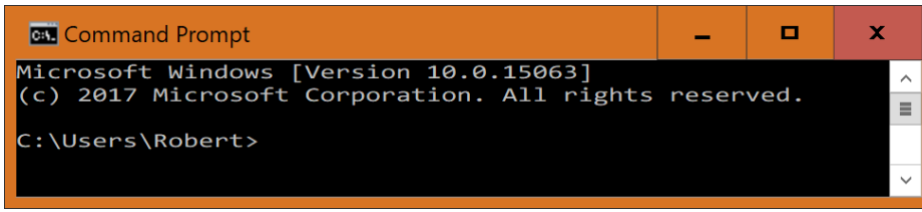
Le fichier est sauvegardé sous le nom *HelloWorld.java* dans le dossier dont le chemin est :

C:\Users\Robert\Documents

Par convention, un programme source Java a l'extension « *.java* ».

- **Invocation d'une fenêtre de commande Windows**

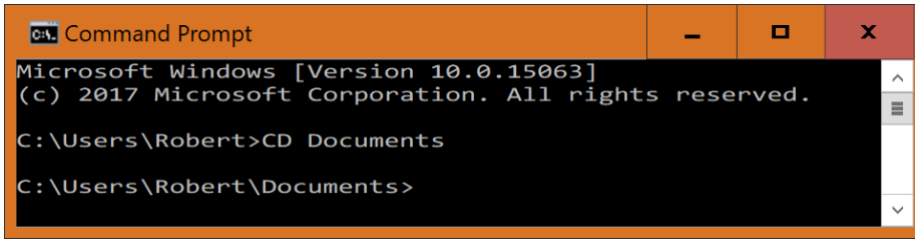
Pour compiler le programme source Java, il faut d'abord invoquer une fenêtre de commande Windows (*Command Prompt*).



Il y a différentes manières d'invoquer cette fenêtre, dont le menu des programmes Windows dans le coin inférieur gauche de l'écran.

- **Compilation et exécution du programme source Java en fenêtre de commande**

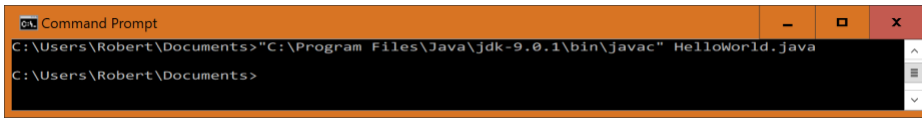
La commande « CD Documents » est donnée pour naviguer dans le répertoire du programme source Java :



Ensuite, le programme source est compilé en tapant le chemin qui mène au compilateur Java (programme *javac.exe*) suivi du nom du fichier[14] :

```
"C:\Program Files\Java\jdk-9.0.1\bin\javac" HelloWorld.java
```

Dans cet exemple, le chemin est encadré par des guillemets parce qu'il contient un espace. L'extension « .java » est optionnelle.



La commande *javac* invoque le compilateur Java (programme nommé *javac.exe*) qui produit le code-octet dans un fichier nommé *HelloWorld.class*. Par défaut, le programme objet compilé en code-octet possède le même préfixe que le programme source auquel le compilateur ajoute l'extension « .class ».

Erreur de compilation, erreur de syntaxe

Si le programme source est incorrect selon les règles de syntaxe du langage Java, des messages d'erreur sont affichés afin de faciliter le repérage des erreurs. Ce sont des erreurs parfois appelées erreurs de compilation ou erreurs de syntaxe.

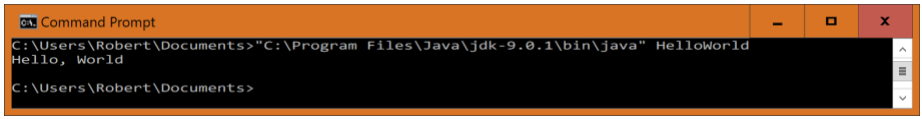
Attention !

Les messages d'erreur des compilateurs ne sont pas toujours faciles à déchiffrer … S'il y a des erreurs, il faut les corriger dans le programme source

[14] Il est important ici de connaître le chemin exact de la commande javac.exe. En cas de doute, vous pouvez naviguer sur votre disque dans les dossiers pour vérifier l'emplacement.

à l'aide de l'éditeur de texte et tenter de compiler le programme à nouveau. En pratique, il y a habituellement plusieurs cycles d'édition/compilation avant qu'un programme ne soit correct d'un point de vue de la compilation.

Le programme est exécuté avec la machine virtuelle Java en tapant le chemin de la machine virtuelle (programme *java.exe*) suivi du nom du fichier du code octet (*HelloWorld.class*) mais en omettant l'extension « .class ».



Le texte « Hello, World » est affiché ! L'exécution du programme Java comme telle est précédée de deux étapes :

Chargement (*load*). Avant d'être exécuté le code-octet du programme doit d'abord être chargé en mémoire principale à partir du fichier d'extension *.class* en mémoire secondaire.

Vérification. Ensuite, le code-octet est vérifié afin de déterminer s'il respecte certaines contraintes, en particulier, concernant la sécurité. Java inclut des mécanismes sophistiqués de sécurité. Ces mécanismes permettent d'empêcher qu'un programme Java ne produise des effets indésirables.

C'est seulement après la vérification que le programme est effectivement exécuté.

Erreur d'exécution

Dans un scénario typique de développement d'un programme, il y souvent des erreurs d'exécution qui peuvent conduire à l'interruption du programme ou à la production d'un résultat incorrect. Il faut alors corriger les erreurs dans le code source et recommencer. On distingue deux types d'erreurs. Il y a des erreurs à la compilation (par ex., lors de la production du code-objet), et il y a des erreurs à l'exécution. Les erreurs ayant lieu lors de l'exécution du programme sont parfois plus difficiles à régler et elles peuvent occasionner plus de désagréments.

- **Mettre à jour les variables d'environnement *Path* et *Classpath* de Windows**

Notez que si vous avez installé Java à partir du site d'AdoptOpenJDK, alors vos variables d'environnement ont été modifiée lors de l'installation par défault. Vous pouvez donc sauter cette étape.

Cette étape n'est pas nécessaire mais elle facilite l'utilisation des outils du JSE en évitant de devoir spécifier les chemins complets pour retrouver les outils et les programmes. La variable d'environnement *Path* identifie des chemins (de dossiers) que le système d'exploitation Windows parcoure afin de retrouver les programmes à exécuter. Dans la variable *Path*, le « ; » sépare les chemins les uns des autres. En ajoutant à la variable *Path* le chemin « C:\Program Files\Java\jdk-9.0.1\bin » qui contient les outils de JSE (le compilateur *javac.exe*, la machine virtuelle *java.exe*, etc.), il n'est pas nécessaire de spécifier ce chemin à chaque fois que l'on veut invoquer ces outils.

La variable d'environnement *Classpath* contient par défaut le chemin « . » qui représente le *dossier courant*. La notion de dossier courant sera développée par la suite. Le *Classpath* indique à la JVM (*Java Virtual Machine*) où trouver les programmes Java lorsque le chemin n'est pas spécifié. Si le dossier courant apparaît en premier, la JVM va toujours chercher dans le dossier courant en premier afin de retrouver les programmes Java. On peut ajouter d'autres chemins pour simplifier l'invocation des outils.

Une autre possibilité consiste à spécifier les options -sourcepath ou -classpath en invoquant les outils Java. Référez-vous à la documentation du JSE pour plus de détails. Ceci illustre la complexité de l'utilisation de ces outils. Comme nous le verrons par la suite, la situation devient encore plus compliquée lorsque les programmes sont composés de plusieurs fichiers et de packages. Un avantage important de l'utilisation d'un IDE est d'éviter d'avoir à spécifier tous ces paramètres.

- **Mettre à jour les variables d'environnement *Path* et *Classpath* sous macOS et Linux**

Sous macOS et Linux, on peut aussi modifier les variables d'environnement au besoin. Pour modifier la variable PATH, il suffit généralement d'éditer le fichier texte « .profile » (en le créant au besoin) dans le répertoire de

l'utilisateur comme /home/monnom ou /Users/monnom. Il suffit d'y ajouter la ligne « export PATH=/chemin/vers/java:$PATH ». La modification de la variable CLASSPATH est similaire. L'installation de Java par l'entremise d'AdoptOpenJDK vous dispense, par défaut, d'ajuster les variables d'environnement.

Attention au conflit entre plusieurs installations de JSE

S'il y a plusieurs installations de JSE sur votre système, les chemins des différentes installations peuvent entrer en conflit. En effet, les chemins sont parcourus en séquence par Windows lors de la recherche d'un programme. C'est le premier rencontré qui sera choisi. Il faut donc placer le chemin voulu en premier pour qu'il soit sélectionné. Ceci peut entraîner des conflits ennuyeux lorsque le chemin prioritaire doit varier selon le contexte. Si vous êtes débutant avec Windows, n'installez pas plusieurs JSE !

2. Introduction à la programmation Java

Ce chapitre introduit les principes de base de la programmation avec le langage Java. Le programme suivant est utilisé pour introduire quelques concepts fondamentaux. Dans un premier temps, chacune des lignes du programme sera examinée l'une après l'autre.

Exemple. JavaPasAPas/chapitre_2/Exemple1.java[15]

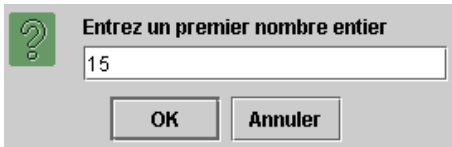
Ce programme permet de saisir deux nombres entiers par le clavier de l'ordinateur dans des fenêtres de dialogue et d'en afficher la somme dans une autre fenêtre.

```
/**
 * Exemple1.java
 * Ce programme saisit deux entiers et en affiche la somme
 */
import javax.swing.JOptionPane; // Importe la
 classe javax.swing.JOptionPane
public class Exemple1{
   public static void main (String args[]) {
      // Déclaration de variables
      String chaine1, chaine2;     // Les entiers lus
      //  sous forme de String
      int entier1, entier2, somme;  // Les entiers à
        additionner et la somme
      // Saisir les deux chaînes de caractères qui
      // représentent des nombres entiers
      chaine1 = JOptionPane.showInputDialog(
        "Entrez un premier nombre entier");
      chaine2 = JOptionPane.showInputDialog(
        "Entrez un second nombre entier");
      // Convertir les deux chaînes de caractères en entiers
      entier1 = Integer.parseInt(chaine1);
      entier2 = Integer.parseInt(chaine2);
      // Calculer la somme des deux entiers
      somme = entier1 + entier2;
      // Afficher la somme avec JOptionPane.showMessageDialog
      JOptionPane.showMessageDialog(null,"La somme des deux entiers est " + somme);
      // Appel de System.exit(0) nécessaire à cause des appels à
      // JOptionPane.showInputDialog
      // et JOptionPane.showMessageDialog
      System.exit(0);
   }
}
```

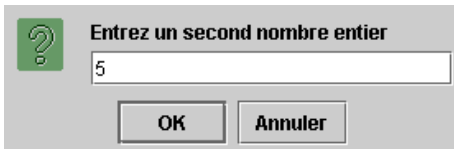
Exercice. Éditer, compiler et faire exécuter ce programme.

[15] Ce titre est un lien vers le répertoire Github qui contient le code des exemples et exercices : https://github.com/RobertGodin/JavaPasAPas

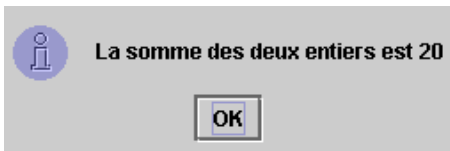
Le scénario suivant illustre le résultat de cette exécution. D'abord la fenêtre de dialogue suivante est affichée. Elle vous permet d'entrer un nombre entier dans la zone d'édition. Lorsque le nombre est entré, il faut cliquer sur le bouton OK pour poursuivre.



Ensuite, la fenêtre de dialogue suivante est affichée et permet de saisir un second nombre entier.



Enfin, la fenêtre suivante affiche la somme des deux entiers lus. Il faut cliquer sur le bouton OK pour terminer le programme.



Examinons maintenant les détails du code du programme.

2.1 Commentaire Java

Le programme *Exemple1* débute par un commentaire :

```
/**
 * Exemple1.java
 * Ce programme saisit deux entiers et en affiche la somme
 */
```

Toute portion du texte source qui débute par /* et se termine par */ est considérée comme un commentaire en Java et n'a aucun effet du point de vue de l'exécution du programme. En d'autres termes, on peut enlever tous les commentaires sans changer le fonctionnement du programme. L'objectif d'un commentaire est de faciliter la compréhension du programme par les

humains (programmeurs). Un commentaire de ce type peut s'étendre sur une ou plusieurs lignes.

Souvent, par souci d'élégance, on débute un tel commentaire par deux astérisque (/**), suivi de lignes qui débutent par un espace suivi d'un astérisque, celui-ci étant aligné avec le second astérisque de la première ligne. On termine le commentaire avec une ligne distincte contenant un seul astérisque aligné sur les autres.

Une autre façon de spécifier un commentaire consiste à le débuter par // comme dans :

```
 import javax.swing.JOptionPane; // Importe la classe
javax.swing.JOptionPane
```

Le texte qui suit le // est considéré comme un commentaire. Un tel commentaire se termine automatiquement à la fin de la ligne courante et ne peut donc pas chevaucher plusieurs lignes.

***Notation* <fin de ligne>**

La notation <fin de ligne> est employée pour représenter la fin de ligne dans le texte de cet ouvrage. La représentation exacte de la fin de ligne peut différer selon le codage employé pour les caractères.

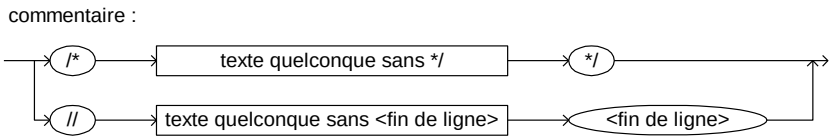
Le *diagramme syntaxique* suivant montre la forme générale d'un commentaire Java.

Diagramme syntaxique

Un diagramme syntaxique est un graphe qui permet de représenter les règles de syntaxe d'un langage de programmation. Le sens des flèches indique comment enchaîner les différents éléments. Un ovale contient des caractères spécifiques. Un rectangle fait référence à une autre règle de syntaxe.

Dans le diagramme suivant, le rectangle contenant le titre « texte quelconque sans */ » représente une séquence de caractères quelconque qui ne peut contenir la suite des deux caractères */. La règle représentée par le rectangle

devrait être détaillée dans un autre diagramme syntaxique. Comme le sens de la règle est facile à comprendre, la règle détaillée est omise ici.



2.2 Importation de classes

Il ne serait pas très pratique devoir écrire entièrement un programme logiciel en utilisant un seul fichier. Les languages de programmation permettent donc de répartir le code en plusieurs fichiers. Et, en particulier, le langage Java vient avec une librairie de code (appelée parfois la librairie standard) très riche que nous pouvons utiliser dans tous les programmes Java. Cette librairie est organisée en « classes », un concept sur lequel nous reviendrons sous peu. La ligne

```
import javax.swing.JOptionPane; // Importe la classe javax.swing.JOptionPane
```

indique que notre programme utilise par la suite un « bout de programme » qui est défini ailleurs. Le bout de programme est une *classe* au sens de Java. Le nom complet de la classe est *javax.swing.JOptionPane*. Le *import* a pour effet de simplifier l'écriture du programme car il est suffisant d'utiliser le nom de classe *JOptionPane* par la suite pour identifier la classe plutôt que d'écrire son nom complet. Techniquement, donc, cette ligne débutant par « import » n'est pas nécessaire : elle équivaut à dire que partout où nous rencontrons *JOptionPane*, on fait référence à *javax.swing.JOptionPane*.

2.3 Packages

Le préfixe « *javax.swing* » du nom complet de classe *javax.swing.JOptionPane* représente un nom de *package*. Un *package* au sens de Java est tout simplement un regroupement de classes.[16] Ce regroupement de classes en *packages* permet d'organiser les classes afin de les retrouver plus facilement. Le rôle des *packages* est analogue aux dossiers (*répertoires*) des systèmes de gestion de fichiers. Un *package* peut contenir des classes ainsi que des *packages*. Dans notre exemple, le *package swing* fait partie du *package javax*. Le nom complet

[16] En français, nous pourrions utiliser le terme paquetage.

d'une classe doit ainsi contenir la liste des *packages* séparés par des points (.). L'ordre des *packages* doit respecter l'ordre de la hiérarchie des contenus.

La figure suivante montre les trois *packages* principaux de Java avec la notation UML. UML (*Unified Modeling Language*) est une notation graphique normalisée qui permet de représenter divers aspects des logiciels.[17]

Figure 6. Principaux packages de Java.

Le *package javax* contient le *package swing* ainsi que d'autres *packages* illustrés à la figure suivante.

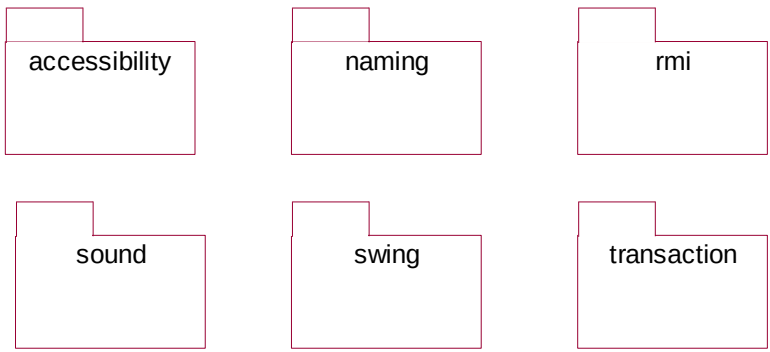


Figure 7. Sous-packages de *javax*.

Enfin, le package *swing* contient la classe *JOptionPane* ainsi que d'autres packages et classes visant le développement d'interfaces à l'utilisateur graphiques. Une classe est représentée par un rectangle en UML. Le rectangle contient le nom de la classe.

[17] Bien que nous utilisons la notation UML dans ce manuel, il n'est pas nécessaire de la maîtriser. Elle peut être comprise de manière intuitive.

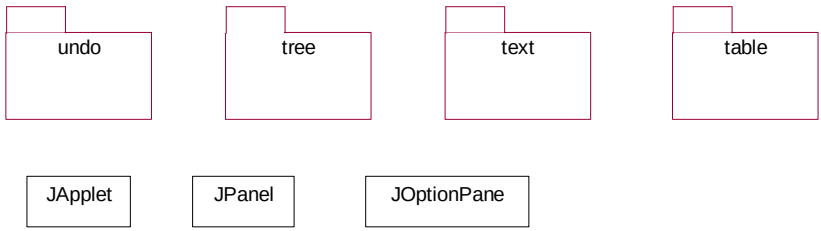


Figure 8. Partie du contenu du package *swing*.

Ainsi, le nom complet de la classe *JOptionPane* est *javax.swing.JOptionPane* parce que la classe *JOptionPane* fait partie du package *swing* qui lui-même fait partie du package *javax*.

2.4 Notion de classe et de méthode

Qu'est-ce qu'une classe ? Pour l'instant, il serait périlleux de tenter de décrire tous les détails de ce concept. En première approximation, un programme Java est composé d'un ensemble de classes.[18] Une classe est donc en quelque sorte un morceau d'un programme Java. Une classe regroupe des *méthodes* ; une *méthode* représente une *action* (*instruction*, *opération*) qui porte un nom et que l'on peut appeler (invoquer) pour effectuer une tâche particulière. Une méthode peut elle-même appeler d'autres méthodes.

Notre programme appelle la méthode *showInputDialog*() de la classe *javax.swing.JOptionPane* qui est une classe prédéfinie en Java. Le JSE contient un grand nombre de classes prédéfinies que tout programme Java peut utiliser. Ces classes prédéfinies fournissent toutes sortes de méthodes que l'on peut appeler pour effectuer diverses tâches sans avoir à les programmer à chacune des utilisations.

Il ne faut pas oublier d'indiquer par des directives *import* les classes prédéfinies nécessaires à notre programme à moins de préciser le nom complet à chacune des utilisations. Sinon, une erreur sera soulevée à la compilation du programme.

[18] Tous les langages de programmation n'ont pas cette caractéristique. Certains langages n'ont pas de classes du tout (comme le C) alors que d'autres permettent d'utiliser des classes, mais sans obligation que tout le code soit au sein de classes (comme le C++).

2.5 Le nom d'une classe

Notre petit exemple de programme est formé d'une seule classe. Le nom de classe (ici *Exemple1*) est défini à la ligne suivante :

```
public class Exemple1{
```

Le contenu de la classe, incluant ses méthodes, vient après son nom et il est placé entre une accolade ouvrante et une accolade fermante. Chacune des classes possède un nom permettant aux autres classes d'y faire référence. Le mot réservé *public* signifie que la classe est publique au sens où elle peut être utilisée par les autres classes sans restriction. Le sens de cette notion sera précisé ultérieurement.

2.6 La méthode *main*()

Au sein de la classe *Exemple1*, la ligne

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

débute la définition de la méthode nommée *main*. Après le nom *main*(), une liste de paramètres formels est spécifiée entre parenthèses :

```
(String args[])
```

La méthode a donc seul paramètre, de type *String*[], soit un tableau de chaînes de caractères. Nous reviendrons sur les paramètres par la suite. Dans un premier temps, ces paramètres ne seront pas utilisés et vous pouvez les ignorer.

Une classe comporte en général un ensemble de méthodes. Chacune des méthodes porte un nom permettant de l'identifier. Java doit être à même de toujours distinguer les méthodes. Les méthodes sont donc uniques en ce qui a trait à leur « signature » : si une classe a deux méthodes du même nom, ces méthodes se distinguent en ce qui a trait aux paramètres. La méthode *main*(*String*[]) est une méthode spéciale qui est exécutée au démarrage du programme. Lors de l'exécution de notre exemple *Exemple1* par la commande

```
java Exemple1
```

que l'on entre à l'invite de commande, le traitement démarre toujours par l'exécution de la méthode *main*(*String*[]) de la classe *Exemple1*.

Les mots réservés *public static void* qui précèdent le nom de la méthode décrivent certaines caractéristiques de la méthode. Pour le moment, vous

pouvez ignorer le sens de ces caractéristiques. Pour les curieux, voici une description sommaire :

- *public* signifie que la méthode peut être appelée de partout et en particulier en dehors de la classe elle-même ;

- *static* signifie que c'est une méthode de classe (par opposition à une méthode d'objet) ;

- *void* signifie que la méthode ne retourne rien.

2.7 Corps d'une méthode

Après les paramètres, vient le corps de la méthode entre accolades. Le corps d'une méthode spécifie ce que la méthode fait. Dans le corps, on retrouve entre autres des énoncés de déclarations de variables et des énoncés d'actions (aussi appelés *instructions* ou *opérations*) qui précisent le traitement à effectuer.

2.7.1 Déclaration de variables

La ligne suivante dans le corps de la méthode *main*()

```
String chaine1, chaine2; // Les entiers sous
            // forme de chaînes
```

est une déclaration de deux variables nommées *chaine1* et *chaine2*. Ces deux variables servent à mémoriser les deux séquences de chiffres qui représentent les entiers à additionner.

Variable

Une variable est un contenant pour une *valeur*. Une variable est donc en quelque sorte une petite mémoire qui permet de stocker une donnée pour utilisation ultérieure. À un moment donné de l'exécution d'un programme, une variable contient une et une seule valeur. Cependant, une variable peut changer de valeur à la suite de l'exécution d'instructions du programme. Votre ordinateur et Java se chargent de stocker la valeur en question : elle peut être stockée au sein d'un ou plusieurs registres, au sein de la mémoire tampon et au sein de mémoire centrale de l'ordinateur.[19]

[19] Dans la pratique, la valeur d'une variable peut être représentée de différentes manières et il arrive fréquemment qu'il y ait plusieurs copies de la valeur dans votre ordinateur. Par exemple,

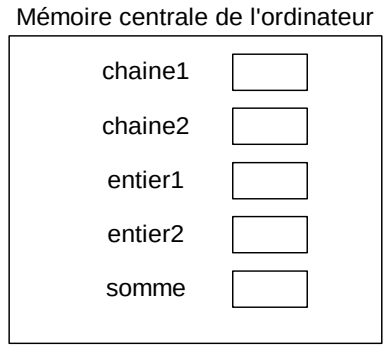
Une variable est identifiée par un nom. Supposons dans un premier temps que le nom est unique dans le corps d'une méthode. Le nom lui-même est sans signification d'un point de vue du langage de programmation mais devrait être choisi avec soin afin d'indiquer aux programmeurs le rôle de la variable dans le contexte du programme.

Une variable ne peut contenir n'importe quoi. Une déclaration de variables débute par un *type* qui détermine l'ensemble des valeurs possibles de la variable. Dans notre exemple, **String** est le type de la variable *chaine1* ainsi que de *chaine2*. Ceci signifie que *chaine1*ne peut contenir qu'une chaîne de caractères. Il en est de même pour *chaine2*. Une chaîne de caractère est une séquence de caractères où chacun des caractères provient d'un code préétabli (tel que *Unicode*). Dans le cas du Java, le code utilisé par l'interface (String) est UTF-16 : dans cet encodage, chaque caractère occupe deux octets au minimum, alors que certains caractères comme les émojis nécessitent quatre octets.

La ligne suivante déclare deux variables qui serviront à mémoriser les entiers eux-mêmes, car *int* représente le type entier en Java.

```
int entier1, entier2, somme;  // Les entiers à additionner
```

Après ces déclarations, on peut imaginer que cinq zones de mémoire ont été réservés pour les cinq variables tel qu'illustré à la figure suivante :



une variable contenant un entier pourra être chargé à partir de la mémoire centrale dans un registre d'un cœur de votre processor, et en ce faisant être propagé dans la mémoire tampon. Un autre cœur de votre processeur pourra charger la valeur de cette variable au même moment dans un autre registre. La plupart du temps, vous n'avez pas à vous préoccuper de ces détails.

Au départ, le contenu des variables est vide. En Java, cela signifie que si on tente d'accéder à la valeur de la variable, nous obtiendrons une erreur lors de la compilation du programme.

2.7.2 Types prédéfinis de Java

Java inclut un ensemble de types prédéfinis. Le tableau suivant énumère les types dits primitifs avec les valeurs possibles.

Type primitif	Valeurs
`boolean`	`true/false`
`char`	Caractère selon l'encodage UTF-16 (mot de 16 bits)
`byte`	Octet en binaire (8 bits) entre -128 (-2^7) et 127 (2^7-1)
`short`	Entier (précision de 16 bits) entre -32 768 (-2^{15}) et 32 767 (2^{15}-1)
`int`	Entier (précision de 32 bits) entre -2 147 483 648 (-2^{31}) et 2 147 483 647 (2^{31}-1)
`long`	Entier (précision de 64 bits) entre -9 223 372 036 854 775 808 (-2^{63}) et 9 223 372 036 854 775 807 (2^{63}-1)
`float`	Nombre à virgule flottante (précision de 32 bits selon le code IEEE 754-1985) entre $-3.4*10^{38}$ et $3.4*10^{38}$ (7 chiffres significatifs)
`double`	Nombre à virgule flottante (précision de 64 bits IEEE 754-1985) entre $-1.7*10^{308}$ et $1.7*10^{308}$ (15 chiffres significatifs)

Figure 9. Types primitifs de Java.

En informatique, on définit l'ensemble des nombres positifs comme étant les nombres plus grands que zéro. Les nombres négatifs sont les nombres plus petits que zéro. Les nombres entiers (par ex., short, int) comportent une seule valeur nulle (0) alors que les nombres à virgule flottante (float, double) comportent deux valeurs nulles : le zéro négatif (-0) et le zéro positif (+0). Quand on écrit un nombre avec un point décimal en Java (par ex., 3.1416), celui-ci est interprété comme un nombre à virgule flottante ayant une précision de 64 bits. Les nombres à virgule flottante comportent aussi les valeurs infinies (Double.POSITIVE_INFINITY et Double.NEGATIVE_INFINITY). La division d'un nombre non nul par

une valeur nulle en Java donne une valeur infinie. La division d'une valeur nulle par une autre valeur nulle donne une valeur spéciale (NaN pour Not a Number) qui a la propriété unique de ne pas être égale à elle-même : l'expression 0.0/0.0 == 0.0/0.0 est fausse en Java. En général, il est possible de représenter l'ensemble des nombres réels entre $-1.7*10^{308}$ et $1.7*10^{308}$ avec 15 chiffres de précision, mais pas 16 chiffres de précision. Par exemple, les nombres comportant 16 chiffres significatifs 0.8825149536132812 et 0.8825149536132813 sont représentés en nombre à virgule flottante comme étant 115673 fois 2 à la puissance -17 ce qui est le nombre 0.88251495361328125. Nous avons donc que l'expression 0.8825149536132812 == 0.8825149536132813 est vraie en Java.

Le type **String** est aussi un type prédéfini mais n'est pas un type primitif. En fait, **String**, dont le nom complet est *java.lang.String*, est une classe Java qui fait partie du package *java.lang*. Ainsi un type peut être soit un type primitif ou une classe. La différence sera expliquée ultérieurement.

2.7.3 Appel de méthode de classe, paramètres et énoncé d'affectation

La ligne

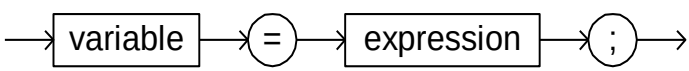
```
 chaine1 = JOptionPane.showInputDialog("Entrez un premier nombre entier");
```

représente un *appel* (ou invocation) *de la méthode de classe javax.swing.JOptionPane.***showInputDialog**() et l'affectation du résultat de l'appel de la méthode à la variable *chaine1*.

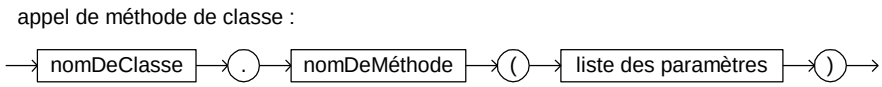
Le terme *affectation* signifie : prendre la valeur de ce qui est produit dans l'expression à droite, et donner cette valeur à la variable qui est à gauche. Le « *javax.swing.* » est omis étant donné la clause *import*.

Cette ligne est un exemple d'un *énoncé d'affectation* simple dont la forme générale est :

énoncé d'affectation (cas simple):

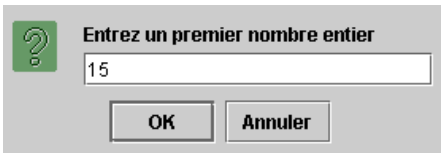


L'expression dans notre exemple est un appel d'une méthode de classe dont la forme générale est :



La méthode provoque d'abord l'affichage de la fenêtre de dialogue suivante :

Cette fenêtre de dialogue permet à l'utilisateur du programme d'entrer une séquence de caractères qui vise à représenter le premier nombre entier à additionner. Par exemple, dans la figure suivante l'utilisateur a entré la séquence des deux chiffres "15".



Lorsque l'utilisateur clique OK, la séquence de caractères saisie est retournée par la méthode et stockée dans la variable *chaine1* par l'énoncé d'affectation. Un appel de méthode doit retourner quelque chose pour qu'il puisse apparaître dans la partie droite d'une affectation. Ce ne sont pas toutes les méthodes qui retournent quelque chose.

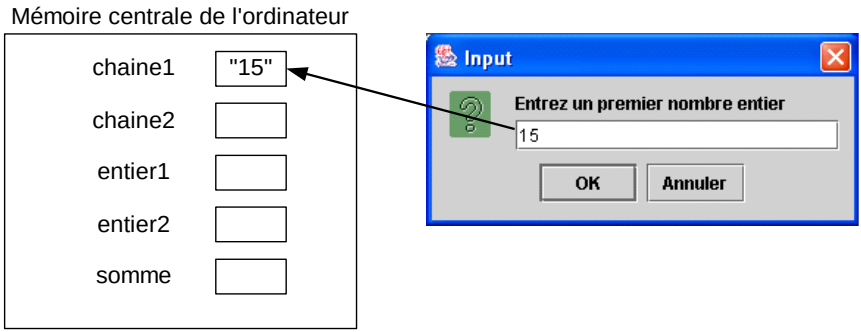


Figure 10. Effet de *chaine1* = *JOptionPane.showInputDialog*("Entrez un premier nombre entier");

Un aspect important à saisir dans cette opération est le fait que la séquence de caractères lue n'est pas interprétée comme un nombre entier à ce point-ci mais comme une chaîne de caractères (type **String**).

Dans l'appel d'une méthode, il faut préciser les valeurs des *paramètres* entre parenthèses après le nom de la méthode. Un paramètre représente une valeur qui est utilisée par la méthode. Un paramètre de méthode est analogue à un paramètre de fonction en mathématiques. Par opposition aux paramètres en mathématiques, un paramètre de méthode n'est pas limité à des valeurs numériques. Le texte

```
"Entrez un premier nombre entier"
```

est la valeur du paramètre de la méthode *JOptionPane*.**showInputDialog**() dans notre exemple. Il représente un titre qui est affiché dans la fenêtre de dialogue. En Java, une séquence de caractères entre guillemets (") est interprétée comme une chaîne de caractères, c'est-à-dire une valeur de type **String**. Le type de la valeur passée en paramètre doit toujours être conforme au type attendu par la méthode. Dans le cas de la méthode **showInputDialog**(), la méthode attend un paramètre de type **String**.

Notation pour les méthodes

Pour désigner une méthode dans le texte, on utilise souvent la notation *nomMéthode*(*listeParamètres*) ou *nomMéthode*(). Les paramètres ne sont pas toujours mentionnés. Pour *showInputDialog*, la notation *showInputDialog*() a déjà été employée. Pour préciser la nature des paramètres, on utiliserait la notation

showInputDialog(*String titre*)

On peut ainsi observer que la méthode a besoin d'un paramètre appelé *titre* de type **String**. Le nom du paramètre (ici titre) ne fait pas partie de la signature de la méthode en Java.

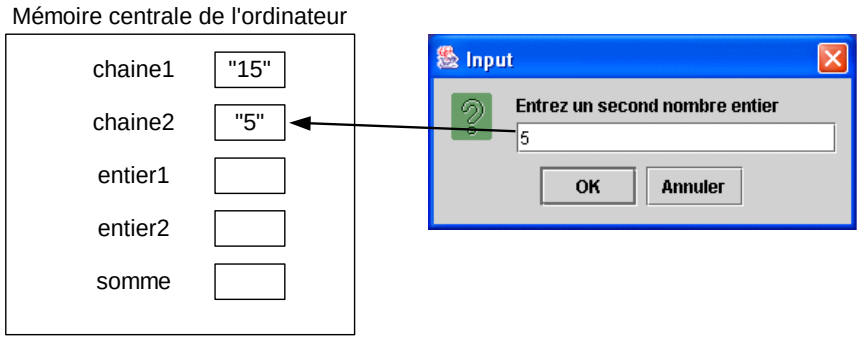
La méthode produit en résultat un autre **String** ("15" dans notre exemple) qui est la séquence de caractères saisie. Pour récupérer ce résultat de l'appel de la méthode, la valeur retournée est affectée à la variable *chaine1*. Le symbole = représente une *opération d'affectation* qui signifie de prendre le

résultat de ce qui est à droite du = et de le placer dans la variable qui est à gauche du =. Pour que cela soit acceptable, il faut que le type de ce qui vient de la partie droite soit compatible avec le type de la variable de la partie gauche. Dans notre cas, il faut que la méthode retourne un **String**.

La ligne suivante permet de saisir la séquence des chiffres du deuxième entier dans la variable *chaine2* :

```
 chaine2 = JOptionPane.showInputDialog("Entrez un second nombre entier");
```

Dans la figure suivante, l'utilisateur a entré "5" :

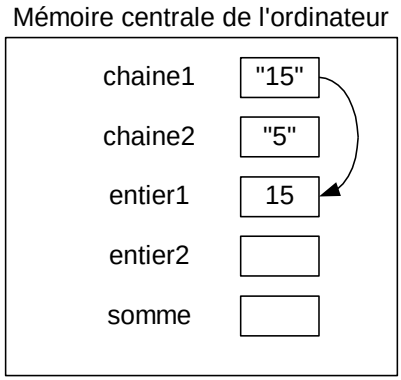


Avant de pouvoir les additionner, il faut convertir les chaines de caractères en nombres entiers. La ligne suivante convertit la chaîne de caractères *chaine1* en un nombre entier qui est affecté à la variable *entier1* de type *int*.

```
        entier1 = Integer.parseInt(chaine1);
```

En effet, la chaîne de caractères contient une série de valeurs à 16 bits, et dans ce cas deux valeurs correspondant à 1 et 5. Il faut convertir ces valeurs en un entier (type int) occupant 32 bits. En Java, on représente la valeur correspondante (soit 15) avec le mot à 32 bits : 00000000000000000000000000001111.

Voici l'effet dans notre exemple.

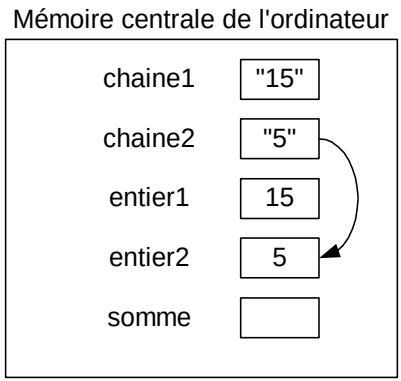


La méthode *Integer.parseInt*() accepte un paramètre **String** et retourne la conversion du **String** en un entier de type *int*[20]. Vous vous demandez peut-être pour quelle raison il n'y a pas de clause *import* pour la classe *Integer* contrairement à *JOptionPane* ? La raison est que *Integer* fait partie du package *java.lang* (nom complet *java.lang.Integer*) dont toutes les classes sont importées automatiquement dans tous les programmes Java.

La ligne suivante convertit le deuxième entier :

```
entier2 = Integer.parseInt(chaine2);
```

Dans notre exemple, cela produit l'effet suivant en mémoire :



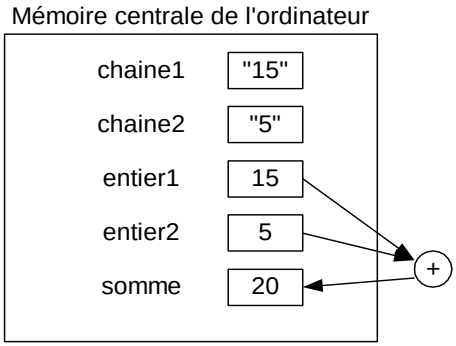
[20] Si la chaîne contient autre chose que des chiffres, une erreur sera provoquée.

2.7.4 Expression

La ligne suivante calcule la somme des deux entiers et affecte le résultat à la variable *somme* :

```
somme = entier1 + entier2;
```

Cet exemple permet d'illustrer que la partie droite d'une instruction d'affectation peut être une expression analogue à une expression mathématique. Dans le cas de types numériques, l'expression peut inclure des opérations arithmétiques typiques telles que l'addition (+) et la soustraction (-), la multiplication (*), la division réelle (/), des parenthèses, etc.[21] L'expression peut aussi inclure des appels de méthodes. Java inclut un grand nombre de méthodes pour les fonctions numériques (dans la classe *java.lang.Math*).

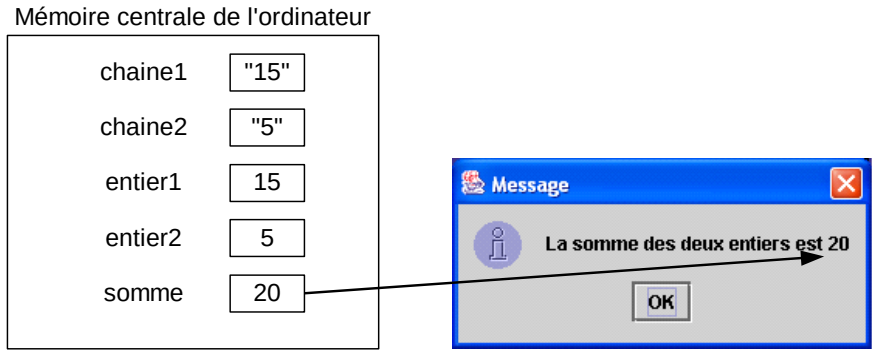


La ligne suivante affiche la somme dans une fenêtre de dialogue :

```
JOptionPane.showMessageDialog(null,"La somme des deux entiers
est " + somme);
```

Dans notre exemple, on obtient :

[21] Il faut que le type du résultat de l'évaluation de l'expression soit compatible avec le type de la variable de la partie gauche de l'opération d'affectation.



La méthode nécessite deux paramètres qui sont séparés par une virgule. Selon la convention Java, tous les paramètres passés à une méthode doivent être séparés par des virgules. Le premier paramètre (valeur spéciale *null*) n'est pas utile pour notre exemple et nous n'expliquerons pas son rôle pour le moment. Le deuxième paramètre est un **String** qui est affiché dans la fenêtre de dialogue.

2.7.5 Expression de type *String*

Dans notre exemple, la valeur du paramètre est en réalité produite par une expression :

```
"La somme des deux entiers est " + somme
```

Cette expression devrait paraître curieuse pour un non-initié. En effet, l'expression additionne un **String** ("La somme des deux entiers est ") à une valeur d'une variable de type *int* (20 dans notre exemple) ? En réalité, le « + » dans cette expression représente une opération de concaténation de deux chaînes de caractères. Le compilateur Java fait cette interprétation car le premier opérande est un **String**. Ainsi, le symbole + possède un sens différent en fonction du contexte. On dit que le symbole + est *surchargé*, car il a plus d'un sens.

La concaténation a pour effet de placer deux chaînes bout à bout. Mais, la variable *somme* n'est pas un **String** mais un *int* ? Le compilateur Java effectue automatiquement une conversion de l'entier en un **String**. Dans notre exemple, le *int* 20 dans *somme* est converti en **String** "20". La concaténation des deux **String**

"La somme des deux entiers est "+"20"

produit le **String**

"La somme des deux entiers est 20".

C'est ce **String** qui est ensuite passé en paramètre à la méthode *JOptionPane.showMessageDialog*() pour être affiché dans la fenêtre de dialogue.

Cette conversion automatique, de type entier au type **String**, n'est pas unique au Java. Il permet d'écrire rapidement du code sans trop se soucier du type des variables. Par contre, on peut toujours écrire du code plus explicite :

```
"La somme des deux entiers est " + Integer.toString(somme)
```

Enfin, la ligne suivante doit être appelée pour terminer le programme correctement. Cette ligne n'est pas toujours nécessaire. Elle l'est lorsque le programme utilise des éléments graphiques tels que les fenêtres de dialogue. C'est le cas de notre programme qui fait appel aux méthodes *JOptionPane.showInputDialog*() et *JOptionPane.showMessageDialog*().

```
System.exit(0);
```

2.8 Diagramme de séquence UML

La Figure 11 montre un *diagramme de séquence* UML qui illustre la séquence d'exécution des méthodes dans le contexte de notre petit scénario. Chacun des rectangles de la partie supérieure correspond à une classe. Sous chacune des classes une ligne verticale, représente l'évolution dans le temps de la classe. D'abord, la méthode *main*() de la classe *Exemple1* est appelée automatiquement par l'intermédiaire de la commande *java Exemple1*. Ceci est illustré par une flèche étiquetée par le nom de la méthode appelée. La flèche part de la ligne sous *java.exe* et aboutit à la classe *Exemple1* de notre programme. Un rectangle mince vertical sous une classe signifie que la méthode est en exécution. La méthode *main*() appelle en premier la méthode *showInputDialog*("Entrez un premier nombre entier") de *JOptionPane* qui s'exécute. Après son exécution, elle retourne le **String** "15" qui correspond au premier entier. La fin de l'exécution d'une méthode est illustrée par une flèche pointillée étiquetée de la valeur retournée par la méthode. Ensuite, la méthode *main*() poursuit son travail en appelant de nouveau la méthode *showInputDialog*("Entrez un second nombre entier") qui retourne le **String** "5" au second appel. La méthode *main*() appelle alors *parseInt*("15") de la

classe *Integer* qui retourne le *int* 15. Ensuite, *main*() appelle *parseInt*("5") de la classe *Integer* qui retourne le *int* 5. La méthode *main*() calcule alors la *somme* et appelle *showMessageDialog*("La somme des deux entiers est 20") pour l'afficher. Enfin, *main*() appelle *exit*(0) de la classe *System*.

Ainsi un programme Java effectue un traitement par le résultat de l'interaction entre plusieurs classes par l'intermédiaire d'appels de méthodes d'une classe à l'autre. Dans notre premier exemple, une seule classe a été définie et cette classe a été utilisée trois fois. Sauf dans les cas les plus simples, le codage d'un programme Java nécessite habituellement la définition de plusieurs classes.

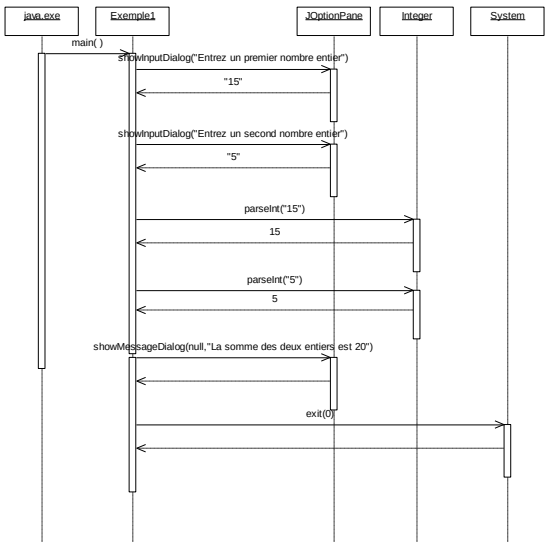


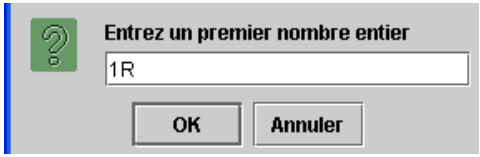
Figure 11. Diagramme de séquence de l'exécution de *Exemple1*.

2.9 Exceptions

Lors de l'exécution d'un programme, divers problèmes peuvent se produire. Par exemple, si l'utilisateur entre autres chose qu'une chaîne de caractère qui représente un nombre entier dans le dialogue de saisie, la méthode *Integer.parseInt*() ne pourra pas convertir la chaîne en un entier ! Ceci provoquera une interruption anormale du programme. Ce genre d'interruption est appelé une *exception* en Java. Lorsqu'une exception est

levée, le programme est interrompu et un message d'erreur est affiché. Ce message permet de retracer l'origine de l'exception.

Par exemple, supposons que l'utilisateur entre la chaîne "1R" plutôt que "15" :



Lorsque l'appel de la méthode *Integer.parseInt*("1R") est effectué, une exception est levée et le message d'erreur suivant est retourné :

```
java.lang.NumberFormatException: 1R
int java.lang.Integer.parseInt(java.lang.String, int)
int java.lang.Integer.parseInt(java.lang.String)
void Exemple1.main(java.lang.String[])
```

Ce message indique la classe de l'exception (*java.lang.NumberFormatException*) et la valeur de paramètre qui a causé le problème (1R). Ensuite, la séquence des appels de méthode qui a conduit au problème est donnée en ordre inverse des appels. Ici, on peut voir que la méthode *main* de la classe *Exemple1* a appelé la méthode *parseInt* de la classe *java.lang.Integer*. La méthode *java.lang.Integer.parseInt*(*java.lang.String*) appelle une autre méthode *java.lang.Integer.parseInt*(*java.lang.String*, *int*) du même nom mais avec deux paramètres ? D'une part, ceci montre que les méthodes que notre programme appelle peuvent elles-mêmes appeler d'autres méthodes. D'autre part, nous verrons plus loin qu'il est possible de définir plusieurs méthodes différentes du même nom sans qu'il n'y ait d'ambiguïté lorsque les paramètres sont différents. Le compilateur peut lever l'ambigüité par la forme des paramètres passés à l'appel.

2.10 Syntaxe des identificateurs Java

Les noms de variables, méthodes et classes doivent respecter des règles précises qui s'appliquent à tous les identificateurs Java. Un identificateur Java doit débuter par une lettre suivie d'une suite d'un nombre quelconque de caractères limités aux lettres, chiffres, $ et _. Les lettres peuvent être majuscules ou minuscules et la casse est importante. Par exemple, « somme » et « Somme » sont deux identificateurs différents en Java. Certains identificateurs sont dits réservés et ne peuvent servir de nom de variable,

méthode ou classe. Ces identificateurs réservés ont un sens prédéfini en Java. Par exemple, les identificateurs *class*, *import* et *int* sont réservés.

2.11 Disposition du texte

Les règles de Java concernant la disposition du texte sont assez flexibles au sens où les éléments du langage peuvent être séparés par une suite quelconque d'espaces blancs. Le terme *espace blanc* désigne un *espace*, une marque de *tabulation* ou une *fin de ligne*. Un compilateur Java ne prend donc pas en compte les espaces blancs lorsqu'il traite votre code informatique. Cependant, il est utile d'utiliser une manière systématique de disposer les différentes parties d'un programme source afin d'en faciliter la lecture. N'oubliez pas qu'un code qui se lit bien sera plus facile à corriger et à modifier. Certains éditeurs mettent automatiquement en forme votre code informatique pour en assurer la lisibilité.

Certains langages de programmation, comme les langages Python et le Swift, prennent en compte les espaces et les tabulations. D'autres langages, comme le language Go, disposent d'une norme fixe pour l'utilisation des espaces et des tabulations.

Lorsque vous travaillez en équipe, ou lorsque vous entendez contribuez au code de quelqu'un d'autre, il est mal vu de reformatter leur code. Vous devez alors modifier le moins possible leur code.

2.12 Initialisation de variable à la déclaration

Dans *Exemple1*, les variables sont déclarées au début du corps de la méthode *main*(). Ceci n'est pas obligatoire. En fait, il est possible de retarder la déclaration à sa première utilisation comme illustré dans *Exemple2*.

Exemple. JavaPasAPas/chapitre_2/Exemple2.java

```
/**
 * Exemple2.java
 * Ce programme saisit deux entiers et en affiche la somme
 */
import javax.swing.JOptionPane; // Importe la classe javax.swing.JOptionPane
public class Exemple2{

   public static void main (String args[]) {

      // Saisir les deux chaînes de caractères qui représentent des nombres entiers
      String chaine1 = JOptionPane.showInputDialog("Entrez un premier nombre entier");
      String chaine2 = JOptionPane.showInputDialog("Entrez un second nombre entier");
```

```
    // Convertir les chaînes en entiers
    int entier1 = Integer.parseInt(chaine1);
    int entier2 = Integer.parseInt(chaine2);

    // Calculer la somme des deux entiers
    int somme = entier1 + entier2;

    // Afficher la somme avec JOptionPane.showMessageDialog
    JOptionPane.showMessageDialog(null,"La somme des deux entiers est " + somme);

    // Appel de System.exit(0) nécessaire à cause des appels à
    // JOptionPane.showInputDialog et JOptionPane.showMessageDialog
    System.exit(0);
  }
}
```

La ligne suivante déclare le type de *chaine1* et l'initialise en même temps par l'appel à *showInputDialog*().

```
String chaine1 = JOptionPane.showInputDialog("Entrez un premier nombre entier");
```

Il est permis de déclarer une variable n'importe où dans le corps d'une méthode et il est permis de l'initialiser au moment de sa déclaration.

Erreur de programmation

L'utilisation d'une variable non déclarée provoque une erreur de compilation.

Déclarer une variable déjà déclarée provoque une erreur de compilation.

L'utilisation de la valeur d'une variable non initialisée provoque une erreur à l'exécution.

Exercice. Modifiez le programme *Exemple1* afin qu'il lise trois entiers (plutôt que deux) et en affiche la somme.

Solution. La solution suivante est une adaptation directe de *Exemple1.java*. La nouvelle variable *chaine3* de type **String** est introduite pour y lire la troisième chaîne de caractère ainsi qu'une variable *entier3* pour l'entier correspondant.

JavaPasAPas/chapitre_2/Exercice1.java

```java
/**
 * Exercice1.java
 * Lire 3 entiers et en afficher la somme
 */
import javax.swing.JOptionPane;
public class Exercice1{

   public static void main (String args[]) {

      String chaine1 = JOptionPane.showInputDialog("Entrez un premier nombre entier");
      String chaine2 = JOptionPane.showInputDialog("Entrez un second nombre entier");
      String chaine3 = JOptionPane.showInputDialog("Entrez un troisième nombre entier");

      int entier1 = Integer.parseInt(chaine1);
      int entier2 = Integer.parseInt(chaine2);
      int entier3 = Integer.parseInt(chaine3);

      int somme = entier1 + entier2 + entier3;

      JOptionPane.showMessageDialog(null,"La somme des trois entiers est " + somme);
      System.exit(0);
   }
}
```

Exercice. Écrivez un programme qui effectue le même traitement que le précédent mais en utilisant une seule variable de type **String** (plutôt que trois) et une variable entière plutôt que quatre !

Solution. La solution suivante réutilise la même variable *chaine* pour lire chacune des chaînes de caractère qui représente un entier et la même variable *somme* pour cumuler les valeurs intermédiaires.

JavaPasAPas/chapitre_2/Exercice2.java

```
/**
 * Exercice2.java
 * Lire 3 entiers et en faire la somme. Utiliser seulement une variable String
 * et deux variables int.
 */
import javax.swing.JOptionPane;
public class Exercice2{

   public static void main (String args[]) {

      String chaine = JOptionPane.showInputDialog ("Entrez un entier dans cette case");
      int somme = Integer.parseInt(chaine);

      chaine = JOptionPane.showInputDialog ("Entrez un entier dans cette case");
      somme = somme + Integer.parseInt(chaine);

      chaine = JOptionPane.showInputDialog ("Entrez un entier dans cette case");
      somme = somme + Integer.parseInt(chaine);

      JOptionPane.showMessageDialog(null,"La somme des trois entiers est " + somme);
      System.exit(0);
   }
}
```

Minimiser la quantité de mémoire consommée

Un aspect à considérer dans le développement d'un programme est la quantité de mémoire consommée. La solution *Exercice2* peut sembler préférable à *Exercice1* de ce point de vue. Cependant, ceci n'est pas le seul aspect à considérer dans l'élaboration d'un programme. Parfois, le fait de minimiser la mémoire consommée de manière extrême peut introduire d'autres défauts tel qu'un temps de traitement plus élevé ou un manque de clarté d'un point de vue de la lisibilité du programme. Par ailleurs, dans le cas qui nous concerne, même si nous utilisons une seule variable, il y aura tout de même trois instances de la classe String.

2.13 Méthode System.out.println()

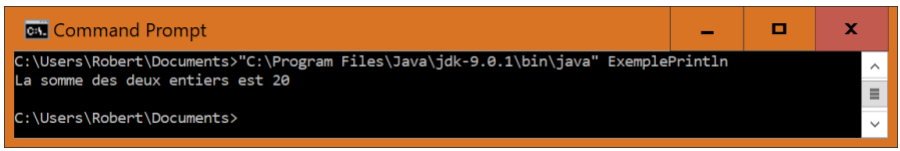
Dans les exemples précédents, nous avons employé la méthode *showMessageDialog*() pour afficher les résultats du programme à l'écran. Une autre manière fréquemment employée consiste à utiliser la méthode *System.out.println*() pour afficher un message dans la fenêtre de commande. De tels messages sont notamment utiles pour mieux comprendre le déroulement du code.

Exemple. Le programme suivant reprend *Exemple1* mais en affichant la somme avec *System.out.println*().

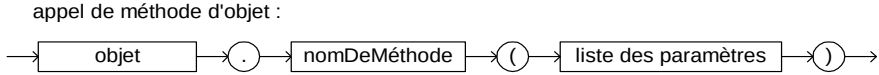
JavaPasAPas/chapitre_2/ExemplePrintln.java

```
/**
 * ExemplePrintln.java
 * Ce programme saisit deux entiers et en affiche la somme avec System.out.println()
 */
import javax.swing.JOptionPane; // Importe la classe javax.swing.JOptionPane
public class ExemplePrintln{

   public static void main (String args[]) {

      // Déclaration de variables
      String chaine1, chaine2;     // Les entiers sous forme de chaînes
      int entier1, entier2, somme;  // Les entiers à additionner

      // Saisir les deux chaînes de caractères qui représentent des nombres entiers
      chaine1 = JOptionPane.showInputDialog("Entrez un premier nombre entier");
      chaine2 = JOptionPane.showInputDialog("Entrez un second nombre entier");

      // Convertir les chaînes en entiers
      entier1 = Integer.parseInt(chaine1);
      entier2 = Integer.parseInt(chaine2);

      // Calculer la somme des deux entiers
      somme = entier1 + entier2;

      // Afficher la somme avec System.out.println()
      System.out.println("La somme des deux entiers est " + somme);

      System.exit(0);
   }
}
```

Sous Windows, le résultat est affiché dans la fenêtre de commande plutôt que dans une fenêtre de dialogue tel qu'illustré par la figure suivante :



La méthode *println*() est une méthode d'objet et non pas une méthode de classe par opposition *à JOptionPane.showInputDialog*() qui est une méthode de classe. On peut voir la différence par la forme de l'appel. L'appel d'une méthode d'objet est de la forme suivante :



L'appel débute par un objet plutôt qu'une classe. Dans notre exemple, le préfixe « *System.out* » représente un objet et non pas une classe. Cet objet correspond à l'unité périphérique de sortie standard de Java qui est la fenêtre de commande lorsque l'on utilise le JSE.

La méthode *println*() représente une méthode d'objet. Qu'est-ce qu'un objet ? Nous ne tenterons pas une explication de cette notion à ce point-ci. Mais, notons qu'une classe peut contenir des méthodes de classe et des méthodes d'objet. Dans un premier temps, dans nos premiers exemples de programmes Java, vous pouvez ignorer la différence entre un appel de méthode de classe ou d'objet.

A noter que l'appel de la méthode *println*() ne retourne rien et l'appel n'est donc pas placé dans une affectation. Il y a des méthodes qui retournent des valeurs et d'autres qui ne retournent rien. Évidemment, même si la méthode ne retourne rien, elle peut avoir un effet ! Dans le cas de *println*(), la méthode affiche quelque chose sur la fenêtre de commande.

Print() vs *println*()

La méthode *println*() provoque un saut de ligne après avoir affiché la chaîne alors que la méthode *print*() fait la même chose que *println*() mais sans saut de ligne.

Exercice. Ecrivez un programme Java qui lit deux entiers et affiche la différence des deux nombres.

Exercice. Écrivez un programme Java qui lit trois entiers *a*, *b* et *c*. Le programme affiche le résultat de :

ab-c.

Exercice. Écrivez un programme Java qui lit trois entiers *a*, *b* et *c*. Le programme affiche le résultat de :

(*a-b*)*c*.

Exercice. Ecrivez un programme Java qui lit deux nombres réels *a* et *b* (type *float*) et affiche *a*/*b*.

Exercice. Ecrivez un programme Java qui lit deux nombres réels (type *double*) qui représentent la largeur et la hauteur d'un rectangle et affiche la surface du rectangle.

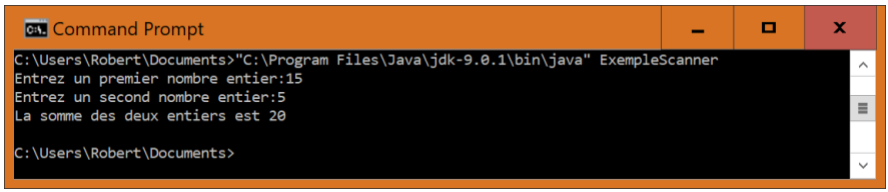
2.14 Classe Scanner

La classe **Scanner** introduite à la version 1.5 de Java permet de saisir des données correspondant à des types primitifs Java directement à la fenêtre de commande (console). Les exercices et exemples des prochains chapitres peuvent être effectué avec **Scanner** plutôt qu'avec les fenêtres de dialogue si la version de Java employée le permet.

Exemple. Le programme suivant saisit deux entiers à la console avec la classe **Scanner** et en affiche la somme.

```
/**
 * ExempleScanner.java
 * Ce programme saisit deux entiers a la console avec Scanner et en affiche la somme avec System.out.println()
 */

import java.util.Scanner;

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

      int entier1, entier2, somme;
      Scanner unScanner = new Scanner(System.in);

      // Saisir les deux entiers avec Scanner
      System.out.print("Entrez un premier nombre entier:");
      entier1 = unScanner.nextInt();
      System.out.print("Entrez un second nombre entier:");
      entier2 = unScanner.nextInt();

      // Calculer la somme des deux entiers
      somme = entier1 + entier2;

      // Afficher la somme avec System.out.println()
      System.out.println("La somme des deux entiers est " + somme);
   }
}
```

Exemple d'exécution :



Exercice. Reprenez les exercices précédents en effectuant la saisie des données avec **Scanner**.

3. Structures de contrôle

Dans le corps d'une méthode, il y a trois manières fondamentales d'enchaîner les actions : la *séquence*, la *boucle* et la *décision*. Ce chapitre introduit les énoncés de base Java qui permettent d'exprimer ces trois types d'enchaînement.

3.1 La séquence

Les exemples vus jusqu'à présent ont utilisé une séquence. Une séquence d'énoncés est de la forme générale suivante :

énoncé$_1$ *énoncé*$_2$ … *énoncé*$_n$

Les énoncés sont placés en séquence les uns après les autres et sont exécutés dans cet ordre. La Figure 1 montre une représentation graphique d'une séquence avec un *diagramme d'activité* UML. Un rectangle aux coins arrondis représente une activité. Dans notre cas, une activité correspond à un énoncé Java. Les flèches indiquent l'ordre d'exécution des activités. Le point noir représente le début et le point noir encerclé la fin de l'exécution.

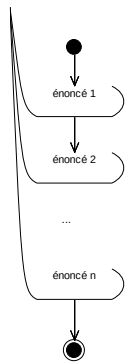
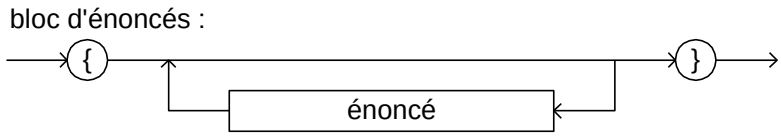
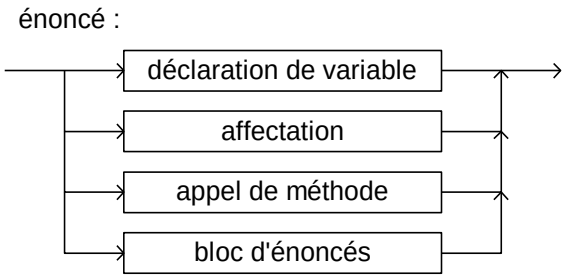


Figure 12. Diagramme d'activité UML pour une séquence

Dans *Exemple1* du chapitre 2, il n'y a pas seulement des énoncés qui correspondent à des actions mais aussi des déclarations. On peut ainsi mélanger les énoncés de déclaration et d'actions dans la séquence. Une séquence d'énoncés (action ou déclaration) placés entre accolades est appelé un *bloc d'énoncés* Java (ou simplement *bloc*). La syntaxe d'un bloc est illustrée à la figure suivante :



Le corps d'une méthode est essentiellement un bloc Java. Il est à noter qu'il est permis d'avoir un seul énoncé dans le bloc. Les différents types d'énoncés seront étudiés en détails. Jusqu'à présent, nous avons rencontrés trois sortes d'énoncés : énoncé de déclaration de variable, d'affectation et d'appel de méthode. Un bloc Java est lui-même considéré comme un énoncé. On obtient ainsi le diagramme syntaxique suivant :



On peut donc imbriquer un bloc Java dans un autre bloc Java.

Exemple. JavaPasAPas/chapitre_3/ExempleBloc.java

Dans l'exemple suivant, les deux énoncés de saisie de chaîne de *Exemple1* ont été regroupés en un bloc qui est imbriqué dans le bloc de la méthode *main*().

```
/**
 * ExempleBloc.java
 * Modification de Exemple1 avec un bloc imbriqué
 */
import javax.swing.JOptionPane; // Importe la classe javax.swing.JOptionPane
public class ExempleBloc{

   public static void main (String args[]) {

      // Déclaration de variables
      String chaine1, chaine2;     // Les entiers sous forme de chaînes
      int entier1, entier2, somme;  // Les entiers à additionner

      // Saisir les deux chaînes de caractères qui représentent des nombres entiers
      {
         chaine1 = JOptionPane.showInputDialog("Entrez un premier nombre entier");
         chaine2 = JOptionPane.showInputDialog("Entrez un second nombre entier");
      }
```

```
    // Convertir les chaînes en entiers
    entier1 = Integer.parseInt(chaine1);
    entier2 = Integer.parseInt(chaine2);

    // Calculer la somme des deux entiers
    somme = entier1 + entier2;

    // Afficher la somme avec JOptionPane.showMessageDialog
    JOptionPane.showMessageDialog(null,"La somme des deux entiers est " + somme);

    // Appel de System.exit(0) nécessaire à cause des appels à
    // JOptionPane.showInputDialog et JOptionPane.showMessageDialog
    System.exit(0);
  }
}
```

Ce programme produit le même effet que *Exemple1*. Dans cet exemple, l'utilisation du bloc imbriqué n'a aucune utilité. Nous verrons par la suite l'importance de cette notion.

3.2 La boucle avec l'énoncé while

Imaginons que l'on veuille afficher les entiers de 1 à 5. On pourrait produire ce résultat avec le programme suivant :

Exemple. JavaPasAPas/chapitre_3/Afficher12345.java

Affichage des entiers de 1 à 5.

```
/**
 * Afficher12345.java
 * Afficher les entiers de 1 à 5
 */
import javax.swing.JOptionPane;
public class Afficher12345{

  public static void main (String args[]) {

    JOptionPane.showMessageDialog(null,"Valeur du compteur: "+1);
    JOptionPane.showMessageDialog(null,"Valeur du compteur: "+2);
    JOptionPane.showMessageDialog(null,"Valeur du compteur: "+3);
    JOptionPane.showMessageDialog(null,"Valeur du compteur: "+4);
    JOptionPane.showMessageDialog(null,"Valeur du compteur: "+5);
    System.exit(0);
  }
}
```

S'il fallait afficher les entiers de 1 à 1 000 000, le programme serait long à écrire… Et même si vous aviez la patience de le faire, ce serait sans aucun doute inefficace parce que le processeur devrait charger et interpréter un

grand nombre d'instructions redondantes. Par ailleurs, si le nombre de répétition n'est pas connu au moment d'écrire le code, par exmple si c'est l'utisateur ou la taille du fichier qui détermine le nombre de répétition, il peut être tout simplement impossible de procéder avec une répétition explicite dans le code. Pour éviter de répéter les énoncés dans le programme, on peut employer une répétition (aussi appelée *boucle* ou *itération*). L'énoncé *while* Java est un des énoncés Java qui permet d'effectuer une répétition.

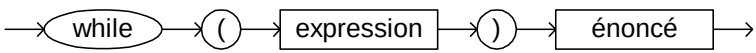
Exemple. JavaPasAPas/chapitre_3/ExempleWhile.java

ExempleWhile illustre la notion de répétition avec compteur en employant un énoncé *while*. Ce programme a le même effet que le précédent.

```
/**
 * ExempleWhile.java
 * Exemple d'utilisation d'une boucle while avec un compteur
 */
import javax.swing.JOptionPane;
public class ExempleWhile{
  public static void main (String args[]) {
      int compteur = 1;
      while(compteur <= 5){
         JOptionPane.showMessageDialog(null,"Valeur du compteur: "+compteur);
         compteur = compteur + 1;
      }
      System.exit(0);
   }
}
```

Voici la syntaxe d'un énoncé *while* :

énoncé while :



L'expression entre parenthèses doit être une *expression booléenne*, aussi appelée *condition* , dont la valeur est vraie (*true*) ou faux (*false*). Si cette condition est respectée (i.e. la valeur retournée par l'*expression* est *true*), l'énoncé après le *while* est répété en boucle jusqu'à ce que la condition ne soit plus respectée (i.e. la valeur retournée par l'expression est *false*).

Il faut prendre soin, lorsque l'on conçoit une boucle, de s'assurer que celle-ci puisse se terminer : une boucle peut être infinie et ne jamais se terminer. Il s'agit généralement d'une erreur.

La figure suivante illustre l'enchaînement des énoncés du programme par un diagramme d'activité UML. Un losange représente une condition. Les flèches qui partent de la condition montrent les deux enchaînements possibles selon le résultat de la condition. Le diagramme montre bien le concept de répétition dans l'enchaînement des énoncés.

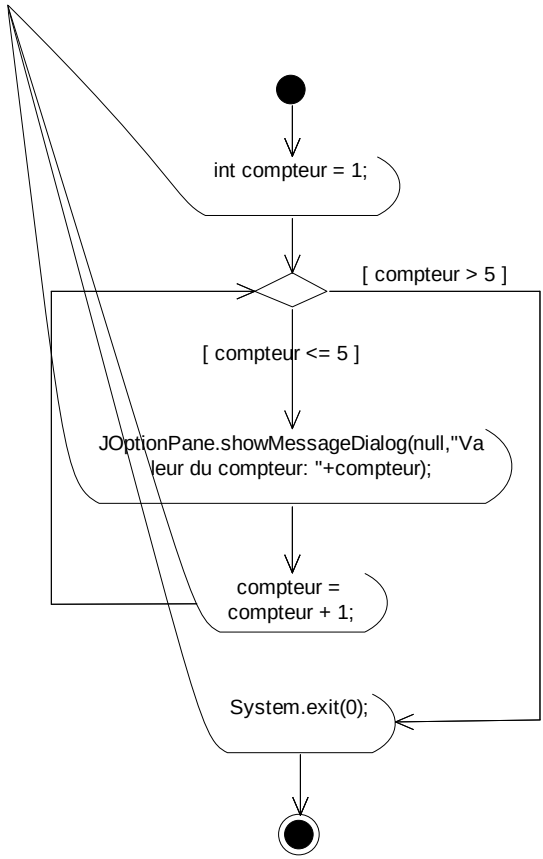


Figure 13. Diagramme d'activité pour le programme.

S'il y a plus d'un énoncé à répéter, comme c'est le cas de notre exemple, il faut les regrouper en un bloc, donc mettre ces énoncés entre accolades. Il est à noter que s'il y a un seul énoncé, les accolades sont facultatives. Dans notre exemple, la condition est

```
(compteur <= 5){
```

Donc, tant que cette condition s'avère vraie (tant que le compteur sera plus petit ou égal à 5), les énoncés

```
JOptionPane.showMessageDialog(null,"Valeur du compteur:
"+compteur);
            compteur = compteur + 1;
```

s'exécuteront en boucle. Dès que le compteur dépasse la valeur de cinq, ces énoncés arrêtent de s'exécuter, et le programme passe à l'énoncé suivant, qui est *System.exit*(0). Ce dernier énoncé met fin au programme.

Expression booléenne

Le nombre de répétition est contrôlé par une expression booléenne. Une expression booléenne peut être formée en comparant des valeurs à l'aide des opérateurs de comparaison du tableau suivant. Il y a quelques différences avec la notation mathématique usuelle.

Opérateur de comparaison	**Signification**
<	Plus petit que
<=	Plus petit ou égal à
>	Plus grand que
>=	Plus grand ou égal à
==	Égal à
!=	N'est pas égal à

Figure 14. Opérateurs de comparaison.

Exercice. Modifiez l'exemple précédent afin qu'il affiche les entiers 0, 2, 4, 6, 8, 10.

Solution. JavaPasAPas/chapitre_3/ExerciceWhile1.java

```
/**
 * ExerciceWhile1.java
 * Afficher les valeurs 0,2,4,6,8,10
 */
import javax.swing.JOptionPane;
public class ExerciceWhile1{
  public static void main (String args[]) {
      int compteur = 0;
      while(compteur <= 10){
         JOptionPane.showMessageDialog(null,"Valeur du compteur: "+compteur);
         compteur = compteur + 2;
      }
      System.exit(0);
   }
}
```

Exercice. Modifiez l'exemple afin d'afficher 5, 4, 3, 2, 1, 0, -1, -2, -3, -4, -5.

Solution. JavaPasAPas/chapitre_3/ExerciceWhile2.java

```
/**
 * ExerciceWhile2.java
 * Afficher les valeurs de compteur 5,4,3,2,1,0,-1,-2,-3,-4,-5
 */
import javax.swing.JOptionPane;
public class ExerciceWhile2{
  public static void main (String args[]) {
      int compteur;
      compteur = 5;
      while(compteur >= -5){
         JOptionPane.showMessageDialog(null,"Valeur du compteur:"+compteur);
         compteur = compteur - 1;
      }
      System.exit(0);
   }
}
```

Exercice. Reprenons l'exemple de lecture d'entiers afin d'en afficher la somme. Nous avons vu le cas de deux et de trois entiers. Maintenant cherchons à extrapoler jusqu'à 10 entiers ! Il serait très long de répéter dix fois la lecture, la conversion et l'accumulation dans somme. Une solution naturelle est d'utiliser un énoncé *while*. Dans cet exercice, utilisez une boucle while pour lire dix entiers et en afficher la somme.

Solution. JavaPasAPas/chapitre_3/ExerciceWhile3.java

```
/**
 * ExerciceWhile3.java
 * Lire dix entiers et en afficher la somme avec un while
 */
import javax.swing.JOptionPane;
public class ExerciceWhile3 extends Object {
   public static void main (String args[]) {
      String serie;
      int entier;
      int compteur = 1;
      int somme = 0;

      while (compteur <=10){
         serie = javax.swing.JOptionPane.showInputDialog("Entrez un entier");
         entier = Integer.parseInt (serie);
         somme = somme + entier;
         compteur = compteur + 1;
      }
      JOptionPane.showMessageDialog(null,"La somme des dix entiers est " + somme);
      System.exit(0);
   }
}
```

Exercice. Supposons que le nombre d'entiers à lire est inconnu à l'avance. Une technique souvent employée pour arrêter la répétition est l'utilisation d'une valeur spéciale appelée sentinelle qui provoque l'arrêt de la répétition. Par exemple, supposons que le nombre 0 représente la sentinelle. Vous devez donc écrire un programme qui lit une série d'entier jusqu'à ce que l'entier 0 soit entré et produit la somme de ces entiers.

Solution. JavaPasAPas/chapitre_3/ExerciceWhileSentinelle.java

```
/**
 * ExerciceWhileSentinelle.java
 * Lire une suite d'entiers jusqu'à ce que l'entier 0 soit entré et afficher la somme
 * des entiers lus.
 */
import javax.swing.JOptionPane;
public class ExerciceWhileSentinelle {
   public static void main (String args[]) {
      String serie;
      int somme = 0;
      int entier = 1; // N'importe quelle valeur différente de 0 ferait l'affaire
      while (entier != 0) {
         serie = JOptionPane.showInputDialog("Entrez un nombre");
         entier = Integer.parseInt (serie);
         somme = somme + entier;
   }
      JOptionPane.showMessageDialog(null,"La somme de tous les nombres est de " +somme+ ".");
      System.exit(0);
   }
}
```

3.3 Qualité du logiciel, tests et débogage

Un aspect fondamental de la qualité d'un programme est la validité des résultats produits par rapport à sa spécification. Une pratique courante en développement de logiciel est de vérifier que les résultats corrects sont produits pour un ensemble de cas de tests.

Exemple. Pour tester le programme précédent, on pourrait employer plusieurs combinaisons de données en entrée et vérifier que pour chacun des cas de tests, le bon résultat est produit.

Numéro de test	Input	Output
1	15 120 30 0	165
2	10 -5 20 0	30
3	0	0
4	2a	Exception

Ce genre de test est dit *fonctionnel* étant donné qu'il vérifie que le fonctionnement du programme est valide par rapport à ce qu'il doit faire (*spécification fonctionnelle*). Dans des programmes plus complexes, d'autres aspects peuvent aussi être mesurés tel que le temps de calcul, la mémoire consommée ou d'autres aspects dits *non fonctionnels*.

Il existe des méthodes et logiciels qui visent à automatiser le processus de vérification par des tests. Généralement, même si tous les tests produisent le résultat correct, ceci ne garantit pas que le programme fonctionne correctement dans tous les cas possibles. Il est habituellement trop complexe en pratique de tester tous les cas. Cependant en choisissant les cas de tests d'une manière judicieuse, on peut obtenir un grand niveau de confiance au sujet du fonctionnement du programme. Différentes stratégies peuvent être employées à cet effet.

Dans l'approche de test par boîte noire ou opaque (*black box testing*), les tests sont choisis sans examiner le code lui-même. On cherche à choisir les cas de tests de manière à produire différentes combinaisons d'input qui couvrent les différentes possibilités prévues dans la spécification du programme. Dans

l'approche de test par boîte blanche ou transparente (*white box testing*, *glass box testing*), les tests sont conçus en tenant compte du code. En particulier, il faut tenter de parcourir toutes les parties du code par l'ensemble des tests.

Lorsqu'un test ne produit pas le résultat voulu, on dit qu'il y a un ou plusieurs bogues (*bug*) dans le programme. Le débogage est le processus d'élimination des bogues. Le débogage peut ressembler à un travail de fin limier qui consiste à trouver le code coupable en examinant les indices produits par l'exécution du programme. Il existe des outils débogueurs qui facilitent le dépistage des bogues par exemple en permettant d'exécuter les énoncés pas à pas et en inspectant les valeurs des variables. En l'absence d'un tel outil, on peut ajouter des énoncés qui affichent l'état de variables à différents endroits du programme pour en suivre le déroulement d'une manière plus détaillée.

Exercice. Vérifiez si les bons résultats sont produits avec les tests précédents pour la solution du dernier exercice.

3.4 La boucle avec l'énoncé for

L'utilisation d'une répétition avec compteur est très fréquente. La boucle *for* simplifie l'écriture de telles boucles.

Exemple. JavaPasAPas/chapitre_3/ExempleForSimple.java

Le programme suivant produit le même effet que *ExempleWhile* en affichant les entiers de 1 à 5. Dans un énoncé *for*, l'initialisation du compteur, l'expression de fin de répétition et la mise-à-jour du compteur sont regroupés entre parenthèses après l'identificateur réservé *for*.

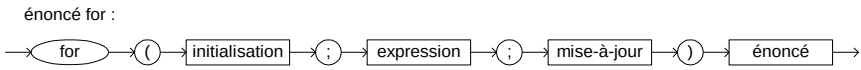
```
/**
 * ExempleForSimple.java
 * Exemple d'utilisation d'un énoncé for qui affiche les entiers de 1 à 5
 */
import javax.swing.JOptionPane;
public class ExempleForSimple {
  public static void main (String args[]) {
   for (int compteur = 1; compteur <=5; compteur = compteur + 1)
      JOptionPane.showMessageDialog(null,"Valeur du compteur: "+compteur);
   System.exit(0);
   }
}
```

Une abréviation syntaxique souvent employée est l'emploi de l'opérateur de post-incrémentation (++) qui a l'effet d'incrémenter de 1.

Exemple. Le for suivant est équivalent au précédent.

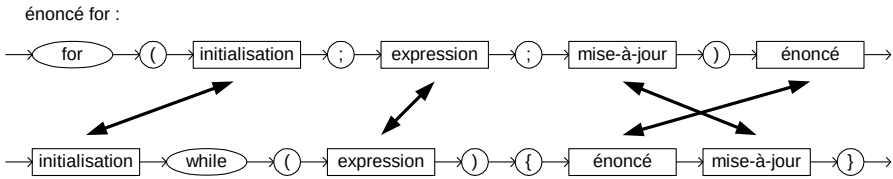
```
    for (int compteur = 1; compteur <=5; compteur++)
```

La syntaxe du *for* est :



A noter qu'il n'y a qu'un énoncé à répéter dans notre exemple, et qu'il n'est pas nécessaire de mettre un bloc pour l'énoncé à répéter. En effet, dans l'exemple, il n'y a pas d'accolades. Cependant, s'il y avait plusieurs énoncés, il aurait été nécessaire de mettre des accolades avant et après les énoncés à répéter.

L'énoncé *for* fonctionne de façon semblable au *while*. La figure suivante montre comment transformer un énoncé *for* en un *while* (il suffit tout simplement de changer les emplacements de certaines composantes). On peut donc se passer du *for* et toujours utiliser le *while*. Le *for* vise tout simplement à simplifier l'écriture des programmes. D'autre part, le *for* n'est pas limité au cas d'un compteur. N'importe quelle expression peut être employée pour décider de la poursuite de la répétition.



Exemple. Les deux bouts de code suivants font la même chose :

```
    for (int compteur = 1; compteur <=5; compteur = compteur + 1)
        JOptionPane.showMessageDialog(null,"Valeur du compteur:
"+compteur);
```

```
        int compteur = 1;
        while(compteur <= 5){
            JOptionPane.showMessageDialog(null,"Valeur          du
compteur: "+compteur);
            compteur = compteur + 1;
```

```
        }
```

Il est possible d'omettre l'initialisation, l'expression ou la mise-à-jour du *for* mais en laissant les « ; ».

Exemple. JavaPasAPas/chapitre_3/ExempleForSentinelle.java

L'exemple suivant reprend l'exercice de lecture d'une série d'entiers avec sentinelle vu précédemment mais en employant un *for* plutôt qu'un *while*. La partie mise-à-jour du *for* est vide mais le dernier « ; » à l'intérieur des parenthèses du *for* doit être présent.

```
/**
 * ExempleForSentinelle.java
 * Lire une suite d'entiers jusqu'à ce que l'entier 0 soit entré et afficher la somme
 * des entiers lus. Exemple illustrant un for sans la partie mise-à-jour.
 */
import javax.swing.JOptionPane;
public class ExempleForSentinelle {
   public static void main (String args[]) {
      String serie;
      int somme = 0;
      for (int entier = 1; entier != 0;){
         serie = JOptionPane.showInputDialog("Entrez un nombre");
         entier = Integer.parseInt (serie);
         somme = somme + entier;
      }
      JOptionPane.showMessageDialog(null,"La somme de tous les nombres est de " +somme+ ".");
      System.exit(0);
   }
}
```

Exercice. Trouver une justification au besoin de conserver le dernier ; du for précédent malgré l'omission de la partie mise-à-jour (indice : essayez de penser comme un ordinateur…).

Exercice. Affichez les entiers 0, 2, 4, 6, 8, 10 avec un *for*.

Exercice. Affichez 5, 4, 3, 2, 1, 0, -1, -2, -3, -4, -5 avec un *for*.

Exercice. Utilisez un *for* pour lire dix entiers et en afficher la somme.

Exercice. Utilisez un *for* pour lire une série d'entier jusqu'à ce que l'entier 0 soit entré (cas de sentinelle) et afficher la somme de ces entiers.

Exercice. Affichez le résultat suivant sur la sortie standard (avec *System.out.print*() et *System.out.println*())

```
1
12
123
1234
12345
123456
1234567
12345678
123456789
```

Solution. JavaPasAPas/chapitre_3/ExerciceForFor.java

Avec *for* imbriqués :

```
/**
 * ExerciceForFor.java
 */
import javax.swing.JOptionPane;
public class ExerciceForFor {
  public static void main (String args[]) {
    for (int compteur1 = 1; compteur1 <=9; compteur1 = compteur1 + 1){
      for (int compteur2 = 1; compteur2 <=compteur1; compteur2 = compteur2 + 1)
        System.out.print(compteur2);
      System.out.println();
    }
  }
}
```

3.5 La décision avec if

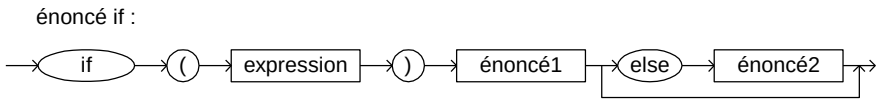
L'énoncé *if* permet au programme de prendre une décision au sujet des actions à exécuter en fonction d'une condition à évaluer.

Exemple. JavaPasAPas/chapitre_3/ExempleIf.java

Le programme suivant lit un entier (*unInt*) et indique s'il est plus grand que 10 ou non. Pour déterminer le message à afficher, une décision est prise en comparant l'entier lu à 10 dans une condition (*unInt* > 10). Selon le résultat de la condition, le *if* permet de choisir entre les deux énoncés alternatifs à exécuter. La première alternative suit la condition et elle est exécutée si la condition est évaluée à vrai (*true*). Sinon (i.e. la valeur de l'expression est *false*), l'énoncé qui suit l'identificateur *else* est exécuté.

```java
/**
 * ExempleIf.java
 * Petit exemple illustrant l'énoncé if.
 */
import javax.swing.JOptionPane;
public class ExempleIf{
   public static void main (String args[]) {
      String unString = JOptionPane.showInputDialog("Entrez un premier nombre entier");
      int unInt = Integer.parseInt(unString);

      // Exemple d'énoncé if
      if (unInt > 10)
         JOptionPane.showMessageDialog(null,unInt + " est plus grand que 10");
      else
         JOptionPane.showMessageDialog(null,unInt + " n'est pas plus grand que 10");

      System.exit(0);
   }
}
```

La syntaxe du *if* est :



Lors de l'utilisation d'un *if*, l'*énoncé1* suivant l'expression ne sera exécutée que si l'expression (la condition) se révèle vraie. L'énoncé ne s'exécute qu'une seule fois. Si l'expression se révèle fausse, l'*énoncé2* suivant le mot *else* sera exécuté (une seule fois). Dans notre exemple, si la condition

```
(unInt > 10)
```

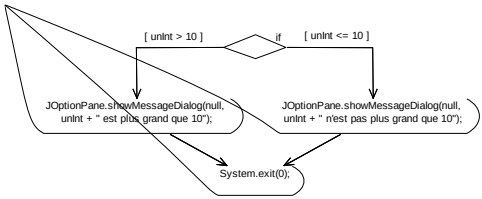
se révèle vraie, l'énoncé

```
            JOptionPane.showMessageDialog(null,unInt + " est plus
grand que 10");
```

sera exécuté. Si l'expression se révèle fausse, l'énoncé qui sera exécuté est

```
            JOptionPane.showMessageDialog(null,unInt  +  "  n'est
pas plus grand que 10");
```

Le diagramme d'activité suivant illustre cet enchaînement.



Il est à noter que s'il y a plusieurs énoncés à exécuter dans la partie *énoncé1* (cas vrai) ou *énoncé2* (cas faux), il faut les regrouper en un bloc, se débutant par l'accolade ouvrante et se terminant par l'accolade fermante.[22] Aussi, la partie *else* est optionnelle. En son absence, lorsque l'expression de condition est fausse, rien n'est exécuté. Ceci peut causer une ambigüité potentielle illustrée par l'exemple suivant.

Exemple. JavaPasAPas/chapitre_3/ExempleElseAmbigu.java

Illustration du *else* ambigu.

```
/**
 * ExempleElseAmbigu.java
 * Petit exemple illustrant l'ambiguïté du else.
 */
import javax.swing.JOptionPane;
public class ExempleElseAmbigu{
   public static void main (String args[]) {
      String chaine1 = JOptionPane.showInputDialog("Entrez un premier nombre entier");
      String chaine2 = JOptionPane.showInputDialog("Entrez un second nombre entier");
      int entier1 = Integer.parseInt(chaine1);
      int entier2 = Integer.parseInt(chaine2);

      // If ambigu
      if (entier1 > 10)
        if (entier2 > 10)
          JOptionPane.showMessageDialog(null,entier1 + " et "+ entier2 + " sont plus grands que 10");
      else
          JOptionPane.showMessageDialog(null,entier1 + " est inférieur ou égal à 10");

      System.exit(0);
   }
}
```

Voici un scénario possible avec ce programme ! Voyez-vous le problème ?

Le problème vient du fait que la disposition du texte donne l'impression que le *else* est associé au premier *if*. Ce n'est pas le cas ! Java associe toujours le *else*

[22] Il est parfois recommandé de toujours mettre des accolades autour des énoncés découlant d'une clause if afin d'éviter les ambiguités.

au *if* le plus rapproché ! La disposition suivante suggère la bonne interprétation :

```
        if (entier1 > 10)
          if (entier2 > 10)
            JOptionPane.showMessageDialog(null,entier1 + " et "+
entier2 + " sont plus grands que 10");
          else
            JOptionPane.showMessageDialog(null,entier1  +  "  est
inférieur ou égal à 10");
```

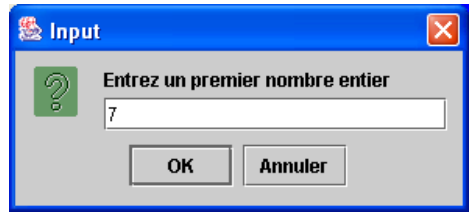
Pour forcer le *else* à être associé au premier *if*, il faut utiliser un bloc Java afin de forcer la terminaison du second *if*.

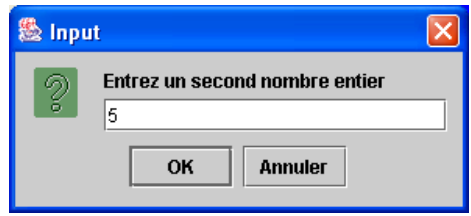
Exemple. JavaPasAPas/chapitre_3/ExempleIfIfElse.java

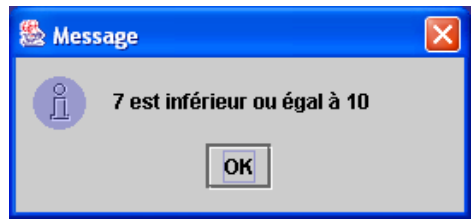
Forcer l'association du *else* avec un *if* éloigné par l'introduction d'un bloc.

```
/**
 * ExempleIfIfElse.java
 * Introduction d'un bloc pour terminer un if sans else
 */
import javax.swing.JOptionPane;
public class ExempleIfIfElse{
   public static void main (String args[]) {
      String chaine1 = JOptionPane.showInputDialog("Entrez un premier nombre entier");
      String chaine2 = JOptionPane.showInputDialog("Entrez un second nombre entier");
      int entier1 = Integer.parseInt(chaine1);
      int entier2 = Integer.parseInt(chaine2);

      // If ambigu
      if (entier1 > 10) {
        if (entier2 > 10)
          JOptionPane.showMessageDialog(null,entier1 + " et "+ entier2 + " sont plus grands que 10");
      }
      else
          JOptionPane.showMessageDialog(null,entier1 + " est inférieur ou égal à 10");

      System.exit(0);
   }
}
```

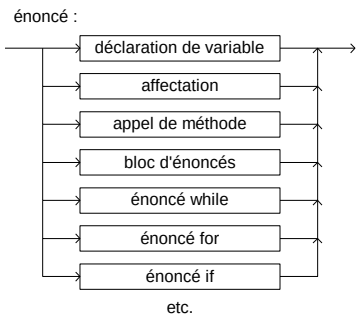
Voici un scénario avec cette nouvelle version :

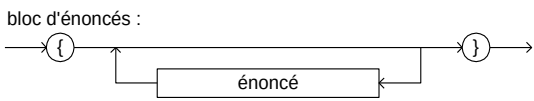






Nous avons maintenant vu les trois manières d'enchaîner les énoncés : séquence, boucle et choix. Il est possible de combiner ces trois types d'énoncés de manière quelconque. Les diagrammes syntaxiques suivants résument les différents cas d'énoncés vus jusqu'à présent :





Dans un bloc d'énoncé, il peut y avoir un while, dans le while, un if et dans le if, un bloc, etc.

Exercice. Lire deux entiers et afficher la division du premier par le deuxième. Si le diviseur est 0, afficher un message à cet effet.

Solution. JavaPasAPas/chapitre_3/ExerciceIf1.java

```
/**
 * ExerciceIf1.java
 * Lire deux entiers et afficher la division entière.
 * Si le diviseur est 0 afficher un message à cet effet.
 */
import javax.swing.JOptionPane;
public class ExerciceIf1 {

   public static void main (String args[]) {

      String chaine1 = JOptionPane.showInputDialog("Entrez un premier nombre entier");
      String chaine2 = JOptionPane.showInputDialog("Entrez un second nombre entier");
      int entier1 = Integer.parseInt(chaine1);
      int entier2 = Integer.parseInt(chaine2);

      if (entier2 == 0)
         JOptionPane.showMessageDialog(null,"La division est impossible");
      else
         JOptionPane.showMessageDialog(null,entier1 + "/" + entier2 + "=" + entier1 / entier2);
      System.exit(0);
   }
}
```

Exercice. Lire deux entiers et afficher le maximum des deux. S'ils sont égaux, afficher n'importe lequel des deux.

Solution. JavaPasAPas/chapitre_3/ExerciceIfMax2.java

```
/**
 * ExerciceIfMax2.java
 * Lire trois entiers et afficher le maximum des trois
 */
import javax.swing.JOptionPane;
public class ExerciceIfMax2{

   public static void main (String args[]) {

   String chaine1 = JOptionPane.showInputDialog("Entrez un premier nombre entier");
   String chaine2 = JOptionPane.showInputDialog("Entrez un second nombre entier");

   int entier1 = Integer.parseInt(chaine1);
   int entier2 = Integer.parseInt(chaine2);

   if (entier1 > entier2)
     JOptionPane.showMessageDialog(null,"Le maximum des deux entiers est " + entier1);
   else
```

```
    JOptionPane.showMessageDialog(null,"Le maximum des deux entiers est " + entier2);

   System.exit(0);
   }
}
```

Exercice. Lire deux entiers et afficher le plus grand des deux s'il y en a un qui est le plus grand, sinon afficher qu'ils sont égaux.

Exercice. Lire trois entiers et afficher le maximum des trois.

Solution. JavaPasAPas/chapitre_3/ExerciceIfMax3.java

```
/**
 * ExerciceIfMax3.java
 * Lire trois entiers et afficher le maximum des trois
 */
import javax.swing.JOptionPane;
public class ExerciceIfMax3{

   public static void main (String args[]) {

   String chaine1 = JOptionPane.showInputDialog("Entrez un premier nombre entier");
   String chaine2 = JOptionPane.showInputDialog("Entrez un second nombre entier");
   String chaine3 = JOptionPane.showInputDialog("Entrez un troisième nombre entier");

   int entier1 = Integer.parseInt(chaine1);
   int entier2 = Integer.parseInt(chaine2);
   int entier3 = Integer.parseInt(chaine3);

   if (entier1 > entier2)
      if (entier1 > entier3)
         JOptionPane.showMessageDialog(null,"Le maximum des trois entiers est " + entier1);
      else
         JOptionPane.showMessageDialog(null,"Le maximum des trois entiers est " + entier3);
   else
      if (entier2 > entier3)
         JOptionPane.showMessageDialog(null,"Le maximum des trois entiers est " +entier2);
      else
         JOptionPane.showMessageDialog(null,"Le maximum des trois entiers est " +entier3);
   System.exit(0);
   }
}
```

On peut imaginer la complexité de cette méthode si on accroît le nombre d'entiers à lire. Dans l'exercice suivant, cherchez à utiliser une boucle pour simplifier le code.

Exercice. Lire 5 entiers et afficher l'entier maximal (le plus grand des cinq entiers).

Solution. JavaPasAPas/chapitre_3/ExerciceWhileIf.java

```
/**
 * ExerciceWhileIf.java
 * Lire 5 entiers et afficher l'entier maximal
 */
import javax.swing.JOptionPane;
public class ExerciceWhileIf{
  public static void main (String args[]) {
    String chaine = JOptionPane.showInputDialog("Entrez un nombre");
    int plusGrand = Integer.parseInt (chaine);
    for (int compteur = 1; compteur < 5; compteur=compteur+1){
      chaine = JOptionPane.showInputDialog("Entrez un nombre entier");
      int entier = Integer.parseInt (chaine);
      if (entier > plusGrand) {
        plusGrand = entier;
      }
    }
    JOptionPane.showMessageDialog(null,"L'entier le plus grand est " + plusGrand);
    System.exit(0);
  }
}
```

Cette solution montre un exemple de *if* imbriqué dans une boucle.

Exercice *. Afficher les nombres premiers plus petits que 100.

Exercice. Lire une note entre 0 et 100 inclusivement et afficher la lettre correspondante selon le barème suivant :

0<=note<60	E
60<=note<70	D
70<=note<80	C
80<=note<90	B
90<=note<=100	A

Types et expressions Java

Dans les premiers chapitres, nous avons rencontré les types *int* et **String** et quelques expressions simples. Ce chapitre approfondit les notions de type et d'expression Java.

3.6 Type primitif et littéral

Rappelons que Java inclut les types *primitifs* du tableau suivant.

Type primitif	**Valeurs**	**Exemples de littéraux**	**Signification**
boolean	true ou false	true false	Vrai faux
char	Caractère selon le code standard UNICODE (16 bits)	'a' 'A' '5' ' ' '\'' '\u0061' '\u000a'	La lettre a (minuscule) Lettre A (majuscule) Le chiffre 5 Un espace Le caractère ' Le caractère *a* dont le code Unicode est 0061 (en hexadécimal) Le caractère non imprimable qui correspond à un saut de ligne
byte	Octet en binaire (8 bits) entre -128 (-2^7) et 127 (2^7-1)	(byte)15	Conversion explicite d'un littéral int
short	Entier (précision de 16 bits) entre -32 768 (-2^{15}) et 32 767 ($2^{15}-1$)	(short)325	Conversion explicite d'un littéral int
int	Entier (précision de 32 bits) entre -2 147 483 648 (-2^{31}) et 2 147 483 647 ($2^{31}-1$)	3015 -3015 0X2A 017	Par défaut, un littéral entier est de type *int* Le moins unaire est employé pour un nombre négatif Représentation hexadécimale du int 42 (2*16+10) Représentation octale du int 15 (1*8 + 7)
long	Entier (précision de 64 bits) entre - 9 223 372 036 854 775 808 (-2^{63}) et 9 223 372 036 8	3015L 3015l	Le L ou l à la fin du littéral entier indique que le nombre entier est de type *long*

	54 775 807 (2^{63}-1)		
float	Nombre réel (précision de 32 bits selon le code standard IEEE 754-1985) entre $-3.4*10^{38}$ et $3.4*10^{38}$ (7 chiffres significatifs)	45.032F 45.032f 32F 32f 12.453E5F 12.053E-5F	Le F ou f à la fin du littéral numérique indique que le nombre doit être de type *float* 1245300 (La notation E*n* représente 10^n) 0.0001205
double	Nombre réel (précision de 64 bits selon le code standard IEEE 754-1985) entre $-1.7*10^{308}$ et $1.7*10^{308}$ (15 chiffres significatifs)	45.032 45.032D 45.032d 32D 32d 12.453E5 12.053E-5 124.	Par défaut, un littéral numérique avec partie décimale est de type *double* Le D ou d à la fin indique que le nombre doit être de type *double* 1245300 (La notation E*n* représente 10^n) 0.0001205

Figure 15. Types primitifs de Java.

Chacun des types primitifs correspond à un ensemble de valeurs décrit dans le tableau de la Figure 15. Une variable d'un type primitif contient une valeur parmi l'ensemble des valeurs du type.

Littéral (*litteral*)

Une valeur particulière d'un type primitif exprimée par une chaîne de caractères dans le code source d'un programme est appelée un *littéral*. Le tableau précédent montre des exemples de littéraux pour chacun des types primitifs.

3.7 Types et expressions numériques

Les types primitifs numériques *byte*, *short*, *int*, *long* correspondent à des entiers de différentes tailles : ils utilisent une quantité variable de mémoire, et ils peuvent représenter des nombres plus ou moins volumineux. À l'exception du type « char » qui peut être considéré comme un type représentant des entiers (de 0 à 65536), les types entiers en Java sont toujours signés dans le sens où ils permettent de représenter à la fois les nombres positifs et négatifs. En contraste, dans des langages comme Go, Swift, C#, C++, etc., il existe des types d'*entiers signés* et *non-signés* (*unsigned*). Notez qu'il y a toujours une valeur négative de plus grande amplitude que n'importe quelle valeur positive (par exemple, -128, -32768, -2147483648, etc.) ce qui signifie qu'on ne peut

toujours prendre la valeur absolue : par exemple, la valeur 2147483648 ne peut pas être représentée par un « int » alors que -2147483648 est parfaitement légitime.

Les types *float* et *double* sont des nombres réels avec partie fractionnaire. En Java, on ne peut pas représenter les valeurs réelles (comme π). On utilise plutôt des nombres à virgule flottante. Ainsi donc, on peut utiliser des « double » pour consacrer 64 bits afin de représenter des nombres. On utilise alors la norme « binary64 » qui accorde 53 bits à la mantisse d'une représentation binaire. En d'autres mots, on peut pratiquement représenter n'importe quel nombre de la forme $m\ 2^p$ tant que m n'excède pas 2^{53}. En particulier, le type « double » en Java peut représenter tous les entiers (positifs et négatifs) qui n'excèdent pas une magnitude de 2^{53}. Quand un nombre ne peut pas être pas être représenté, Java va trouver le nombre à virgule flottante le plus proche. Si nous arrivons exactement entre deux nombres à virgule flottante, comme c'est le cas avec le nombre 9000000000000002.5, Java va choisir le nombre le plus proche qui a une mantisse paire (ici 9000000000000002). Tous les nombres décimaux à 6 chiffres significatifs peuvent être représentés avec le type *float*. Par contre, certains nombres comprenant 6 chiffres significatifs correspondent à plus d'un nombre de type *float*. On peut distinguer deux nombres de type *float* en utilisant 9 chiffres significatifs. Tous les nombres décimaux à 15 chiffres significatifs peuvent être représentés avec le type *double*; on peut distinguer deux nombres de type *double* en utilisant 17 chiffres significatifs. Dans la pratique, nous utilisons donc souvent soit 9 chiffres significatifs, soit 17 chiffres significatifs afin de stocker en format décimal les nombres. Par exemple, si on sait que le résultat sera stocké sous la forme d'un double, il est inutile de saisir le nombre π avec plus de précision que 3.1415926535897932. Les types *float* et *double* peuvent aussi avoir pour valeur –infini, +infini ainsi qu'une valeur spéciale NaN (not-a-number) que nous pouvons générer en divisant zéro par zéro (par exemple).

Exemple. JavaPasAPas/chapitre_3/ExempleZero.java

L'exemple suivant illustre le comportement du zéro positif, du zéro négatif et du NaN et Java. Il affiche ceci à l'écran :

```
true
-Infinity
Infinity
```

```
false
NaN
False
```

```
public class ExempleZero {
 public static void main(String[] s) {
   double minus_zero = -0.0;
   double plus_zero = +0.0;
   System.out.println(minus_zero == plus_zero);
   System.out.println(1/minus_zero);
   System.out.println(1/plus_zero);
   System.out.println(1/minus_zero == 1/plus_zero);
   double n = 0.0 / 0.0;
   System.out.println(n);
   System.out.println(n == n);
 }
}
```

La Figure 15 montre les conventions Java pour exprimer les littéraux des types numériques. Java permet de former des expressions numériques complexes de manière analogue aux expressions mathématiques. Le tableau suivant montre les opérations numériques binaires de Java.

Opération binaire	Signification
+	Addition Exemples : 4 + 5 donne 9, 4.2 + 16.3 donne 20.5
-	Soustraction Exemples : 5 - 2 donne 3, 20.5 – 16.3 donne 4.2
*	Multiplication Exemple : 3 * 4 donne 12
/	Division Exemples : 16 / 5 donne 3, 16.0 / 5.0 donne 3.2
%	Reste après division entière Exemples : 16 % 4 donne 0, 16 % 5 donne 1

La division des nombres entiers positifs se fait comme à la petite école : le résultat de la division est un entier qui représente le nombre de fois que le diviseur peut être soustrait au numérateur sans obtenir un résultat négatif. Par exemple, 4/4, 5/4 et 7/4 donnent toutes comme réponse le quotient 1. Le reste de la division est obtenu avec le symbole %. On peut vérifier que l'entier x divise l'entier y en comparant y % x avec la valeur zéro.

La division par zéro (avec / ou %) dans le cas des entiers génère une erreur et peut terminer un programme. La division par zéro dans le cas des nombres

à virgule flottante est permise, mais elle génère une valeur infinie ou la valeur NaN.

Dans le cas d'une operation arithmétique dont le résultat ne peut pas être représenter par le type choisi, un résultat incorrect peut être obtenu. Il est de votre responsabilité de vérifier que le résultat du calcul peut être représenté.

Exemple. Le *int* 2000000000 ajouté au *int* 2000000000 donne la valeur -294967296 :

```
int x = 2000000000;
int y = 2000000000;
int z = x + y; // = -294967296
```

Il est permis de former des expressions arithmétiques complexes en combinant les opérations au besoin. Comme pour les conventions mathématiques usuelles, lorsqu'il y a plusieurs opérations, les opérations à plus grande priorité sont effectuées en premier. À priorité égale, les opérations sont effectuées de gauche à droite. Le tableau suivant montre la priorité des opérateurs en ordre décroissant de priorité. Le + et - *unaires* servent à préciser le signe d'une valeur numérique comme dans +3 ou -15, le + étant toujours facultatif.

Opération	Priorité
()	0
+, - unaires	1
*, /, %	2
+, - binaires	3

Exemple illustrant les priorités. L'expression suivante

```
3 + 2 * 6 - 3 - 2 * 4
```

est équivalente à

```
(((3 + (2 * 6)) - 3) - (2*4))
```

dont le résultat est 4. L'évaluation procède donc selon les étapes suivantes :

```
(((3 + (2 * 6)) - 3) - (2*4))
(((3 + 12) - 3) - (2*4))
```

```
(((3 + 12) - 3) - 8)
((15 - 3) - 8)
(12 - 8)
4
```

Les parenthèses permettent de modifier cet ordre d'évaluation au besoin.

Exercice. Quel est le résultat de l'expression suivante :

```
2+4*2*5+10/2
```

Réécrire l'expression avec des parenthèses qui reflètent la priorité d'évaluation des opérations.

Conseil

Ne vous fiez pas à la priorité, et mettez des parenthèses en cas de doute !

Lorsque des opérandes de types différents sont combinés, Java effectue des conversions de type automatiques en convertissant à un type unique tous les opérandes de l'expression.

Exemple. L'expression suivante

```
3.4 + 7
```

fait intervenir le *double* 3.4 et le *int* 7. Le *int* sera converti automatiquement en un *double* avant d'effectuer l'opération.

La conversion cherche à éviter la perte d'information en faisant une *promotion* à un type plus général. Le tableau suivant montre les promotions valides en Java pour les types numériques. Les conversions sont appliquées non seulement à l'évaluation d'expressions mais aussi lors de l'affectation du résultat de l'expression à une variable et lors du passage d'un paramètre.

Type	Promotions automatiques valides
double	aucune
float	double
long	float ou double
int	long, float ou double
short	int, long, float ou double
byte	short, int, long, float ou double

Lorsqu'une conversion non valide est voulue par le programmeur, il est possible de forcer une conversion par une opération de conversion (*cast*) dont la syntaxe est :

```
(nomDuType)valeur
```

La valeur sera alors convertie dans le type entre parenthèses. La conversion peut entraîner une perte de précision comme l'illustre l'exemple suivant.

Exemple. Le *double* 15.2 est converti en *int* 15 et affecté à la variable *unInt* :

```
int unInt = (int)15.2
```

Sans conversion explicite, une erreur serait levée car cette promotion automatique n'est pas valide en Java. La conversion explicite est potentiellement dangereuse. En tant que programmeur, vous avez la responsabilité de vérifier que le nouveau type peut représenter la valeur. Considérons l'exemple suivant.

Exemple. Le *int* 131072 est converti en *short* 0 et affecté à la variable *unShort* :

```
short unShort = (short)131072
```

Conseil de génie logiciel

Il est préférable de spécifier explicitement les conversions dans les expressions pour clarifier le comportement désiré.

Exemple. L'expression suivante

```
3.4 + 7
```

est équivalente à

```
3.4 + (double)7
```

La deuxième formulation est préférable car elle clarifie la conversion désirée.

3.8 Expressions booléennes

Le type *boolean* comprend deux littéraux, *true* et *false*. Ce type est souvent utilisé dans les conditions des énoncés *if* et des énoncés de boucle (*while*, *for*, *do*). Nous avons vu comment formuler des expressions booléennes simples

avec les opérateurs de comparaison (<, ==, >, etc.) dans le chapitre 3. Il est aussi possible de formuler des expressions booléennes complexes avec les opérateurs logiques (et, ou et négation) du tableau suivant :

Opérateur logique	Signification
&	et
|	ou
!	négation

Exemple. JavaPasAPas/chapitre_3/ExempleLogique.java

Le programme suivant illustre les opérateurs logiques &, | et !.

```
/**
 * ExempleLogique.java
 * Petit exemple illustrant l'énoncé if.
 */
import javax.swing.JOptionPane;
public class ExempleLogique{
   public static void main (String args[]) {
      String unString = JOptionPane.showInputDialog("Entrez un premier nombre entier");
      int unInt = Integer.parseInt(unString);
      if (unInt > 10 & unInt < 20) { JOptionPane.showMessageDialog(null,unInt + " est entre 10 et 20"); }
      if (unInt == 100 | unInt == 200) { JOptionPane.showMessageDialog(null,unInt + " est 100 ou 200"); }
      if (!(unInt > 30)) { JOptionPane.showMessageDialog(null,unInt + " n'est pas plus grand que 30"); }
      System.exit(0);
   }
}
```

La condition

```
(unInt > 10 & unInt < 20)
```

est évaluée à vrai si *unInt* est à la fois plus grand que 10 et plus petit que 20. Lorsque l'on utilise un &, il faut que les deux parties de la condition soient vraies pour que le résultat soit vrai.

La condition

```
if (unInt == 100 | unInt == 200)
```
est évaluée à vrai si *unInt* est égal à 10 ou *unInt* est égal à 20. Une seule des deux parties doit être vraie pour que le résultat soit vrai. Le ou n'est pas exclusif, c'est-à-dire que si les deux parties sont vraies, le résultat est vrai.

La condition

```
!(unInt > 30)
```
est évaluée à vrai si *inInt* > 30 est faux, donc si *inInt* <= 30.

Java inclut aussi le && et le ||. Le && est une variante du & qui court-circuite l'évaluation lorsque possible. Le & évalue toujours les deux parties de la condition alors que le && n'évalue pas la deuxième partie si la première est fausse. On dit que l'évaluation est court-circuitée. En effet, si la première partie est fausse, le résultat de la condition sera nécessairement faux. Il n'est donc pas nécessaire d'évaluer la deuxième partie.

Opérateur logique	Signification
&&	et (évaluation court-circuité)
||	ou (évaluation court-circuité)

Exemple. L'expression

$$2 > 3 \;\&\&\; 2 < 10$$

donne *false*. Comme 2 > 3 donne *false*, il n'est pas nécessaire d'évaluer la deuxième partie 2 < 10. Le && évite d'évaluer la seconde partie, 2 < 10. Dans la plupart des circonstances, le résultat de & et && est identique. Cependant, si l'évaluation de la partie droite peut avoir un effet autre que l'évaluation de la condition elle-même, le résultat ne sera pas toujours le même.

Exemple. JavaPasAPas/chapitre_3/ExempleEtCourtcircuite.java

Le programme suivant vérifie si l'entier saisi est un diviseur de 10. Il vérifie si l'entier est non nul et si le reste après division entière est nul. Dans le cas où l'entier est nul, il ne faudrait pas effectuer la division qui provoquerait une exception en Java et l'arrêt du programme. En utilisant le &&, la division ne sera pas effectuée.

```java
/**
 * ExempleEtCourtcircuite.java
 * Petit exemple illustrant l'énoncé if.
 */
import javax.swing.JOptionPane;
public class ExempleEtCourtcircuite{
   public static void main (String args[]) {
      String unString = JOptionPane.showInputDialog("Entrez un premier nombre entier");
      int unInt = Integer.parseInt(unString);

      // Exemple de l'opérateur &
      if (unInt != 0 && 10 % unInt == 0)
         JOptionPane.showMessageDialog(null,unInt + " est un diviseur de 10");
      else
         JOptionPane.showMessageDialog(null,unInt + " n'est pas un diviseur de 10");
      System.exit(0);
   }
}
```

Dans le cas du ||, si la première partie est vraie, la deuxième partie de la condition n'est pas évaluée, car le résultat doit être vrai. L'évaluation se fait toujours de la gauche vers la droite. La première condition à gauche est toujours évaluée en premier.

3.9 Traitement de caractères

Le type *char* correspond souvent en pratique à l'ensemble des caractères. Cet ensemble est défini par le standard Unicode (www.unicode.org). Les codes sont énumérés dans

http://www.unicode.org/Public/UNIDATA/UnicodeData.txt

Plus précisément, un char en Java permet de représenter n'importe quel caractère Unicode pouvant être représenté avec un mot de 16 bits dans la norme UTF-16. Certains caractères spécialisés, comme les émojis, requièrent 32 bits et donc deux mots de 16 bits (deux *char*). La norme Unicode permet aussi de combiner plusieurs caractères pour former un seul caractère visible (par exemple, les émojis « famille »).

On peut aussi voir le type *char* comme une valeur entière entre 0 et 65536. Le type *char* occupe autant d'espace en mémoire que le type short, mais le type *short* est utilisé pour représenter les valeurs entières entre -32768 et 327678.

La Figure 15 montre des exemples de littéraux du type *char*. Un caractère imprimable est représenté entre apostrophes. On peut aussi employer le code Unicode du caractère selon le format :

```
'\uyyyy'
```

où *yyyy* est une suite de quatre chiffres hexadécimaux qui correspond au code Unicode hexadécimal du caractère. Le \u est une *séquence* dite *d'échappement* qui altère l'interprétation normale par le compilateur Java. La séquence indique au compilateur que ce qui suit est le code Unicode du caractère et non pas le caractère lui-même. Le tableau suivant montre d'autres séquences d'échappement prévues en Java.

Séquence d'échappement	**Code Unicode**	**Description**
\ b	\u0008	backspace BS
\ t	\u0009	tabulation HT
\ n	\u000a	saut de ligne LF
\ f	\u000c	saut de page FF
\ r	\u000d	retour de chariot CR
\ "	\u0022	guillemets "
\ '	\u0027	apostrophe '
\ \	\u005c	backslash \

3.9.1 Type *String*, objets et classes

Le type *char* ne permet que de manipuler un caractère à la fois. Il est souvent nécessaire de manipuler une chaîne de caractères, par exemple pour afficher un message ou pour saisir une donnée. Java inclut le type **String** à cet effet. Le type **String** est aussi un type prédéfini mais n'est pas un type primitif. En fait, **String**, dont le nom complet est *java.lang.String*, est une classe Java qui fait partie du package *java.lang*. Le type d'une variable peut aussi être une classe. Dans ce cas, une *valeur* de la variable est une référence à un *objet* de la classe. Par analogie à un type primitif qui correspond à un ensemble de valeurs, une classe correspond à un ensemble d'objets. La notion d'objet est fondamentale en Java ainsi que dans les autres langages objet. Par analogie avec les types primitifs, on peut dire qu'un objet est analogue à une valeur possible pour une classe. La différence entre une valeur d'un type primitif et une référence à un objet d'une classe n'est pas apparente dans les exemples que nous avons rencontrés jusqu'à présent. Nous allons maintenant faire

ressortir certains aspects fondamentaux qui distinguent les objets (de classes) et les valeurs (de types primitifs).

- **Constructeur d'objet**

Pour créer un nouvel objet, il faut normalement employer un *constructeur d'objet*.

Exemple.
JavaPasAPas/chapitre_3/ExempleCreationObjetString.java

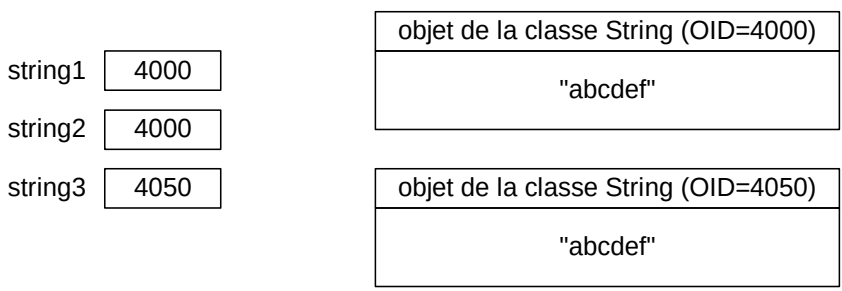
L'exemple de programme suivant permet d'illustrer concrètement la notion d'objet et de constructeur d'objet pour la classe **String**.

```
import javax.swing.JOptionPane; // Importe la classe javax.swing.JOptionPane
public class ExempleCreationObjetString{
   public static void main (String args[]) {
     String string1 = new String("abcdef");
     String string2 = string1;
     String string3 = new String("abcdef");

     System.out.println(string1 == string2); //true
     // string1 et string3 sont deux objets différents
     System.out.println(string1 == string3); //false
     // par contre, string1 et string3 ont le même contenu
     System.out.println(string1.equals(string3)); //true
   }
}
```

Après les trois affectations

```
        String string1 = new String("abcdef");
        String string2 = string1;
        String string3 = new String("abcdef");
```

le résultat suivant est produit :



L'appel *new String*("abcdef") dans la ligne suivante

```
String string1 = new String("abcdef");
```

crée un objet de la classe **String** dont le contenu est "abcdef". L'objet contient la chaîne de caractères "abcdef". Il *contient* la chaîne mais n'est pas la chaîne ! Lorsqu'un objet est créé, un identifiant d'objet (OID) lui est assigné automatiquement. Dans notre exemple, l'objet crée a l'OID = 4000. Cette valeur n'est donnée qu'à titre d'exemple et n'a pas d'importance en soi. On ne doit pas se préoccuper de la manière dont les OID sont générés. En fait, l'OID n'est pas visible dans le programme Java et n'est pas manipulable directement. Un OID est en quelque sorte une adresse pour retrouver un objet. Il est analogue à un numéro d'assurance sociale pour un citoyen. Le numéro en soi n'a pas d'importance. Ce qui compte, c'est qu'il permet d'identifier un citoyen sans ambiguïté. L'adresse en mémoire centrale est une manière de réaliser un OID, mais il y a aussi d'autres implémentations possibles.

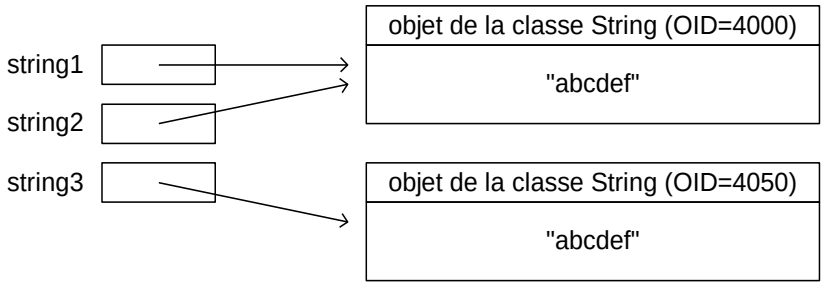
Dans l'énoncé

```
String string1 = new String("abcdef");
```

l'objet de la classe **String** créé dans la partie droite est affecté à la variable *string1* de la partie gauche. Le type de *string1* doit être le même que celui de l'objet créé[23]. C'est pourquoi, le type de *string1* est **String**. Il est fréquent de rencontrer en Java ce genre d'énoncé où une variable de type *ClasseX* est déclarée et on lui affecte un objet de type *ClasseX* créé par *new ClasseX*().

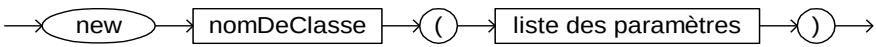
Lorsqu'un objet est affecté à une variable, c'est l'OID de l'objet qui est placé dans la variable. On dit alors que la variable contient une *référence* à l'objet. Souvent les références sont représentées graphiquement par des flèches tel qu'illustré dans la figure suivante car les valeurs exactes des OID sont sans importance. Ce qui compte, c'est que la variable fasse référence au bon objet.

[23] Nous verrons plus loin que ce n'est pas nécessairement le même type dans certaines circonstances



Un objet est créé avec un constructeur d'objet. Un constructeur d'objet est une méthode spéciale dont le rôle est de créer un objet d'une classe. Il est appelé en utilisant la syntaxe suivante :

appel de constructeur d'objet :



Un constructeur porte le même nom que la classe. De ce point de vue, une classe est comme un moule à objet. La classe **String** est donc un moule pour construire des objets de type **String**.

Une valeur de type **String** ne peut pas être modifiée. Une fois que la valeur « abcdef » a été assignée à la variable, on ne peut plus changer la chaîne. C'est un choix spécifique à cette classe puisqu'il aurait été possible pour les créateurs du Java de faire en sorte que la classe **String** puisse modifier sont contenu.

La ligne

```
String string2 = string1;
```

affecte le contenu de *string1* à *string2*. Ceci ne copie pas l'objet mais plutôt l'OID de l'objet. En conséquence, les deux variables font maintenant référence au même objet !

La ligne

```
String string3 = new String("abcdef");
```

crée un **autre** objet de la classe **String**, dont l'OID = 4050. Cet autre objet contient aussi "abcdef" mais c'est un objet différent du premier !

Par opposition aux objets, il n'y a pas de distinction entre une valeur et son contenant pour les types primitifs. Comment fait-on la différence entre la référence à l'objet et le contenu de l'objet dans un programme ? La réponse à cette question est illustrée par le reste du code du programme.

Dans le cas d'objets, le « == » Java compare les références aux objets et non pas le contenu des objets. Ainsi le test *string1 == string2* dans

```
System.out.println(string1 == string2); //true
```

produit la valeur *true* car les deux variables font référence au même objet mais *string1 == string3* dans

```
System.out.println(string1 == string3); //false
```

est *false* car les deux variables *string1* et *string3* font référence à des objets différents ! Pour comparer le contenu des objets **String**, on peut utiliser la méthode *equals*() de la classe **String**. Ainsi le test *string1.equals*(*string3*) dans

```
System.out.println(string1.equals(string3)); //true
```

produit la valeur *true* parce que le contenu des deux objets est le même.

Attention ! Une erreur fréquente en Java est de confondre == et *equals*().

La méthode *equals*() de la classe **String** est une méthode d'objet. La possibilité d'appeler des méthodes sur les objets est un autre aspect qui les distingue des valeurs des types primitifs. Nous avons déjà dit qu'une classe regroupe un ensemble de méthodes. Parmi ces méthodes, il y a des méthodes de classe et des méthodes d'objets. On ne peut appeler une méthode de classe sur un objet ou une méthode d'objet sur une classe.

- **Documentation des classes et méthodes**

Un aspect important de la programmation Java est le fait qu'un grand nombre de classes et de méthodes sont déjà définies et mises à la disposition du programmeur. Le programmeur doit pouvoir facilement retrouver les méthodes et les classes. À cet effet, le programmeur peut consulter la documentation des classes et méthodes prédéfinies. Cette documentation est accessible sur le site de Oracle. Pour la version 8, vous pouvez y accéder par :

https://docs.oracle.com/javase/8/docs/api/

Cette documentation est sous forme HTML, et elle peut être consultée à partir d'un fureteur Web. Notez que même si les versions de Java se succède rapidement, il est souvent possible de tout faire avec les classes de la version 8. Il y a d'ailleurs des bénéfices à ne pas trop rapidement adopter des fonctions et des classes qui ne sont disponibles qu'avec des versions récentes du Java.

Exercice. À ce point-ci, vous devriez vous familiariser un peu avec cette documentation en cherchant la classe **String**. Vous pouvez la retrouver dans la liste *All Classes* du panneau inférieur gauche. Cliquez sur METHOD de la rubrique SUMMARY NESTED dans le panneau de droite en haut et vous obtenez la liste des méthodes de la classe **String**.

- La documentation montre pour chacune des méthodes, le type de ce qui est retourné sous la première colonne du tableau *Method summary*.
    - L'identificateur réservé *void* signifie que la méthode ne retourne rien.
- Les méthodes de classe sont distinguées des méthodes d'objet par l'identificateur réservé *static* qui apparaît avant le type de ce qui est retourné. Par exemple, la méthode **copyValueOf**(char[] data) est une méthode de classe alors que **charAt**(int index) est une méthode d'objet.
- Dans la deuxième colonne du tableau *Method Summary*, la liste des paramètres apparaît après le nom de la méthode. Les paramètres sont séparés par des virgules.
    - Pour chacun des paramètres, il y a son type suivi d'un nom de paramètre. Le nom n'a pas d'importance comme tel. Lorsqu'on appelle la méthode, les paramètres doivent apparaître dans le même ordre et doivent être du bon type.

La classe **String** inclut plusieurs autres méthodes d'objet visant la manipulation de chaînes de caractères.

Exemple. Le programme suivant illustre quelques méthodes de la classe **String**.

```
import javax.swing.JOptionPane; // Importe la classe javax.swing.JOptionPane
public class ExempleMethodesDeString{

   public static void main (String args[]) {

    String string1 = new String("abcdef");
    String string2 = new String("cd");
    System.out.println("String string1 = new String(\"abcdef\")");
    System.out.println("String string2 = new String(\"cd\")");
    System.out.println("La longueur de string1 est :" + string1.length());
    System.out.println("Le caractère en position 2 de string1 est :" + string1.charAt(2));
    System.out.println("La sous-chaine en position 2 de string1 est :" + string1.substring(2));
    System.out.println("La sous-chaine qui débute en position 2 et fini en 4 est :" + string1.substring(2,5));
    System.out.println("La première occurrence de string2 dans string1 est à la position :" + string1.indexOf(string2));
    System.out.println("La concaténation de string2 à string1 donne :" + string1.concat(string2));
    }
}
```

Résultat affiché :

```
String string1 = new String("abcdef")
String string2 = new String("cd")
La longueur de string1 est :6
Le caractère en position 2 de string1 est :c
La sous-chaine en position 2 de string1 est :cdef
La sous-chaine qui débute en position 2 et fini en 4 est :cde
La première occurrence de string2 dans string1 est à la position
:2
La concaténation de string2 à string1 donne :abcdefcd
```

Notez dans la ligne suivante l'utilisation de la séquence d'échappement \". Ceci est nécessaire pour inclure un guillemet dans une chaîne de caractères étant donné que le guillemet est aussi le délimiteur de fin de chaîne.

La méthode **length**() retourne la taille d'un **String**. Elle compte le nombre de mots de 16 bits contenu dans la chaîne de caractères. Dans plusieurs cas, le résultat de la méthode donne le nombre de caractères UTF-16 : tant qu'au caractère n'appartient à un plan supplémentaire. Les plans supplémentaires comprenent les emojis, les symboles mathématiques et musicaux et autres symboles spécialisés.

```
      System.out.println("La longueur de string1 est :" +
string1.length());
```

L'appel *string1*.*charAt*(2) retourne le *caractère* (ou plutôt le mot de 16 bits) en position 2 de *string1*, ce qui correspond au **troisième** *caractère* de "abcdef", c'est-à-dire le caractère « c » car les positions des caractères sont numérotées à partir de la position 0 !

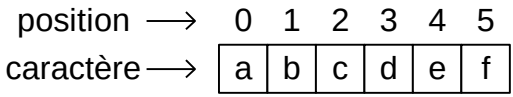


Figure 16. Numérotation des positions des caractères à partir de 0 !

Le fait d'inclure 0 comme indice peut sembler étrange à un non-initié …

L'appel *string1*.*substring*(2) de la ligne suivante retourne un objet de la classe **String** qui contient la sous-chaîne qui débute en position 2 (troisième caractère).

```
System.out.println("La sous-chaine en position 2 de string1 est
:" + string1.substring(2));
```

L'appel *string1*.*substring*(2,5) de la ligne suivante retourne un objet de la classe **String** qui contient la sous-chaîne qui débute en position 2 (troisième caractère) et se termine en position 5 (excluant la position 5, ce qui correspond au cinquième caractère).

```
System.out.println("La sous-chaine qui débute en position 2 et
fini en 4 est :" + string1.substring(2,5));
```

A noter que les deux appels utilisent le même nom de méthode mais avec des paramètres différents ! En réalité, ces deux appels invoquent deux méthodes différentes. D'ailleurs, dans l'extrait suivant de la documentation, la méthode *substring*() apparaît deux fois.

String	**substring**(int beginIndex) Returns a string that is a substring of this string.
String	**substring**(int beginIndex, int endIndex)

Returns a string that is a substring of this string.

Ces deux méthodes de même nom désignent en réalité deux méthodes différentes. Le compilateur peut distinguer les méthodes de même nom par le fait que la nature des paramètres est différente. Dans **substring**(int beginIndex), il n'y a qu'un paramètre de type *int*. Cette méthode retourne un objet **String** dont la chaîne est la sous-chaîne qui débute à la position *beginIndex* et termine au dernier caractère du **String**. Dans **substring**(int beginIndex, int endIndex), il y a deux paramètres et la méthode retourne un objet **String** qui débute à la position `beginIndex` et se termine à la position `endIndex`. Ce sont bien deux méthodes différentes mais apparentées.

Surcharge d'un nom de méthode

Le fait d'employer le même nom pour désigner plusieurs méthodes différentes distinguées par la nature des paramètres est appelé la *surcharge* des noms de méthodes.

Ensuite, l'appel *string1.indexOf*(*string2*) retourne la position de la première occurrence de la chaîne de *string1* dans *string2*.

```
      System.out.println("La première occurrence de string2 dans
string1 est à la position :" + string1.indexOf(string2));
```

Enfin, l'appel *string1.concat*(*string2*) retourne un objet **String** formé de la concaténation de la chaîne de *string2* à la fin de celle de *string1*.

```
System.out.println("La concaténation de string2 à string1 donne
:" + string1.concat(string2));
```

Exercice. Lire un **String**, en afficher la taille, et compter le nombre d'occurrences de la lettre «a» dans le **String**.

- **Littéral *String***

À cause de l'importance de la manipulation de chaînes de caractères, Java prévoit un raccourci pour la création d'objets de la classe **String**. Plutôt que d'utiliser un constructeur pour créer un objet de la classe **String**, Java

permet d'utiliser un *littéral* **String**. Un littéral **String** est une séquence de caractères entre guillemets.

Le littéral est en fait un raccourci pour désigner un objet de la classe **String**. Par exemple, il est permis d'écrire le code suivant :

```
String string1 = "abc";
```

Cette utilisation de littéraux donne l'illusion que **String** est un type primitif. Mais cette vue n'est qu'une partie du portrait car un littéral **String** désigne un objet. Cet énoncé assigne un objet de la classe **String** qui contient la suite de caractères "abc" à la variable *string1*. Ceci produit presque le même effet que :

```
String string1 = new String("abc");
```

L'effet de ces deux énoncés n'est cependant pas exactement le même mais dans la majorité des cas, la différence n'est pas importante. L'exemple suivant illustre la différence.

Exemple. JavaPasAPas/chapitre_3/ExemplesString.java

L'exemple de programme suivant illustre quelques subtilités concernant l'usage de littéraux **String**. Le principe important à saisir est le suivant : l'utilisation d'un littéral qui est connu à la compilation du programme est toujours remplacé par une référence à un objet de la classe **String** qui contient la chaîne de caractère du littéral. S'il y a plusieurs occurrences du même littéral, disons "abc", c'est toujours le même objet qui est utilisé. Ceci permet une certaine économie de mémoire car il n'est pas nécessaire de créer un objet différent à chaque utilisation du même littéral. Cependant, si un objet est créé à l'exécution (par *new* par exemple), ce ne sera pas le même objet !

```
public class ExemplesString{

  public static void main (String args[]) {

    String string1 = "abc";
    String string2 = "def";
    String string3 = "abcdef";
    String string4 = new String("abcdef");

    // Tous les littéraux identiques (à la compilation) sont traduits
    // par une référence au même objet
    System.out.println(string3 == "abcdef"); // true
    System.out.println("abc"+"def" == "abcdef"); //true

    // Cependant, si le littéral est calculé à l'exécution, ce n'est pas le cas
    System.out.println(string1 + string2 == "abcdef"); //false

    // Le constructeur String produit toujours un objet différent de l'objet
    // correspondant au littéral
    System.out.println(string4 == "abcdef"); //false

    // La méthode intern() de la classe String permet de convertir
    // la référence à l'objet correspondant au littéral
    System.out.println((string1 + string2).intern() == "abcdef"); //true
    System.out.println(string4.intern() == "abcdef"); //true

    // La méthode equals() permet de comparer le contenu de l'objet plutôt que la référence
    System.out.println((string1 + string2).equals("abcdef")); //true
    System.out.println(string4.equals("abcdef")); //true
  }
}
```

La condition dans

```
System.out.println(string3 == "abcdef"); // true
```

retourne true car *string3* fait référence au même objet que "abcdef".

La condition dans

```
System.out.println("abc"+"def" == "abcdef"); //true
```

retourne aussi true car la concaténation "abc"+"def" est calculée au moment de la compilation, ce qui produit le littéral "abcdef".

La condition dans

```
System.out.println(string1 + string2 == "abcdef"); //false
```

retourne false car la concaténation *string1* + *string2* ne peut être calculée à la compilation. Donc, l'objet créé ne sera pas le même que l'objet correspondant au littéral "abcdef".

La condition

```
        System.out.println(string4 == "abcdef"); //false
```
retourne false car le constructeur *String*() produit toujours un objet distinct et donc différent de l'objet correspondant au littéral.

Les deux conditions dans

```
        System.out.println((string1    +    string2).intern()    ==
"abcdef"); //true
        System.out.println(string4.intern() == "abcdef"); //true
```
retournent true car la méthode *intern*() convertit la référence à une référence à l'objet correspondant au littéral "abcdef".

Les deux conditions dans

```
        System.out.println((string1  +  string2).equals("abcdef"));
//true
        System.out.println(string4.equals("abcdef")); //true
```
retournent *true* car ce n'est pas la référence à l'objet qui est comparée mais bien le contenu, c'est-à-dire la chaîne de caractère elle-même. La morale de cet exemple : il est habituellement préférable de comparer les **String** avec *equals*().

Un autre aspect qui porte souvent à confusion pour un novice est la distinction entre

- le littéral qui représente la chaîne vide ""
- le littéral *null*

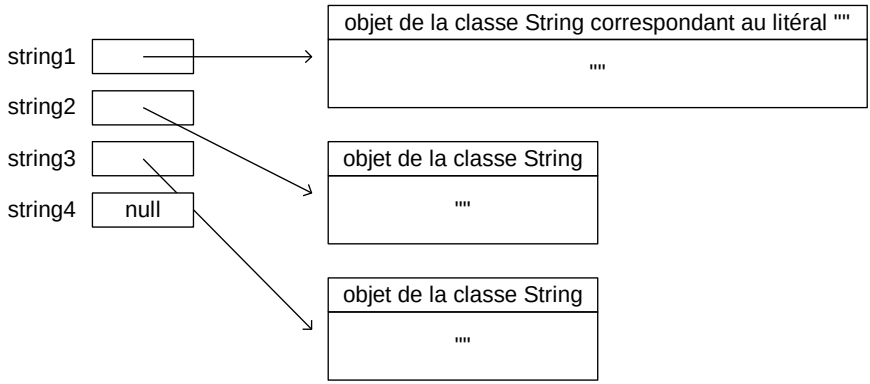
Exemple. JavaPasAPas/chapitre_3/ExemplesStringVide.java

L'exemple suivant illustre la différence entre ces concepts.

```
public class ExemplesStringVide{

  public static void main (String args[]) {

    String string1 = "";
    String string2 = new String("");
    String string3 = new String();
    String string4 = null;

    System.out.println(string1 == ""); // true
    System.out.println(string1.equals("")); //true
    System.out.println(string1 == null); //false

    System.out.println(string2 == ""); // false
```

```
    System.out.println(string2.equals("")); //true
    System.out.println(string2 == null); //false

    System.out.println(string3 == ""); // false
    System.out.println(string3.equals("")); //true
    System.out.println(string3 == null); //false

    System.out.println(string4 == ""); //false
    // System.out.println(string4.equals("")); provoquerait une exception à l'exécution
    System.out.println(string4 == null); //true
  }
}
```

La figure suivante montre l'effet du programme.



String1 fait référence à l'objet **String** qui correspond au littéral de la chaîne vide "". *String2* fait référence à un autre objet qui contient aussi la chaîne vide. Il en est de même pour *string3*. Ceci signifie que le constructeur de *String* sans paramètre *String*() initialise automatiquement son contenu à la chaîne vide. Enfin, *string4* contient la référence *null*. Le littéral spécial *null* signifie que la variable ne fait référence à aucun objet. L'expression *string4==null* permet de détecter cette situation. Lorsque l'objet fait référence à *null*, l'accès à son contenu provoque une exception Java à l'exécution. Ce serait le cas de l'énoncé suivant car la méthode *equals*() doit extraire le contenu de l'objet *string4* mais cet objet n'existe pas !

```
      // System.out.println(string4.equals("")); provoquerait une
exception à l'exécution
```

Enfin, notons que l'accès à une variable non initialisée provoque une erreur de compilation. Java vous protège contre l'accès aux variables non initialisée parce que de tels accès sont souvent la source de problèmes et de bogues.

Exemple. JavaPasAPas/chapitre_3/ExempleStringNonInitialise.java

En Java, nous distinguons la déclaration d'une variable (sans initialisation) et son initialisation (ou attribution de valeur). Avant de pouvoir utiliser une variable, celle-ci doit avoir été initialisée. Au sein d'une fonction, la simple déclaration d'une variable ne suffit par à l'initialiser et Java refusera l'accès à une variable non-initialisée.

```
public class ExempleStringNonInitialise{
   public static void main (String args[]) {
     String unString;
     System.out.println(unString == null); // erreur de compilation car non initialisé
   }
}
```

Exemple. JavaPasAPas/chapitre_3/ExempleEmoji.java

Java représente ses caractères sous le format UTF-16. La notion de caractère au sein d'une chaîne de caractères en Java présume que tous les caractères occupent 16 bits. Or les caractères des plans supplémentaires (comme les emojis) occupent 32 bits. Le code suivant va donc afficher 8 pour la longueur de la chaîne et va retourner un caractère invalide après l'appel à *charAt*. Le traitement des chaînes de caractères dans de tels cas peut se faire avec les méthodes *codePointAt* et *offsetByCodePoints*. Il s'agit d'un sujet avancé.

```
public class ExempleEmoji {
  public static void main(String[] args) {
    String s = "😂😍🎉👍";
    System.out.println(s.length());
    System.out.println(s.charAt(1));
  }
}
```

3.10 Fonctions mathématiques : java.lang.Math

Au-delà des opérations de base permises dans les expressions arithmétiques, le package *java.lang.Math* contient plusieurs méthodes pour le calcul de fonctions mathématiques courantes.

Exemple. JavaPasAPas/chapitre_3/ExemplesMath.java

L'exemple suivant montre quelques exemples de fonctions mathématiques usuelles.

```
public class ExemplesMath {

   public static void main (String args[]) {
     System.out.println("Math.log(1.0)="+Math.log(1.0));
     System.out.println("Math.exp(1.0)="+Math.exp(1.0));
     System.out.println("Math.cos(0)="+Math.cos(0));
     System.out.println("Math.sin(0)="+Math.sin(0));
     System.out.println("Math.sqrt(4)="+Math.sqrt(4));
   }
}
```

Résultat affiché :

```
Math.log(1.0)=0.0
Math.exp(1.0)=2.718281828459045
Math.cos(0)=1.0
Math.sin(0)=0.0
Math.sqrt(4)=2.0
```

Les classes *java.math.BigInteger* (pour les entiers) et *java.math.BigDecimal* (pour les nombres décimaux) permettent de traiter des nombres d'une précision plus grande que ce qui est permis avec les types primitifs.

3.11 Sommaire des opérations et priorités

Le tableau suivant montre la liste des opérations pour les expressions Java en ordre décroissant de priorité.

Opérateur	
++ --	Post-incrémentation Post-décrémentation
++ -- + - ! ~ (type)	Pré-incrémentation Pré-décrémentation + unaire - unaire négation logique complément (niveau bit) conversion de type
* / %	Multiplication binaire Division Reste après division entière
+ -	Addition binaire Soustraction binaire
<< >>	Décalage à gauche (niveau bit) Décalage à droite (niveau bit)

>>>	Décalage à droite sans signe (niveau bit)
< > <= >= instanceof	Plus petit que Plus grand que Plus petit ou égal à Plus grand ou égal à Est une instance de
== !=	Est égal à Est différent de
&	Et (niveau bit / logique)
^	Ou exclusif (niveau bit / logique)
|	Ou inclusif (niveau bit / logique)
&&	Et logique
||	Ou logique
?:	Expression conditionnelle
= += -= /= %= ^= |= <<= >>= >>>=	Affectation Auto-addition Auto-soustraction Auto-division Auto-reste Auto-ou-exclusif (niveau bit) Auto-ou (niveau bit) Auto-décalage à gauche (niveau bit) Auto-décalage à droite (niveau bit) Auto-décalage à droite sans signe (niveau bit)

4. Graphisme 2D et concepts de programmation objet

Ce chapitre examine les principaux mécanismes de dessins en deux dimensions (2D) de Java et approfondit les concepts de programmation objet suivants : sous-classe, interface, variables de classe et d'objet, méthode de classe et d'objet.

4.1 Dessin avec les classes Graphics et une sous-classe de JFrame

Étudions d'abord un exemple de programme qui dessine un bonhomme simple dans une fenêtre.

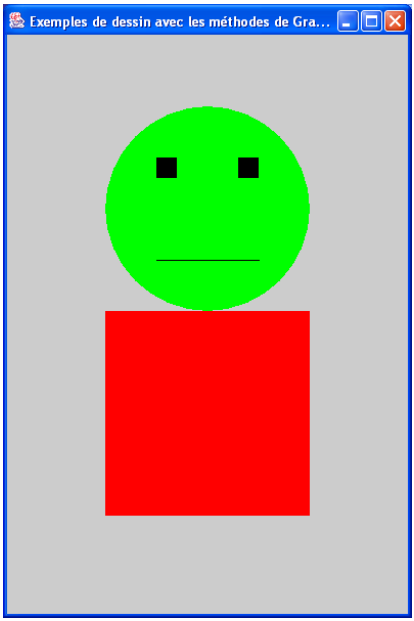
Exemple. JavaPasAPas/chapitre_4/ExempleDessin2DDansJFrame.java

Le programme *ExempleDessin2DDansJFrame.java* dessine un bonhomme simple (appelons-le Bot) dans une fenêtre produite par la classe *javax.swing*.**JFrame**.

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


public class ExempleDessin2DDansJFrame extends JFrame {

  public ExempleDessin2DDansJFrame() {
    super("Exemples de dessin avec les méthodes de Graphics");
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    this.setSize(400, 600);
    this.setVisible(true);
  }


  // La méthode paint() est appelée automatiquement lors de la création
  // du JFrame
  // La méthode paint() fait un dessin d'un bonhomme
  public void paint(Graphics g) {


    // Il faut appeler la méthode paint() de la super-classe
    super.paint(g);

    g.setColor(Color.green);
    g.fillOval(100, 100, 200, 200); // La tête

    g.setColor(Color.black);
    g.fillRect(150, 150, 20, 20); // L'oeil gauche
    g.fillRect(230, 150, 20, 20); // L'oeil droit
    g.drawLine(150, 250, 250, 250); // La bouche
```

```
  g.setColor(Color.red);
  g.fillRect(100, 300, 200, 200); // Le corps
 }

 public static void main(String args[]) {
  new ExempleDessin2DDansJFrame();
 }
}
```

Voici le résultat de l'exécution du programme :



Le dessin est effectué dans une fenêtre graphique qui correspond à un espace à deux dimensions (2D) illustré à la figure suivante. L'axe des *x* est l'axe horizontal et l'axe des *y*, le vertical. Par opposition à la convention mathématique usuelle, l'axe des *y* est orienté vers le bas. Les figures graphiques font référence aux coordonnées de cet espace.

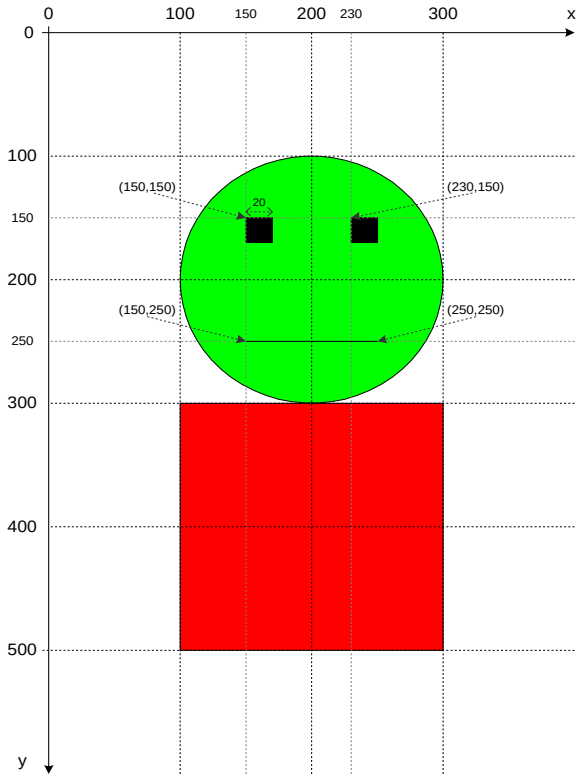


Figure 17. Coordonnées du Bonhomme.

Certains aspects sembleront flous à ce point-ci et seront détaillés par la suite. La clause *import java.awt.** apparaît en début de programme étant donné que les classes *java.awt.Graphics* et *java.awt.Color* sont utilisées. Le * permet d'importer toutes les classes du package *java.awt* sans devoir spécifier chacune des classes individuellement.

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

La méthode *main*() ne fait que créer un objet de la classe *ExempleDessin2DDansJFrame* par :

```
        new ExempleDessin2DDansJFrame();
```

Cet objet représente la fenêtre dans laquelle est effectué le dessin.

- **Notion de sous-classe**

La classe *ExempleDessin2DDansJFrame* est une sous-classe de la classe **JFrame**. Ceci est exprimé par la clause *extends JFrame* dans la ligne suivante.

```
public class ExempleDessin2DDansJFrame extends JFrame {
```

Inversement, on dit que *JFrame* est la super-classe de *ExempleDessin2DDansJFrame*. La figure suivante montre un diagramme de classe UML qui illustre ce concept. Un tel diagramme est utile pour comprendre l'organisation des classes d'un programme Java. Chacune des classes est représentée par un rectangle. Le nom de la classe apparaît dans la partie supérieure, et les méthodes de la classe dans la partie inférieure. La flèche représente une relation de sous-classe (aussi appelée relation d'*héritage* ou encore de *généralisation/spécialisation*). À noter que le diagramme est partiel. La classe **JFrame** contient beaucoup plus de méthodes et elle est elle-même sous-classe d'une autre classe.

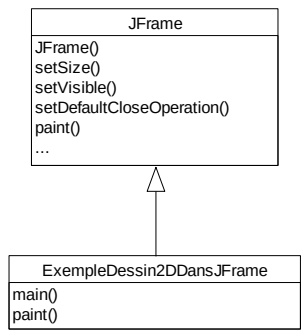


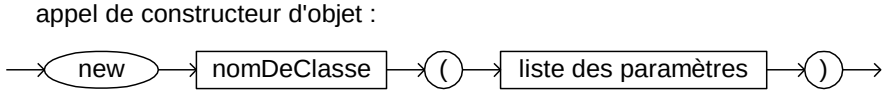
Figure 18. Représentation d'une sous-classe en UML.

En définissant une classe X comme une sous-classe d'une autre classe Y, la classe X hérite de toutes les méthodes définies dans la classe Y. Ainsi, en définissant la classe *ExempleDessin2DDansJFrame* comme une sous-classe de **JFrame** à l'aide de l'utilisation de la directive *extends*, la classe *ExempleDessin2DDansJFrame* hérite d'un ensemble de méthodes définies dans la classe *JFrame* pour la manipulation des fenêtres. Ceci entraîne qu'un objet de la classe *ExempleDessin2DDansJFrame* est aussi considéré comme un objet de la super-classe **JFrame**. Il s'agit d'un exemple de ce que nous

appelons parfois du *polymorphisme* : une instance de la classe *ExempleDessin2DDansJFrame* est aussi une instance de la classe **JFrame** ce qui implique qu'une classe peut prendre plusieurs formes (littéralement : poly [plusieurs] morph [forme]). Une sous-classe est parfois appelée une « classe héritée » ou une « classe dérivée ».

Dans le diagramme, on ne répète pas les méthodes héritées dans la sous-classe. Ceci est implicite. Les méthodes d'objet héritées de la classe **JFrame** peuvent être appelées sur un objet de la classe *ExempleDessin2DDansJFrame* comme si elles y avaient été définies. Le mécanisme d'héritage permet ainsi de réutiliser les méthodes de la classe **JFrame** dans la classe *ExempleDessin2DDansJFrame* sans avoir à les répéter. En particulier, dans notre exemple, les méthodes *setDefaultCloseOperation*(), *setSize*(), *setVisible*(), *paint*() sont héritées de la classe **JFrame** mais elles sont utilisées sur un objet de la sous-classe *ExempleDessin2DDansJFrame* dans notre exemple. Cette manière de programmer en créant une sous-classe d'une classe existante en réutilisant le code d'une classe existante tout en ajoutant un comportement plus spécialisé est une approche typique et très puissante de la programmation objet. Dans notre exemple, ceci nous permet de créer des fenêtres contenant des dessins d'une manière très simple sans avoir à en programmer tous les détails.

Rappelons la syntaxe de la création d'un objet :



Ainsi, le *new ExempleDessin2DDansJFrame*() dans la méthode *main*() crée un objet de la classe *ExempleDessin2DDansJFrame*. Concrètement, le *new* appelle une méthode dite *constructeur* d'objet qui doit porter le même nom que celui de la classe.

Dans notre exemple, le *new ExempleDessin2DDansJFrame*() provoque l'appel de la méthode *ExempleDessin2DDansJFrame*() qui est une méthode *constructeur d'objet* de la classe *ExempleDessin2DDansJFrame*. Le corps de la méthode constructeur est utilisé pour initialiser certains aspects d'un objet au moment de sa création.

La ligne suivante déclare le constructeur d'objet :

```
    public ExempleDessin2DDansJFrame() {
```

La ligne suivante spécifie un titre qui apparaît dans le haut de la fenêtre.

```
        super("Exemples de méthodes de Graphics dans un JFrame");
```

L'identificateur réservé *super* signifie d'appeler la méthode constructeur correspondante de la super-classe **JFrame**. La méthode correspondante est le constructeur de la super-classe qui a les mêmes paramètres. Il y a donc, dans la classe **JFrame**, une méthode constructeur qui prend un titre (**String**) en paramètre. Cette manière d'appeler le constructeur d'une super-classe permet, dans une sous-classe, de compléter le travail du constructeur de la super-classe en ajoutant des aspects particuliers à la sous-classe.

La ligne suivante spécifie qu'il faut terminer le programme (provoque un appel de *System.exit*(0)) lorsque l'utilisateur clique dans le X dans le coin supérieur droit de la fenêtre. Ceci est spécifié par l'appel de méthode d'objet **setDefaultCloseOperation**(int operation) de la classe **JFrame**.

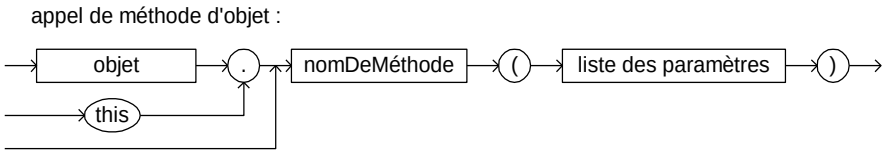
```
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);
```

Identificateur réservé *this*

L'identificateur réservé *this* représente une référence à l'objet « *ceci* » qui est en processus de construction par le constructeur. Comme cet objet est en phase de construction, il ne peut y avoir une variable qui contient cet objet pour y faire référence. Le *this* est un identificateur réservé qui permet de désigner l'objet en phase de construction.

Le constructeur ne construit qu'un objet à la fois et il n'y a donc pas d'ambigüité au sujet de l'objet représenté par *this*. Lorsqu'un appel de méthode est fait sans spécifier un objet en préfixe, c'est comme si on mettait le *this* en préfixe. Le *this* n'est donc pas obligatoire. Dans notre exemple, on pourrait remplacer *this.setDefaultCloseOperation*(EXIT_ON_CLOSE) par *setDefaultCloseOperation*(EXIT_ON_CLOSE). Cependant, dans certains cas, il faut mettre le *this* pour éviter certaines ambiguïtés. Nous y reviendrons.

Voici le diagramme de syntaxe d'un appel de méthode d'objet tenant compte de ces deux nouvelles possibilités. Rappelons qu'une méthode d'objet est une méthode qu'on appelle sur un objet.



La méthode *setSize*() spécifie la largeur (400) et la hauteur (600) de la fenêtre :

```
        this.setSize(400,600);
```

La méthode *setVisible*() rend la fenêtre visible. Ceci provoque indirectement l'appel de la méthode *paint*(*Graphics* g) qui effectue le dessin.

```
        this.setVisible(true);
```

Le dessin du *Bot* est alors effectué par la méthode *paint*().

La méthode *paint*() et le multifenêtrage

L'appel à *paint*() n'est pas visible dans notre programme. Cet appel se passe au niveau de la super-classe **JFrame**. Le fait de ne pas voir les détails de cet appel est un aspect un peu surprenant de la programmation Java, surtout pour un novice. En fait, la méthode *paint*() est appelée automatiquement à chaque fois qu'il faut redessiner le contenu de la fenêtre. Ceci est nécessaire lorsqu'une partie cachée de la fenêtre est rendue visible au premier plan lors de la manipulation des fenêtres par l'utilisateur. Plusieurs opérations de manipulation de fenêtre conduisent à cette circonstance.

Donc, à chaque fois qu'on utilise la classe **JFrame**, la méthode *paint*() est employée pour afficher quelque chose à l'intérieur de la fenêtre. D'autres façons de procéder seront vues plus loin.

Regardons maintenant le détail de la méthode *paint*() qui effectue le dessin. La méthode est déclarée par :

```
    public void paint (Graphics g) {
```

Dans notre code, cette méthode est une méthode d'objet de la classe *ExempleDessin2DDansJFrame*.

Rappelons le sens des identificateurs réservés *public void* :

- *public* signifie que la méthode peut être appelée de partout
- *void* signifie que la méthode ne retourne rien

L'absence de l'identificateur réservé *static* signifie que c'est une méthode d'objet.

Cette méthode existe déjà dans la super-classe ! Cependant, la méthode de la super-classe doit être redéfinie pour effectuer le dessin. Cette manière d'adapter une classe en définissant une sous-classe qui redéfinit quelques méthodes de la super-classe afin de spécialiser le comportement de la sous-classe est typique de la programmation objet.

La méthode *paint*() a un paramètre appelé *g* dont le type est la classe *java.awt.***Graphics**. Rappelons que le nom de paramètre est précisé par le programmeur et n'a pas d'importance d'un point de vue de l'exécution du programme. Ce nom de paramètre est utilisé dans le corps de la méthode pour désigner le paramètre. Le type du paramètre (**Graphics**) représente un espace 2D de dessin qui fait partie de la fenêtre. Un tel objet de la classe **Graphics** est appelé un *contexte graphique*. Les opérations de dessin dans la méthode *paint*() seront effectuées en appelant des méthodes d'objet de l'objet *g*. Lorsque la méthode *paint*() est appelée (on ne voit pas l'appel ici), l'objet **Graphics** de la fenêtre est passé en paramètre à la méthode *paint*(). Comme on ne voit pas l'appel à la méthode *paint*(), le programmeur curieux peut se sentir dans un état un peu flou … Comment est créé l'objet **Graphics** ? Qui appelle *paint*() ?

La méthode *paint*() de la classe *ExempleDessin2DDansJFrame* appelle tout d'abord la méthode *paint*() de la super-classe **JFrame**. Notez la syntaxe spéciale *super.paint*(). Par convention Java, l'usage du préfixe *super* désigne la méthode de même nom et paramètres de la super-classe. L'appel à la méthode *paint*() de la super-classe est une convention préétablie des classes d'interface graphique de Java, car la méthode *paint*() de la super-classe

effectue certaines opérations nécessaires au bon fonctionnement de fenêtres à structure complexe tel qu'une fenêtre **JFrame**.

```
        // Il faut appeler la méthode paint() de la super-classe
        super.paint(g);
```

Redéfinition d'une méthode par spécialisation et surcharge dynamique

Après avoir appelé le *paint*() de la super-classe, le *paint*() de la sous-classe ajoute les opérations de dessin. De ce point de vue la méthode *paint*() de la sous-classe est une spécialisation de la méthode *paint*() de la super-classe. On dit que la méthode *paint*() de la sous-classe *ExempleDessin2DDansJFrame* est une redéfinition de la méthode *paint*() de la super-classe **JFrame** par spécialisation de la méthode de la super-classe. Le résultat est qu'il y a plusieurs méthodes *paint*() avec le même nom et les mêmes paramètres ! Comment savoir quelle méthode doit être appelée ? Java détermine la méthode appropriée en fonction du type de l'objet qui est désigné pour l'appel de la méthode. Si c'est un objet de la classe **JFrame**, c'est le *paint*() de **JFrame** qui est appelé. Si c'est un objet de la classe *ExempleDessin2DDansJFrame* , c'est le *paint*() de *ExempleDessin2DDansJFrame* qui est appelé. Ce principe est appelé *surcharge dynamique* (ou encore *polymorphisme dynamique*) *d'un nom de méthode.*

Identificateur réservé *super*

Lorsqu'une sous-classe redéfinie une méthode d'une super-classe, la syntaxe *super.nomMéthode*() désigne l'appel de la méthode *nomMéthode*() de la super-classe. Dans le cas d'un constructeur, il suffit d'utiliser *super* tout court pour désigner le constructeur correspondant de la super-classe, c'est-à-dire celui qui a les mêmes paramètres.

Voyons maintenant les méthodes de dessin. La ligne suivante spécifie que la couleur courante de dessin sera le vert.

```
g.setColor(Color.green);
```

La méthode **setColor**(**Color** c) est une méthode d'objet de la classe **Graphics** qui fixe la couleur utilisée pour les méthodes de dessin. Le paramètre *Color.green* est un objet de la classe **Color** qui correspond à une

constante de classe définie dans la classe **Color** pour représenter la couleur verte. La notion de constante de classe sera détaillée par la suite. La classe **Color** inclut plusieurs autres constantes pour définir les principales couleurs.

La ligne suivante dessine le cercle vert de la tête du bonhomme.

```
        g.fillOval(100,100,200,200); // La tête
```

La méthode **fillOval**(int x, int y, int width, int height) dessine un ovale plein inscrit dans un rectangle dont la coordonnée du coin supérieur droit est (*x*, *y*) la largeur est *width* et la hauteur est *height*. Comme la hauteur est égale à la largeur, ceci produit un cercle. La couleur employée est celle qui a été prédéterminée par l'appel précédent à *setColor*().

La ligne suivante établit la couleur noire comme couleur courante.

```
        g.setColor(Color.black);
```

La ligne suivante dessine un carré noir qui correspond l'œil gauche. Les paramètres de **fillRect**(int x, int y, int width, int height) ont la même signification que pour *fillOval*(). Comme la hauteur est égale à la largeur, ceci produit un carré.

```
        g.fillRect(150,150,20,20); // L'oeil gauche
```

La ligne suivante dessine l'œil droit.

```
        g.fillRect(230,150,20,20); // L'oeil droit
```

La ligne suivante dessine une ligne qui correspond à la bouche.

```
        g.drawLine(150,250,250,250); // La bouche
```

La méthode **drawLine**(int x1, int y1, int x2, int y2) dessine une ligne qui part de la coordonnées (*x1*, *y1*) et se termine à la coordonnée (*x2*, *y2*).

Ensuite, les deux lignes suivantes dessinent le rectangle rouge du corps.

```
        g.setColor(Color.red);
        g.fillRect(100,300,200,200); // Le corps
```

La classe **Graphics** contient plusieurs autres méthodes de dessin de formes de base.

Exercice. En vous inspirant de l'exemple précédent, écrivez un programme qui dessine un bonhomme de votre cru ou encore le bonhomme *Iti* suivant. La taille de la fenêtre de *Iti* est 300 par 300.

Solution. JavaPasAPas/chapitre_4/ExerciceDessinIti.java

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

public class ExerciceDessinIti extends JFrame {

  public ExerciceDessinIti() {
    super("Dessin de Iti");
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    this.setSize(300, 300);
    this.setVisible(true);
  }

  public void paint(Graphics g) {
    super.paint(g);
    // Dessin de Iti
    // La tête
    g.setColor(Color.pink);
    g.fillOval(133, 50, 33, 50);

    // Le sourire
    g.setColor(Color.black);
    g.drawArc(133, 34, 33, 50, -125, 70);
    // Les yeux
    g.fillOval(138, 66, 8, 8);
    g.fillOval(154, 66, 8, 8);
```

```
  // Le corps
  g.drawLine(150, 100, 150, 200);
  // Les bras
  g.drawLine(100, 100, 150, 150);
  g.drawLine(200, 100, 150, 150);
  // Les jambes
  g.drawLine(100, 250, 150, 200);
  g.drawLine(200, 250, 150, 200);
 }

 public static void main(String args[]) {
  new ExerciceDessinIti();
 }
}
```

- **Création de plusieurs objets (fenêtres de dessin)**

L'exemple suivant créé trois fenêtres qui correspondent à trois objets de la classe *ExempleDessin2DDansJFrame*.

Exemple. Création de trois fenêtres de dessin.

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

public class ExempleDessin2DDansJFrame extends JFrame {

   public ExempleDessin2DDansJFrame() {
      super("Exemples de dessin avec les méthodes de Graphics");
      this.setDefaultCloseOperation(EXIT_ON_CLOSE);
      this.setSize(400,600);
      this.setVisible(true);
   }

   // La méthode paint() est appelée automatiquement lors de la création du JFrame
   // La méthode paint() fait un dessin d'un bonhomme
   public void paint (Graphics g) {
      super.paint(g);

      g.setColor(Color.green);
      g.fillOval(100,100,200,200); // La tête
      g.setColor(Color.black);
      g.fillRect(150,150,20,20); // L'oeil gauche
      g.fillRect(230,150,20,20); // L'oeil droit
      g.drawLine(150,250,250,250); // La bouche
      g.setColor(Color.red);
      g.fillRect(100,300,200,200); // Le corps
   }

   public static void main (String args[]) {
      new ExempleDessin2DDansJFrame();
      new ExempleDessin2DDansJFrame();
      new ExempleDessin2DDansJFrame();
   }
}
```

Chaque appel à *new ExempleDessin2DDansJFrame*() crée une nouvelle fenêtre.

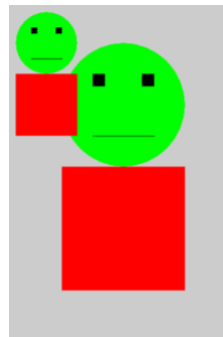
NB Si vous exécutez ce programme, les fenêtres sont superposées. Il faut les déplacer pour voir les trois.

4.2 Simplification du programme par une méthode avec paramètres

Cette section approfondit la création de méthode et l'utilisation des paramètres en montrant comment l'utilisation d'une méthode avec paramètre peut grandement simplifier un programme et ceci dans le contexte du dessin 2D. La notion de passage des paramètres sera étudiée en même temps.

L' exercice suivant permet de motiver l'utilisation d'une méthode avec paramètres.

Exercice. Dessiner deux bonhommes de taille et position différentes tel qu'illustré par la figure suivante :



Solution. JavaPasAPas/chapitre_4/Exercice2Bots.java

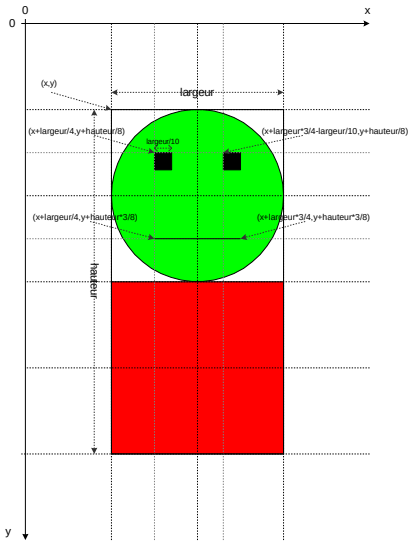
```
import java.awt.*;
import javax.swing.*;

public class Exercice2Bots extends JFrame {
  public Exercice2Bots() {
    super("Dessiner deux Bots");
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    this.setSize(400, 600);
    this.setVisible(true);
  }
```

```
public void paint(Graphics g) {
  super.paint(g);
  // Le premier Bot
  g.setColor(Color.green);
  g.fillOval(100, 100, 200, 200); // La tête

  g.setColor(Color.black);
  g.fillRect(150, 150, 20, 20); // L'oeil gauche
  g.fillRect(230, 150, 20, 20); // L'oeil droit
  g.drawLine(150, 250, 250, 250); // La bouche

  g.setColor(Color.red);
  g.fillRect(100, 300, 200, 200); // Le corps

  // Le deuxième Bot
  g.setColor(Color.green);
  g.fillOval(25, 50, 100, 100); // La tête

  g.setColor(Color.black);
  g.fillRect(50, 75, 10, 10); // L'oeil gauche
  g.fillRect(90, 75, 10, 10); // L'oeil droit
  g.drawLine(50, 125, 100, 125); // La bouche

  g.setColor(Color.red);
  g.fillRect(25, 150, 100, 100); // Le corps
 }

 public static void main(String args[]) {
  new Exercice2Bots();
 }
}
```

La solution précédente rappelle les mêmes méthodes deux fois. Pour la position et la taille du *Bot*, il faut calculer de nouvelles valeurs des paramètres à chaque fois. Une manière plus élégante de traiter ce problème consiste à chercher une solution plus générale au dessin d'un *Bot* où sa position et sa taille sont variables. Une technique souvent employée en graphisme 2D consiste à définir un rectangle englobant à l'intérieur duquel sera dessiné le *Bot* tel qu'illustré à la figure suivante. Le *Bot* est dessiné à l'échelle à l'intérieur du rectangle. Comme pour la méthode *fillRect*(), quatre variables sont définies

pour représenter le rectangle englobant : les coordonnées *x* et *y* du coin inférieur droit, la *largeur* et la *hauteur* du rectangle englobant.

Exemple.



JavaPasAPas/chapitre_4/ExempleBotRectangleEnglobant.java

La version suivante dessine le même *Bot* que notre premier exemple mais en utilisant des variables qui représentent le rectangle englobant.

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

public class ExempleBotRectangleEnglobant extends JFrame {

  public ExempleBotRectangleEnglobant() {
    super("Bot avec rectangle englobant");
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    this.setSize(400, 600);
    this.setVisible(true);
  }

  public void paint(Graphics g) {
    super.paint(g);

    int x = 100;
    int y = 100;
    int largeur = 200;
    int hauteur = 400;
```

```
    // Bonhomme à l'échelle dans un rectangle englobant défini
    // par x,y,largeur,hauteur
    g.setColor(Color.green);
    g.fillOval(x, y, largeur, hauteur / 2); // La tête

    g.setColor(Color.black);
    g.fillRect(x + largeur / 4, y + hauteur / 8, largeur / 10, hauteur / 20); // L'oeil gauche
    g.fillRect(
        x + largeur * 3 / 4 - largeur / 10,
        y + hauteur / 8,
        largeur / 10,
        hauteur / 20); // L'oeil droit
    g.drawLine(
        x + largeur / 4,
        y + hauteur * 3 / 8,
        x + largeur * 3 / 4,
        y + hauteur * 3 / 8); // La bouche

    g.setColor(Color.red);
    g.fillRect(x, y + hauteur / 2, largeur, hauteur / 2); // Le corps
  }

  public static void main(String args[]) {
    new ExempleBotRectangleEnglobant();
  }
}
```

Exemple. JavaPasAPas/chapiter_5/ Exemple2BotsRectangleEnglobant.java

On peut maintenant facilement redessiner le *Bot* deux fois en changeant la position et la taille par la modification des valeurs des variables *x*, *y*, *largeur* et *hauteur* mais en répétant exactement les mêmes instructions deux fois.

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

public class ExempleBotRectangleEnglobant extends JFrame {

  public ExempleBotRectangleEnglobant() {
    super("Bot avec rectangle englobant");
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    this.setSize(400, 600);
    this.setVisible(true);
  }
```

```
 public void paint(Graphics g) {
  super.paint(g);

  int x = 100;
  int y = 100;
  int largeur = 200;
  int hauteur = 400;

  // Bonhomme à l'échelle dans un rectangle englobant défini
  // par x,y,largeur,hauteur
  g.setColor(Color.green);
  g.fillOval(x, y, largeur, hauteur / 2); // La tête

  g.setColor(Color.black);
  g.fillRect(x + largeur / 4, y + hauteur / 8, largeur / 10, hauteur / 20); // L'oeil gauche
  g.fillRect(
     x + largeur * 3 / 4 - largeur / 10,
     y + hauteur / 8,
     largeur / 10,
     hauteur / 20); // L'oeil droit
  g.drawLine(
     x + largeur / 4,
     y + hauteur * 3 / 8,
     x + largeur * 3 / 4,
     y + hauteur * 3 / 8); // La bouche

  g.setColor(Color.red);
  g.fillRect(x, y + hauteur / 2, largeur, hauteur / 2); // Le corps
 }

 public static void main(String args[]) {
  new ExempleBotRectangleEnglobant();
 }
}
```

Cette solution oblige de répéter deux fois les mêmes énoncés. On peut éviter cette répétition en les regroupant dans une méthode et en appelant cette méthode à deux reprises.

Exemple. JavaPasAPas/chapitre_4/ExempleMethodePaintBot.java

Dans l'exemple suivant la méthode *paintBot*(Graphics g, int x, int y, int largeur, int hauteur) regroupe les énoncés de dessin du *Bot*. Plutôt que de répéter ces énoncés deux fois, il suffit d'appeler la méthode deux fois !

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

public class ExempleMethodePaintBot extends JFrame {

 public ExempleMethodePaintBot() {
  super("2 Bots avec méthode paintBot()");
  this.setDefaultCloseOperation(EXIT_ON_CLOSE);
  this.setSize(400, 600);
  this.setVisible(true);
 }

 // Méthode qui dessine un Bot dans un objet Graphics g
 // à l'échelle dans un rectangle englobant de paramètres
 // x,y,largeur,hauteur
 public static void paintBot(Graphics g, int x, int y, int largeur, int hauteur) {
  g.setColor(Color.green);
  g.fillOval(x, y, largeur, hauteur / 2); // La tête
  g.setColor(Color.black);
  g.fillRect(x + largeur / 4, y + hauteur / 8, largeur / 10, hauteur / 20); // L'oeil gauche
  g.fillRect(
     x + largeur * 3 / 4 - largeur / 10,
     y + hauteur / 8,
     largeur / 10,
     hauteur / 20); // L'oeil droit
  g.drawLine(
     x + largeur / 4,
     y + hauteur * 3 / 8,
     x + largeur * 3 / 4,
     y + hauteur * 3 / 8); // La bouche

  g.setColor(Color.red);
  g.fillRect(x, y + hauteur / 2, largeur, hauteur / 2); // Le corps
 }

 public void paint(Graphics g) {
  super.paint(g);
  // Dessin du premier Bot
  paintBot(g, 100, 100, 200, 400);
  // Dessin du deuxième Bot
  paintBot(g, 25, 50, 100, 200);
 }

 public static void main(String args[]) {
  new ExempleMethodePaintBot();
 }
}
```

La ligne suivante dans le code de l'exemple est une déclaration de la *signature* de la méthode *paintBot*(). Cette déclaration est similaire à celle de la méthode *main*() :

```
    public static void paintBot (Graphics g, int x, int y, int
largeur, int hauteur) {
```

Signature d'une méthode

La signature d'une méthode précise :

- le nom de la méthode

- la forme des paramètres

- la forme du résultat

La signature d'une méthode permet de déterminer comment appeler la méthode.

La méthode *paintBot*() fait partie de la classe *ExempleMethodePaintBot* car elle est déclarée dans le corps de la classe, c'est-à-dire entre les accolades qui suivent le nom de la classe. Dans cet exemple, on aurait aussi bien pu en faire une méthode d'objet en omettant le *static*. Nous reviendrons sur cet aspect.

La liste des paramètres entre virgules spécifie le type et le nom de chacun des paramètres. Le programmeur décide des noms, types et nombre des paramètres. Cette liste de paramètres qui apparaît dans la déclaration de la signature de la méthode représente les paramètres dits *formels*. D'autre part, lorsqu'on appelle la méthode, on doit spécifier les paramètres dits *réels*.

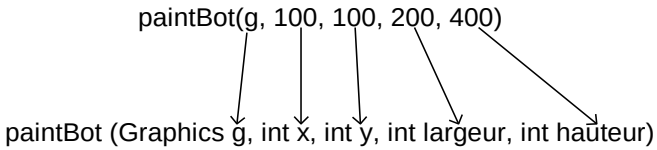
Dans le corps de la méthode, les énoncés font référence aux noms des paramètres formels comme s'ils étaient des variables. Par exemple, dans la ligne suivante, les variables *x*, *y*, *largeur* et *hauteur* sont les paramètres formels de la méthode *paintBot*() :

```
        g.fillOval(x,y,largeur,hauteur/2); // La tête
```

Pour dessiner les deux *Bot*, la méthode *paint*() appelle deux fois la méthode *paintBot*() en précisant les paramètres réels. La ligne suivante est le premier appel.

```
        paintBot(g, 100, 100, 200, 400);
```

Les paramètres g, 100, 100, 200, 400 sont les paramètres réels. Il est à noter qu'un paramètre réel n'est pas nécessairement une constante. Ce peut aussi être un nom de variable, de paramètre, et même une expression. Lors de l'appel de la méthode, on peut imaginer que la valeur du paramètre réel est en quelque sorte affectée à la variable qui correspond au paramètre formel. Ensuite, la méthode est exécutée.



On obtient ainsi exactement le même effet que si l'on avait effectué la séquence suivante qui dessine le premier Bot dans *Exemple2BotsRectangleEnglobant* :

```
int x = 100;
int y = 100;
int largeur = 200;
int hauteur = 400;
// Bonhomme à l'échelle dans un rectangle englobant défini par x,y,largeur,hauteur
g.setColor(Color.green);
g.fillOval(x,y,largeur,hauteur/2); // La tête
g.setColor(Color.black);
g.fillRect(x+largeur/4,y+hauteur/8,largeur/10,hauteur/20); // L'oeil gauche
g.fillRect(x+largeur*3/4-largeur/10,y+hauteur/8,largeur/10,hauteur/20); // L'oeil droit
g.drawLine(x+largeur/4,y+hauteur*3/8,x+largeur*3/4,y+hauteur*3/8); // La bouche
g.setColor(Color.red);
g.fillRect(x,y+hauteur/2,largeur,hauteur/2); // Le corps
```

Passage de paramètre par valeur ou par référence

Le fait de copier la valeur du paramètre réel en l'affectant au paramètre formel correspond au concept de passage de paramètre par valeur. Ainsi, dans la méthode, si la valeur du paramètre formel est modifiée, ceci n'affecte pas le paramètre réel. En Java, c'est la seule manière de passer un paramètre lorsqu'il s'agit d'un type primitif (int, double, etc.). D'autres langages offrent une autre option : le passage de paramètre par référence qui permet de modifier la valeur du paramètre réel. Dans le cas d'un paramètre objet en Java, la valeur passée est la référence à l'objet.

Pour le paramètre *g*, le contexte graphique (objet de la classe **Graphics**) de *paint*() est tout simplement passé à la méthode *paintBot*() qui va donc dessiner

sur le même contexte graphique que si le dessin avait été effectué directement dans *paint*(). À noter que le nom du paramètre réel *g* est ici le même que celui du paramètre formel. Ceci n'est pas obligatoire. Par exemple, on aurait pu définir la méthode *paintBot*() de la manière suivante sans que cela n'ait d'effet sur le comportement du programme.

```
public static void paintBot (Graphics unGraphics, int x, int y, int largeur, int hauteur) {
   unGraphics.setColor(Color.green);
   unGraphics.fillOval(x,y,largeur,hauteur/2); // La tête
   unGraphics.setColor(Color.black);
   unGraphics.fillRect(x+largeur/4,y+hauteur/8,largeur/10,hauteur/20); // L'oeil gauche
   unGraphics.fillRect(x+largeur*3/4-largeur/10,y+hauteur/8,largeur/10,hauteur/20); // L'oeil droit
   unGraphics.drawLine(x+largeur/4,y+hauteur*3/8,x+largeur*3/4,y+hauteur*3/8); // La bouche
   unGraphics.setColor(Color.red);
   unGraphics.fillRect(x,y+hauteur/2,largeur,hauteur/2); // Le corps
}
```

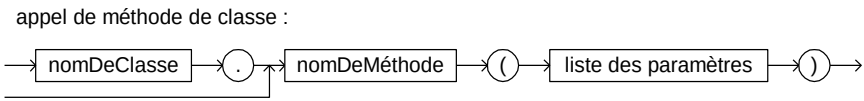
Le nom en soi du paramètre formel n'a pas d'importance. L'important c'est de respecter l'ordre et le type des paramètres lors de l'appel.

Même si le passage de paramètre est par valeur, c'est-à-dire qu'une copie de la référence à l'objet graphique *g* est passée à la méthode, la méthode peut modifier le contenu de l'objet en appelant les méthodes de modification de l'objet.

Raccourci pour l'appel d'une méthode de la même classe

La méthode *paintBot*() est une méthode de classe. Rappelons que l'appel d'une méthode de classe débute normalement par le nom de la classe. Dans le cas présent, il n'est pas nécessaire de mettre le nom de classe car la méthode appelée fait partie de la même classe que la méthode appelante. Ce raccourci est permis lorsqu'une méthode en appelle une autre de la même classe.

En considérant cette nouvelle possibilité, la syntaxe d'un appel de méthode de classe est donc :



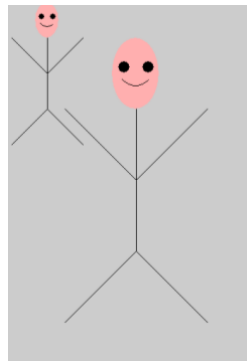
Exercice. Reprenez l'exemple de méthode avec votre bonhomme (ou le *Iti*). Définissez une méthode *paintXXX* (*Graphics g*, *int x*, *int y*, *int largeur*, *int hauteur*) qui dessine votre bonhomme et appelez-la à deux reprises avec des valeurs différentes pour les paramètres.

Solution avec Iti :

JavaPasAPas/chapitre_4/ExerciceMethodePaintIti.java

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

public class ExerciceMethodePaintIti extends JFrame {

 public ExerciceMethodePaintIti() {
  super("2 Itis avec méthode paintIti()");
  this.setDefaultCloseOperation(EXIT_ON_CLOSE);
  this.setSize(400, 600);
  this.setVisible(true);
 }

 // Méthode qui dessine un Iti dans un objet Graphics g
 // à l'échelle dans un rectangle englobant de paramètres x,y,largeur,hauteur
 public static void paintIti(Graphics g, int x, int y, int largeur, int hauteur) {
  // Coordonnées du milieu du rectangle englobant pour faciliter les calculs
  int milieux = x + largeur / 2;
  int milieuy = y + hauteur / 2;
  g.setColor(Color.pink); // La tête
  g.fillOval(x + largeur / 3, y, largeur / 3, hauteur / 4);
  g.setColor(Color.black); // Le sourire
  g.drawArc(x + largeur / 3, y - hauteur / 12, largeur / 3, hauteur / 4, -125, 70);
  g.fillOval(milieux - largeur / 8, y + hauteur / 12, largeur / 12, hauteur / 24); // Les yeux
  g.fillOval(milieux + largeur / 8 - largeur / 12, y + hauteur / 12, largeur / 12, hauteur / 24);
  g.drawLine(milieux, y + hauteur / 4, milieux, y + hauteur * 3 / 4); // Le corps
  g.drawLine(x, y + hauteur / 4, milieux, milieuy); // Les bras
  g.drawLine(x + largeur, y + hauteur / 4, milieux, milieuy);
  g.drawLine(x, y + hauteur, milieux, y + hauteur * 3 / 4); // Les jambes
  g.drawLine(x + largeur, y + hauteur, milieux, y + hauteur * 3 / 4);
 }

 public void paint(Graphics g) {
  super.paint(g);
  // Dessin du premier Bot
  paintIti(g, 100, 100, 200, 400);
  // Dessin du deuxième Bot
  paintIti(g, 25, 50, 100, 200);
 }

 public static void main(String args[]) {
  new ExerciceMethodePaintIti();
 }
}
```

Résultat :



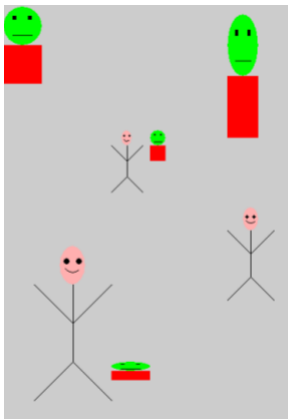
Exercice. Dessiner plusieurs bonhommes *Bot* et *Iti* dans la même fenêtre.

Solution. JavaPasAPas/chapitre_4/ExercicePlusieursBotEtIti.java

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

public class ExercicePlusieursBotEtIti extends JFrame {

  public ExercicePlusieursBotEtIti() {
    super("Rassemblement de Bots et Itis");
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    this.setSize(400, 600);
    this.setVisible(true);
  }

  // Méthode qui dessine un Bot dans un objet Graphics g
  // à l'échelle dans un rectangle englobant de paramètres x,y,largeur,hauteur
  public static void paintBot(Graphics g, int x, int y, int largeur, int hauteur) {
    g.setColor(Color.green);
    g.fillOval(x, y, largeur, hauteur / 2); // La tête

    g.setColor(Color.black);
    g.fillRect(x + largeur / 4, y + hauteur / 8, largeur / 10, hauteur / 20); // L'oeil gauche
    g.fillRect(
        x + largeur * 3 / 4 - largeur / 10,
        y + hauteur / 8,
        largeur / 10,
        hauteur / 20); // L'oeil droit
    g.drawLine(
        x + largeur / 4,
        y + hauteur * 3 / 8,
        x + largeur * 3 / 4,
        y + hauteur * 3 / 8); // La bouche

    g.setColor(Color.red);
    g.fillRect(x, y + hauteur / 2, largeur, hauteur / 2); // Le corps
  }

  // Méthode qui dessine un Iti dans un objet Graphics g
  // à l'échelle dans un rectangle englobant de paramètres x,y,largeur,hauteur
  public static void paintIti(Graphics g, int x, int y, int largeur, int hauteur) {
    // Coordonn�es du milieu du rectangle englobant pour faciliter les calculs
```

```
  int milieux = x + largeur / 2;
  int milieuy = y + hauteur / 2;
  // La tête
  g.setColor(Color.pink);
  g.fillOval(x + largeur / 3, y, largeur / 3, hauteur / 4);
  // Le sourire
  g.setColor(Color.black);
  g.drawArc(x + largeur / 3, y - hauteur / 12, largeur / 3, hauteur / 4, -125, 70);
  // Les yeux
  g.fillOval(milieux - largeur / 8, y + hauteur / 12, largeur / 12, hauteur / 24);
  g.fillOval(milieux + largeur / 8 - largeur / 12, y + hauteur / 12, largeur / 12, hauteur / 24);
  // Le corps
  g.drawLine(milieux, y + hauteur / 4, milieux, y + hauteur * 3 / 4);
  // Les bras
  g.drawLine(x, y + hauteur / 4, milieux, milieuy);
  g.drawLine(x + largeur, y + hauteur / 4, milieux, milieuy);
  // Les jambes
  g.drawLine(x, y + hauteur, milieux, y + hauteur * 3 / 4);
  g.drawLine(x + largeur, y + hauteur, milieux, y + hauteur * 3 / 4);
 }

 public void paint(Graphics g) {
  super.paint(g);
  paintBot(g, 10, 40, 50, 100);
  paintBot(g, 200, 200, 20, 40);
  paintBot(g, 150, 500, 50, 25);
  paintBot(g, 300, 50, 40, 160);
  paintIti(g, 150, 200, 40, 80);
  paintIti(g, 50, 350, 100, 200);
  paintIti(g, 300, 300, 60, 120);
 }

 public static void main(String args[]) {
  new ExercicePlusieursBotEtIti();
 }
}
```

Résultat :



4.3 Traitement des événements de souris (interface *MouseListener*)

Dans les applications interactives, il faut pouvoir détecter les actions de l'utilisateur (déplacement de la souris, click de la souris, touche de clavier, etc.) et y réagir. Le package *java.awt* inclut les mécanismes à cet effet.

Exemple. JavaPasAPas/chapitre_4/ExempleEvenementSouris.java

Le programme suivant illustre les mécanismes de base de détection des actions de la souris. Le programme répond à un *click de la souris* (bouton de gauche enfoncé avec Windows) en déplaçant le *Bot* à la position du click. La position du click est déterminée par la position du curseur de souris au moment où le bouton est enfoncé.

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

public class ExempleEvenementSouris extends JFrame implements MouseListener {
  // Variables d'objet qui contiennent les coordonnées de la souris
  // Le premier sera dessiné à la coordodonnée (0,0)
  private static int x = 0; // Coordodonnée x du Bot à dessiner
  private static int y = 0; // Coordodonnée y du Bot à dessiner

  public ExempleEvenementSouris() {
    super("Exemple de traitement d'événements de la souris");

    // Le paramètre this de addMouseListener() indique que l'objet qui doit
    // réagir aux événements de souris est l'objet
    // qui est créé par ce constructeur
    addMouseListener(this);

    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    this.setSize(400, 600);
    this.setVisible(true);
  }

  // méthode d'objet de la classe ExempleEvenementSouris qui est
  // appelée si le bouton de souris est enfoncé
  public void mousePressed(MouseEvent leMouseEvent) {
    x = leMouseEvent.getX(); // place la coordodonnée x de la souris dans la variable x
    y = leMouseEvent.getY(); // place la coordodonnée y de la souris dans la variable y
    // repaint() provoque un nouvel appel à paint()
    repaint();
  }

  // Il faut absolument définir les autres méthodes pour les autres
  // événements de souris même s'ils ne font rien
  public void mouseClicked(MouseEvent leMouseEvent) {}

  public void mouseEntered(MouseEvent leMouseEvent) {}

  public void mouseExited(MouseEvent leMouseEvent) {}

  public void mouseReleased(MouseEvent leMouseEvent) {}
```

```
  public static void paintBot(Graphics g, int x, int y, int largeur, int hauteur) {
    g.setColor(Color.green);
    g.fillOval(x, y, largeur, hauteur / 2); // La tête

    g.setColor(Color.black);
    g.fillRect(x + largeur / 4, y + hauteur / 8, largeur / 10, hauteur / 20); // L'oeil gauche
    g.fillRect(
        x + largeur * 3 / 4 - largeur / 10,
        y + hauteur / 8,
        largeur / 10,
        hauteur / 20); // L'oeil droit
    g.drawLine(
        x + largeur / 4,
        y + hauteur * 3 / 8,
        x + largeur * 3 / 4,
        y + hauteur * 3 / 8); // La bouche

    g.setColor(Color.red);
    g.fillRect(x, y + hauteur / 2, largeur, hauteur / 2); // Le corps
  }

  public void paint(Graphics g) {
    super.paint(g);
    paintBot(g, x, y, 50, 100);
  }

  public static void main(String args[]) {
    new ExempleEvenementSouris();
  }
}
```

Chaque fois que l'utilisateur emploie la souris, ceci produit un *événement* (*event*) *d'interface à l'utilisateur*. Un événement d'interface à l'utilisateur est le résultat d'une action de l'utilisateur sur un périphérique d'entrée tel que la souris ou le clavier. Pour réagir à l'événement, il faut programmer une méthode qui sera automatiquement activée lorsque l'événement est détecté. Cette méthode doit avoir une signature prédéfinie (par exemple, mousePressed(MouseEvent e)) pour un événement correspondant à enfoncer le bouton de la souris). Elle doit être incluse dans la classe d'un objet qui est désigné par le programmeur comme écouteur (*listener*) de l'événement.

Dans notre exemple, l'objet écouteur est l'objet de la classe *ExempleEvenementSouris* qui représente une fenêtre **JFrame**. La ligne suivante du constructeur désigne *this* comme écouteur des événements de la souris. L'identificateur réservé *this* représente une référence à l'objet de la classe *ExempleEvenementSouris* qui est construit par le constructeur.

```
        addMouseListener(this);
```

- **Notion d'interface Java**

Il faut que la classe de l'objet écouteur implémente des méthodes particulières pour répondre aux différents événements de la souris. Ceci est représenté par le fait que la classe de l'objet écouteur (*ExempleEvenementSouris*) doit implémenter l'interface java.awt.event.**MouseListener**. Ceci est spécifié dans la ligne suivante par la clause *implements MouseListener* dans la déclaration de la classe :

```
public class ExempleEvenementSouris extends JFrame implements
MouseListener
```

Mettre en œuvre l'interface *java.awt.event.MouseListener* signifie que la classe doit contenir la définition d'un certain nombre de méthodes dont les signatures sont précisées dans *l'interface* Java désignée par le nom *java.awt.event.MouseListener*.

Qu'est-ce qu'une interface au sens de Java? Une interface Java est simplement un squelette de classe au sens où une interface contient la définition d'un ensemble de signatures de méthodes mais sans nécessairement spécifier le corps de la méthode. Par exemple, l'interface java.awt.event.**MouseListener** est définie de la manière suivante :

```
package java.awt.event;

public abstract interface MouseListener extends java.util.EventListener
{
        public abstract void mouseClicked(java.awt.event.MouseEvent e){}
        public abstract void mousePressed(java.awt.event.MouseEvent e){}
        public abstract void mouseReleased(java.awt.event.MouseEvent e){}
        public abstract void mouseEntered(java.awt.event.MouseEvent e){}
        public abstract void mouseExited(java.awt.event.MouseEvent e){}
}
```

Les méthodes de l'interface (*mouseClicked*(), *mousePressed*(), …) ont un corps vide. Il est prévu que le corps de ces méthodes soit précisé dans la classe qui *implémente* l'interface.

Comme elle implémente l'interface java.awt.event.**MouseListener** (clause *implements java.awt.event.MouseListener*), la classe *ExempleEvenementSouris* doit contenir toutes ces méthodes. Ainsi en forçant une classe à implémenter une interface, la classe est contrainte à fournir les méthodes de cette interface. Cette manière de contraindre une classe à répondre à des méthodes garantit que la classe sera en mesure de fournir une réponse lors de l'exécution du

programme. Le rôle des méthodes de l'interface java.awt.event.**MouseListener** est de spécifier comment réagir aux différents événements de la souris. En forçant un écouteur d'événement de souris à implémenter l'interface java.awt.event.**MouseListener**, il est garanti que l'objet écouteur puisse répondre aux différents événements de la souris. La souris peut générer plusieurs sortes d'événements tel que : bouton enfoncé, bouton cliqué, bouton relâché, curseur de souris déplacé hors des limites de la fenêtre, etc. Pour réagir à chacune des sortes d'événements, il faut définir la méthode correspondante de l'interface **MouseListener**. Par exemple, pour réagir à l'enfoncement du bouton de la souris, il faut définir la méthode **mousePressed**(**MouseEvent** e) dans la classe de l'écouteur *ExempleEvenementSouris*. Ces méthodes seront appelées automatiquement lorsque les événements se produisent.

Voici la méthode dans notre exemple :

```
public void mousePressed(MouseEvent leMouseEvent){
  x = leMouseEvent.getX(); // place la coordonnée x de la souris dans la variable x
  y = leMouseEvent.getY(); // place la coordonnée y de la souris dans la variable y
  // repaint() provoque un nouvel appel à paint()
  repaint();
}
```

La méthode a un paramètre qui est un objet de la classe **MouseEvent**. Cet objet contient des informations au sujet de l'événement de souris. La méthode d'objet **getX**() de **MouseEvent** retourne la coordonnée *x* du curseur au moment où le bouton de la souris est enfoncé et la méthode **getY**(), la coordonnée *y*. Ces coordonnées sont employées dans notre programme afin de pouvoir dessiner un Bot à cet endroit.

Ensuite, la méthode *repaint*() est appelée. Cette méthode est héritée de la classe **JFrame** qui est la super-classe de *ExempleEvenementSouris*. Elle provoque automatiquement un appel à *paint*() qui dessine un nouveau Bot à la position courante de la souris.

Méthodes *paint*(), *repaint*() et *update*()

Pourquoi ne devrait-on pas appeler *paint*() directement ? Ceci est dû au fait que la gestion de l'affichage est en réalité un processus complexe dans un contexte à multifenêtrage comme c'est le cas avec un système d'exploitation tel que Windows. Pour s'assurer que tout se passe correctement, il faut une coordination avec les autres événements des autres fenêtres. Il est prévu que la méthode *repaint*() effectue cette coordination avant que *paint*() ne soit appelée. En particulier, la méthode *repaint*() appelle la méthode *update*() qui efface le contenu du contexte graphique en rétablissant la couleur de fond et elle appelle *paint*() par la suite. D'autre part, c'est la méthode *repaint*() qui est appelée automatiquement lorsqu'il faut redessiner le contenu de la fenêtre lors des manipulations des fenêtres.

Les autres méthodes de l'interface **MouseListener** sont vides dans notre exemple car il n'y a rien à faire pour ces autres sortes d'événements :

```
    public void mouseClicked(MouseEvent leMouseEvent){}
    public void mouseEntered(MouseEvent leMouseEvent){}
    public void mouseExited(MouseEvent leMouseEvent){}
    public void mouseReleased(MouseEvent leMouseEvent){}
```

Elles doivent tout de même être déclarées dans la classe *ExempleEvenementSouris* car elle implémente l'interface **MouseListener**. Nous verrons plus loin qu'il est possible d'éviter ces déclarations vides en employant une classe *Adaptor*.

Un peu comme pour la méthode *paint*(), cette manière de répondre aux événements peut sembler un peu mystérieuse car on ne voit nulle part l'appel aux méthodes de réponse à l'événement. Il faut comprendre que tout le code de gestion des événements se trouve en quelque part dans du code hérité de la super-classe **JFrame** et qu'il serait compliqué d'expliquer le détail de la gestion d'événement à ce point-ci.

- **Variables déclarées au niveau de la classe**

Vous avez peut-être noté que les variables *x* et *y* ne sont pas définies dans la méthode *mousePressed*() mais au début du corps de la classe dans les lignes suivantes :

```
    private int x = 0; // Coordonnée x du Bot à dessiner
    private int y = 0; // Coordonnée y du Bot à dessiner
```

On appelle parfois ces variables des *attributs* de la classe. En Java, les attributs peuvent avoir différents degrés de visibilité indiqué par un terme tel que *public*, *private* ou *protected*. Si aucun de ces termes qualifiant la visibilité précède la déclaration de la variable, alors cette variable a une visibilité par défaut ce qui signifie que la variable est accessible par toutes les classes dans le même package, mais par seulement ces classes. Le terme *protected* signifie que l'attribut est accessible par toutes les classes dans le même package mais aussi par les classes dérivées. Le terme *public* signifie que l'attribut est librement accessible. Le *private* signifie que ces variables ne peuvent être accédées à partir des autres classes. Les méthodes d'une classe ont la même notion de visibilité : nous avons déjà vu que la méthode main d'un programme doit avoir la visibilité *public*.

Dans notre exemple, les variables sont des variables d'objet de la classe *ExempleEvenementSouris*. L'emploi de variables d'objet permet à ces variables d'être accédées non seulement dans *mousePressed*() mais aussi dans la méthode *paint*() :

```
    public void paint (Graphics g) {
        super.paint(g);
        paintBot(g,x,y,50,100);
    }
```

L'utilisation de variables d'objet permet d'accéder aux mêmes variables dans les deux méthodes. Ainsi des données peuvent être partagées entre méthodes d'objet de la même classe. Si on avait déclaré les variables localement dans la méthode *mousePressed*(), la méthode *paint*() n'aurait pu y accéder. D'ailleurs, ceci aurait provoqué une erreur de compilation.

Exemple. Si vous compilez le programme suivant, vous obtiendrez une erreur de compilation indiquant que *x* et *y* n'existe pas dans *paint*() :

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

public class ExempleVariablesLocalesErreur extends JFrame implements MouseListener {

  public ExempleVariablesLocalesErreur() {
    super("Exemple de traitement d'événements de la souris");
```

```
  // Le paramètre this de addMouseListener() indique que l'objet qui doit
  // réagir aux événements de souris est l'objet
  // qui est créé par ce constructeur
  addMouseListener(this);

  this.setDefaultCloseOperation(EXIT_ON_CLOSE);
  this.setSize(400, 600);
  this.setVisible(true);
 }

 // Méthode d'objet de la classe ExempleEvenementSouris qui est
 // appelée si le bouton de souris est enfoncé
 public void mousePressed(MouseEvent leMouseEvent) {
  int x = leMouseEvent.getX(); // place la coordonnée x de la souris dans la variable x
  int y = leMouseEvent.getY(); // place la coordonnée y de la souris dans la variable y
  // repaint() provoque un nouvel appel à paint()
  repaint();
 }

 // Il faut absolument définir les autres méthodes pour les autres
 // événements de souris même s'il ne font rien
 public void mouseClicked(MouseEvent leMouseEvent) {}

 public void mouseEntered(MouseEvent leMouseEvent) {}

 public void mouseExited(MouseEvent leMouseEvent) {}

 public void mouseReleased(MouseEvent leMouseEvent) {}

 public static void paintBot(Graphics g, int x, int y, int largeur, int hauteur) {
  g.setColor(Color.green);
  g.fillOval(x, y, largeur, hauteur / 2); // La tête

  g.setColor(Color.black);
  g.fillRect(x + largeur / 4, y + hauteur / 8, largeur / 10, hauteur / 20); // L'oeil gauche
  g.fillRect(
     x + largeur * 3 / 4 - largeur / 10,
     y + hauteur / 8,
     largeur / 10,
     hauteur / 20); // L'oeil droit
  g.drawLine(
     x + largeur / 4,
     y + hauteur * 3 / 8,
     x + largeur * 3 / 4,
     y + hauteur * 3 / 8); // La bouche

  g.setColor(Color.red);
  g.fillRect(x, y + hauteur / 2, largeur, hauteur / 2); // Le corps
 }

 public void paint(Graphics g) {
  super.paint(g);
  paintBot(g, x, y, 50, 100);
 }

 public static void main(String args[]) {
  new ExempleVariablesLocalesErreur();
 }
}
```

L'erreur de compilation vient du fait qu'une variable déclarée localement dans une méthode ne peut être accédée dans une autre méthode.

Portée d'une variable, variable _locale_

La *portée* d'une variable déclarée dans une méthode est limitée au corps de la méthode. En d'autres mots, la variable déclarée dans une méthode est *locale* à la méthode.

Concrètement, ceci signifie que la variable disparaît en quelque sorte lorsque l'exécution de la méthode est terminée. Il serait possible de déclarer des variables *x* et *y* dans chacune des deux méthodes, *mousePressed*() et *paint*(), mais ceci n'aurait pas produit le résultat recherché car ces variables seraient en réalité des variables différentes même si elles portent le même nom !

Pour que le contenu d'une variable soit accessible à plusieurs méthodes de la même classe, il faut que la variable soit déclarée de manière globale au niveau de classe. D'autre part, une variable déclarée au niveau de la classe peut-être soit une variable de classe ou une variable d'objet.

***Variable de classe* (*static*)**

L'identificateur réservé *static* indique que c'est une variable de classe. Quand la mention *static* est omise, on dit parfois qu'il s'agit d'une variable d'instance ou variable d'objet.

Dans l'exemple précédent, nous avons employé des variables d'objet. Mais, on aurait aussi bien pu les définir comme des variables de classe sans que cela n'affecte le comportement de l'exemple puisqu'il n'y a qu'un seul objet de la classe *ExempleEvenementSouris*.

Exercice. Ajoutez l'identificateur réservé *static* dans la déclaration des variables *x* et *y* de *ExempleEvenementSouris* et faites exécuter le programme.

```
    private static int x = 0; // Coordonnée x du Bot à dessiner
    private static int y = 0; // Coordonnée y du Bot à dessiner
```

Dans le contexte de cet exemple, le résultat est le même que si les variables étaient des variables d'objet. Voyons maintenant un cas où l'emploi d'une variable de classe ou d'objet ne produit pas le même effet parce qu'il y a plus d'un objet de la même classe.

Différence entre *variable d'objet* et *variable de classe*

Il y a une différence importante entre variable d'objet (ou variable d'instance) et variable de classe. Il n'y a qu'une valeur pour une variable de classe peu importe le nombre d'objets de la classe. Ceci signifie que tous les objets de la classe partagent la même variable. Dans le cas d'une variable d'objet, il y a en quelque sorte une variable différente pour chacun des objets.

Pour voir la différence entre variable d'objet et de classe, il faut créer au moins deux objets de la classe *ExempleEvenementSouris*, c'est-à-dire deux fenêtres.

Exemple. JavaPasAPas/chapitre_4/
ExempleEvenementSouris2Fenetres.java

L'exemple suivant illustre l'effet de l'utilisation de variables de classe dans le cas de plusieurs objets de la même classe.

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

public class ExempleEvenementSouris2Fenetres extends JFrame implements MouseListener {
 // Variables de classe x et y
 private static int x = 0; // Coordonnée x du Bot à dessiner
 private static int y = 0; // Coordonnée y du Bot à dessiner

 public ExempleEvenementSouris2Fenetres() {
  super("Exemple de traitement d'événements de la souris");

  // Le paramètre this de addMouseListener() indique que l'objet qui doit
  // réagir aux événements de souris est l'objet
  // qui est créé par ce constructeur
  addMouseListener(this);

  this.setDefaultCloseOperation(EXIT_ON_CLOSE);
  this.setSize(400, 600);
  this.setVisible(true);
 }

 // Méthode d'objet de la classe ExempleEvenementSouris qui est
 // appelée si le bouton de souris est enfoncé
 public void mousePressed(MouseEvent leMouseEvent) {
  x = leMouseEvent.getX(); // place la Coordonnée x de la souris dans la variable x
  y = leMouseEvent.getY(); // place la Coordonnée y de la souris dans la variable y
  // repaint() provoque un nouvel appel à paint()
  repaint();
 }

 // Il faut absolument définir les autres méthodes pour les autres
 // événements de souris même s'il ne font rien
 public void mouseClicked(MouseEvent leMouseEvent) {}
```

```
 public void mouseEntered(MouseEvent leMouseEvent) {}

 public void mouseExited(MouseEvent leMouseEvent) {}

 public void mouseReleased(MouseEvent leMouseEvent) {}

 public static void paintBot(Graphics g, int x, int y, int largeur, int hauteur) {
  g.setColor(Color.green);
  g.fillOval(x, y, largeur, hauteur / 2); // La tête

  g.setColor(Color.black);
  g.fillRect(x + largeur / 4, y + hauteur / 8, largeur / 10, hauteur / 20); // L'oeil gauche
  g.fillRect(
     x + largeur * 3 / 4 - largeur / 10,
     y + hauteur / 8,
     largeur / 10,
     hauteur / 20); // L'oeil droit
  g.drawLine(
     x + largeur / 4,
     y + hauteur * 3 / 8,
     x + largeur * 3 / 4,
     y + hauteur * 3 / 8); // La bouche

  g.setColor(Color.red);
  g.fillRect(x, y + hauteur / 2, largeur, hauteur / 2); // Le corps
 }

 public void paint(Graphics g) {
  super.paint(g);
  paintBot(g, x, y, 50, 100);
  g.drawString("x=" + x + " y=" + y, 10, 550);
 }

 public static void main(String args[]) {
  new ExempleEvenementSouris2Fenetres();
  new ExempleEvenementSouris2Fenetres();
 }
}
```

La figure suivante montre les deux fenêtres dont le *Bot* a été positionné à deux endroits différents en utilisant la souris. La ligne suivante a été ajoutée à la méthode *paint*() pour afficher la valeur des variables *x* et *y*. La méthode **drawString(String** str, int x, int y) a comme paramètres un **String** à afficher et les coordonnées de l'emplacement où l'on veut afficher ce **String**.

```
        g.drawString("x="+x+" y="+y,10,550);
```

Dans la figure suivante produite par l'exécution du programme, le *Bot* de la fenêtre de gauche a été positionné avant celui de la fenêtre de droite. À ce point-ci de l'exécution du programme, les variables de classe *x* et *y* contiennent les coordonnées de *Bot* de la fenêtre de droite, soit *x* = 288 et *y* =442, parce que le *Bot* de la fenêtre de droite a été positionné le dernier. Il

n'y a pas de valeurs spécifiques à chacun des objets ! Les deux objets partagent la variable de classe.

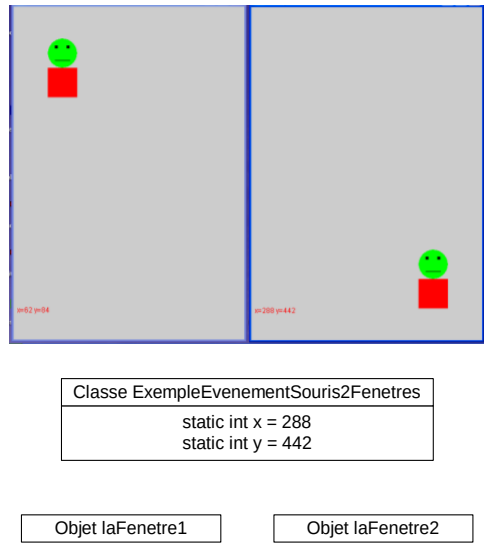
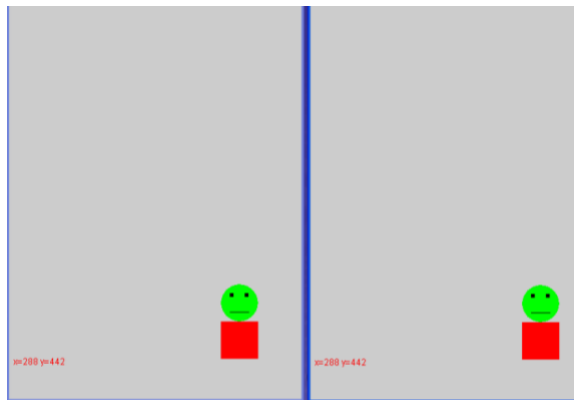


Figure 19. Variables de classe (static).

Si l'on réduit la fenêtre de gauche et on la réaffiche (le bouton de réduction est celui qui contient une petite barre horizontale dans le coin supérieur droit de la fenêtre), on obtient l'effet suivant :



Le *Bot* de la fenêtre de gauche est maintenant à la même position que celui de la fenêtre de droite ! Mais, nous ne l'avons pas repositionné ! Ceci est dû au fait qu'il n'y a qu'une seule valeur de x et y dans le cas de variables de

classe. Lorsque la fenêtre de gauche est réaffichée, ceci provoque un appel automatique à la méthode *paint*() sur l'objet qui correspond à la fenêtre de gauche. La méthode *paint*() accède aux coordonnées *x* et *y* afin d'afficher le *Bot*. Mais comme, la valeur courante de *x* et *y* contient les coordonnées du Bot de la fenêtre de droite (288, 442) parce qu'il a été positionné le dernier, la méthode *paint*() dessine le *Bot* au même endroit dans la fenêtre de gauche.

- **Variable d'objet**

Pour que chacune des deux fenêtres ait ses propres valeurs des coordonnées *x* et *y* du Bot, il faut en faire des variables d'objet. Dans ce cas, les deux objets ont leurs propres variables *x* et *y*. L'appel à *paint*() peut ainsi employer les valeurs de *x* et *y* particulières à l'objet.

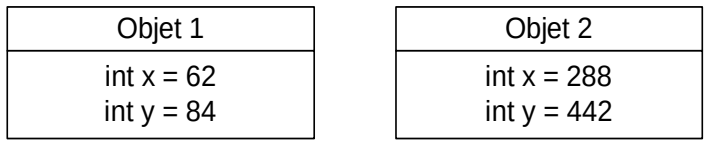


Figure 20. Variables d'objet.

Exemple. JavaPasAPas/chapitre_4/ExempleVariableDobjet.java

Dans *ExempleVariableDobjet*, les coordonnées du Bot sont des variables d'objet. Vous pouvez vérifier que le comportement des fenêtres est maintenant correct. Chacune des fenêtres a ses propres valeurs de *x* et *y* pour le Bot.

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

public class ExempleVariableDobjet extends JFrame implements MouseListener {
  // Variables d'objet qui contiennent les coordonnées de la souris
  // Le premier sera dessiné à la coordonnée (0,0)
  private int x = 0; // Coordonnée x du Bot é dessiner
  private int y = 0; // Coordonnée y du Bot é dessiner

  public ExempleVariableDobjet() {
    super("Exemple de variable d'objet x et y");

    // Le paramètre this de addMouseListener() indique que l'objet qui doit
    // réagir aux événements de souris est l'objet
    // qui est créé par ce constructeur
    addMouseListener(this);

    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
```

```
  this.setSize(400, 600);
  this.setVisible(true);
 }

 // Méthode d'objet de la classe ExempleEvenementSouris qui est
 // appelée si le bouton de souris est enfoncé
 public void mousePressed(MouseEvent leMouseEvent) {
  x = leMouseEvent.getX(); // place la coordonnée x de la souris dans la variable x
  y = leMouseEvent.getY(); // place la coordonnée y de la souris dans la variable y
  // repaint() provoque un nouvel appel é paint()
  repaint();
 }

 // Il faut absolument définir les autres méthodes pour les autres
 // événements de souris même s'il ne font rien
 public void mouseClicked(MouseEvent leMouseEvent) {}

 public void mouseEntered(MouseEvent leMouseEvent) {}

 public void mouseExited(MouseEvent leMouseEvent) {}

 public void mouseReleased(MouseEvent leMouseEvent) {}

 public static void paintBot(Graphics g, int x, int y, int largeur, int hauteur) {
  g.setColor(Color.green);
  g.fillOval(x, y, largeur, hauteur / 2); // La tête

  g.setColor(Color.black);
  g.fillRect(x + largeur / 4, y + hauteur / 8, largeur / 10, hauteur / 20); // L'oeil gauche
  g.fillRect(
     x + largeur * 3 / 4 - largeur / 10,
     y + hauteur / 8,
     largeur / 10,
     hauteur / 20); // L'oeil droit
  g.drawLine(
     x + largeur / 4,
     y + hauteur * 3 / 8,
     x + largeur * 3 / 4,
     y + hauteur * 3 / 8); // La bouche

  g.setColor(Color.red);
  g.fillRect(x, y + hauteur / 2, largeur, hauteur / 2); // Le corps
 }

 public void paint(Graphics g) {
  super.paint(g);
  paintBot(g, x, y, 50, 100);
  g.drawString("x=" + x + " y=" + y, 10, 550);
 }

 public static void main(String args[]) {
  new ExempleVariableDobjet();
  new ExempleVariableDobjet();
 }
}
```

En conclusion, une variable de classe peut être employée si tous les objets de la classe partagent la même valeur pour cette variable. Si chacun des objets doit avoir sa propre valeur distincte pour la variable, il faut employer une variable d'objet.

- **Méthode d'objet ou de classe**

Une distinction analogue existe entre les méthodes d'objet et de classe. Une méthode de classe est appropriée lorsqu'elle n'utilise pas les variables d'objet. En revanche, si la méthode utilise des variables d'objet, il faut en faire une méthode d'objet.

Examinons le cas de la méthode *paintBot*() dans l'exemple précédent. C'est une méthode de classe et ceci est correct car elle n'utilise pas directement les variables d'objet *x* et *y*. En effet, elle utilise les paramètres *x* et *y* qui sont passés par l'appel *paintBot*(g,x,y,50,100) dans la méthode *paint*().

```
    public void paint (Graphics g) {
        super.paint(g);
        paintBot(g,x,y,50,100);
        g.drawString("x="+x+" y="+y,10,550);
    }
```

Dans *paint*(), les variables *x* et *y* correspondent aux variables d'objet. Les valeurs des variables d'objet sont passées à la méthode de classe par l'appel *paintBot*(g,x,y,50,100).

Regardons maintenant un autre exemple, où plutôt que de passer les valeurs de *x* et *y* à *paintBot*() par les paramètres, la méthode *paintBot*() utilise directement les variables d'objet *x* et *y*.

Exemple.

JavaPasAPas/chapitre_4/ExempleMethodeDobjetPaintBot.java

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

public class ExempleMethodeDobjetPaintBot extends JFrame implements MouseListener {
  // Variables d'objet qui contiennent les coordonnées de la souris
  // Le premier sera dessiné à la coordonnée (0,0)
  private int x = 0; // Coordonnée x du Bot à dessiner
  private int y = 0; // Coordonnée y du Bot à dessiner

  public ExempleMethodeDobjetPaintBot() {
    super("Exemple de traitement d'événements de la souris");

    // Le paramètre this de addMouseListener() indique que l'objet qui doit
    // réagir aux événements de souris est l'objet
    // qui est créé par ce constructeur
    addMouseListener(this);

    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
```

```
  this.setSize(400, 600);
  this.setVisible(true);
 }

 // Méthode d'objet de la classe ExempleEvenementSouris qui est
 // appelée si le bouton de souris est enfoncé
 public void mousePressed(MouseEvent leMouseEvent) {
  x = leMouseEvent.getX(); // place la coordonnée x de la souris dans la variable x
  y = leMouseEvent.getY(); // place la coordonnée y de la souris dans la variable y
  // repaint() provoque un nouvel appel à paint()
  repaint();
 }

 // Il faut absolument définir les autres méthodes pour les autres
 // événements de souris méme s'il ne font rien
 public void mouseClicked(MouseEvent leMouseEvent) {}

 public void mouseEntered(MouseEvent leMouseEvent) {}

 public void mouseExited(MouseEvent leMouseEvent) {}

 public void mouseReleased(MouseEvent leMouseEvent) {}

 public void paintBot(Graphics g, int largeur, int hauteur) {
  // La méthode d'objet utilise directment les variables d'objet x et y
  g.setColor(Color.green);
  g.fillOval(x, y, largeur, hauteur / 2); // La tête

  g.setColor(Color.black);
  g.fillRect(x + largeur / 4, y + hauteur / 8, largeur / 10, hauteur / 20); // L'oeil gauche
  g.fillRect(
     x + largeur * 3 / 4 - largeur / 10,
     y + hauteur / 8,
     largeur / 10,
     hauteur / 20); // L'oeil droit
  g.drawLine(
     x + largeur / 4,
     y + hauteur * 3 / 8,
     x + largeur * 3 / 4,
     y + hauteur * 3 / 8); // La bouche

  g.setColor(Color.red);
  g.fillRect(x, y + hauteur / 2, largeur, hauteur / 2); // Le corps
 }

 public void paint(Graphics g) {
  super.paint(g);
  paintBot(g, 50, 100);
  g.drawString("x=" + x + " y=" + y, 10, 550);
 }

 public static void main(String args[]) {
  new ExempleMethodeDobjetPaintBot();
  new ExempleMethodeDobjetPaintBot();
 }
}
```

Dans cet exemple, la méthode d'objet *paintBot*() ne peut être une méthode de classe car elle accède aux variables d'objet *x* et *y*. Si la méthode *paintBot*() était déclarée *static*, une erreur de compilation serait signalée.

4.4 Constantes (final)

Lorsqu'il précède une variable, l'identificateur réservé *final* indique que la variable ne peut être modifiée. Une telle variable désigne donc une *constante* dans le programme Java. Une pratique souvent employée consiste à définir les constantes utilisées dans un programme comme des variables *final*. Ceci améliore la lisibilité du programme. D'autre part, si le programmeur doit changer la valeur de la constante, il évite d'avoir à parcourir le programme pour retrouver toutes les occurrences de la valeur constante pour la modifier.

Exemple. JavaPasAPas/chapitre_4/ExempleConstantesFinal.java

Dans l'exemple suivant qui reprend l'exemple précédent, la hauteur et la largeur de la fenêtre sont définies comme des variables *final*. Ces constantes sont employées dans le constructeur de fenêtre *ExempleConstantesFinal*() et la méthode *paint*() pour ajuster la dimension du Bot et la position du message affichée.

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

public class ExempleConstantesFinal extends JFrame
   implements MouseListener { // Constantes pour la taille de la fenetre
 private static final int LARGEURFENETRE = 400;
 private static final int HAUTEURFENETRE = 600;

 // Variables d'objet qui contiennent les coordonnées de la souris
 // Le premier sera dessiné à la coordonnée (0,0)
 private int x = 0; // Coordonnée x du Bot à dessiner
 private int y = 0; // Coordonnée y du Bot à dessiner

 public ExempleConstantesFinal() {
  super("Exemple de traitement d'événements de la souris");

  // Le paramêtre this de addMouseListener() indique que l'objet qui doit
  // réagir aux événements de souris est l'objet
  // qui est créé par ce constructeur
  addMouseListener(this);

  this.setDefaultCloseOperation(EXIT_ON_CLOSE);
  this.setSize(LARGEURFENETRE, HAUTEURFENETRE);
  this.setVisible(true);
 }

 // Méthode d'objet de la classe ExempleEvenementSouris qui est
 // appelée si le bouton de souris est enfoncé
 public void mousePressed(MouseEvent leMouseEvent) {
  x = leMouseEvent.getX(); // place la coordonnée x de la souris dans la variable x
  y = leMouseEvent.getY(); // place la coordonnée y de la souris dans la variable y
  // repaint() provoque un nouvel appel à paint()
  repaint();
 }

 // Il faut absolument définir les autres méthodes pour les autres
```

```
// événements de souris même s'il ne font rien
public void mouseClicked(MouseEvent leMouseEvent) {}

public void mouseEntered(MouseEvent leMouseEvent) {}

public void mouseExited(MouseEvent leMouseEvent) {}

public void mouseReleased(MouseEvent leMouseEvent) {}

public void paintBot(Graphics g, int largeur, int hauteur) {
  // La méthode d'objet utilise directment les variables d'objet x et y
  g.setColor(Color.green);
  g.fillOval(x, y, largeur, hauteur / 2); // La tête

  g.setColor(Color.black);
  g.fillRect(x + largeur / 4, y + hauteur / 8, largeur / 10, hauteur / 20); // L'oeil gauche
  g.fillRect(
      x + largeur * 3 / 4 - largeur / 10,
      y + hauteur / 8,
      largeur / 10,
      hauteur / 20); // L'oeil droit
  g.drawLine(
      x + largeur / 4,
      y + hauteur * 3 / 8,
      x + largeur * 3 / 4,
      y + hauteur * 3 / 8); // La bouche

  g.setColor(Color.red);
  g.fillRect(x, y + hauteur / 2, largeur, hauteur / 2); // Le corps
 }

 public void paint(Graphics g) {
  super.paint(g);
  paintBot(g, LARGEURFENETRE / 8, HAUTEURFENETRE / 6);
  g.drawString("x=" + x + " y=" + y, 10, HAUTEURFENETRE - 50);
 }

 public static void main(String args[]) {
  new ExempleConstantesFinal();
  new ExempleConstantesFinal();
 }
}
```

Exercice. Modifiez le programme précédent en définissant dans la classe *ExempleConstantesFinal* deux constantes LARGEURBOT et HAUTEURBOT qui seront utilisées dans l'appel à *paintBot*(). La valeur de ces constantes sera calculée à partir des constantes LARGEURFENETRE et HAUTEURFENETRE.

- **Ensemble de constantes (types de données énumérés)**

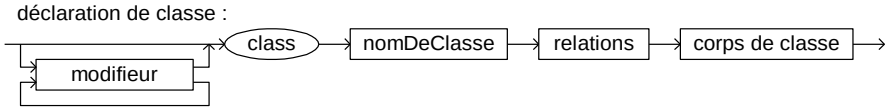
À noter que la constante **Color.green** qui représente une couleur de dessin est en fait un nom de variable *public static final* de la classe **Color**. Il en est de même pour les autres couleurs prédéfinies énumérées précédemment. Ainsi cet ensemble des couleurs est un ensemble de constantes dans la classe **Color**. Voici un extrait de la classe **Color** qui montre quelques déclarations de couleurs. Une couleur est un objet de la classe **Color** créé avec le constructeur **Color**(int r, int g, int b). Chacun des paramètres est un entier entre 0 et 255 qui spécifie l'intensité d'une des trois composantes de la couleur : rouge, vert et bleu. Cette manière de représenter les couleurs correspond au système RGB.

```
public class Color
   implements java.awt.Paint, java.io.Serializable
{
...
   public static final java.awt.Color white = new Color(255,255,255);
   public static final java.awt.Color lightGray = new Color(192,192,192);
   public static final java.awt.Color gray = new Color(128,128,128);
   public static final java.awt.Color darkGray = new Color(64,64,64);
   public static final java.awt.Color black = new Color(0,0,0);
   public static final java.awt.Color red = new Color(255,0,0);
   public static final java.awt.Color pink = new Color(255,175,175);
   public static final java.awt.Color orange = new Color(255,200,0);
   public static final java.awt.Color yellow = new Color(255,255,0);
   public static final java.awt.Color green = new Color(0,255,0);
   public static final java.awt.Color magenta =  new Color(255,0,255);
   public static final java.awt.Color cyan = new Color(0,255,255);
   public static final java.awt.Color blue = new Color(0,255,0);

...

   // Constructeur de couleur
   public void Color(int r, int g, int b)
   {...
   }
...
```

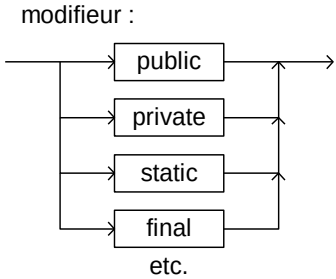
Pour accéder à une variable *public* d'une autre classe, il faut préfixer le nom de la variable avec le nom de sa classe, d'où l'utilisation de la syntaxe *Color.green* dans les exemples de programmes vus jusqu'à présent.

4.5 Sommaire d'une déclaration de classe

Résumons les concepts vus jusqu'à présent en portant un regard sommaire au sujet de la déclaration d'une classe dont le diagramme syntaxique est le suivant.



L'identificateur réservé *class* est optionnellement précédé d'une suite de modifieurs. Nous n'avons rencontré que *public* dans le cas d'une classe. Le sens des autres modifieurs sera expliqué ultérieurement.

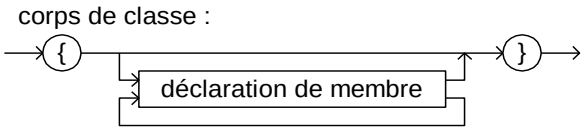


Après le nom de classe, les relations avec d'autres classes sont spécifiées (super-classe et interfaces à implémenter). Il ne peut y avoir qu'une seule super-classe mais il peut y avoir plus d'une interface à implémenter. Lorsque la clause *extends* est absente, par défaut, la classe est une sous-classe de *java.lang.Object* qui est la racine de la hiérarchie des classes Java. Cette classe contient des méthodes supportées par tous les objets. Par exemple, la méthode *getClass*() de *Object* retourne la classe de l'objet.

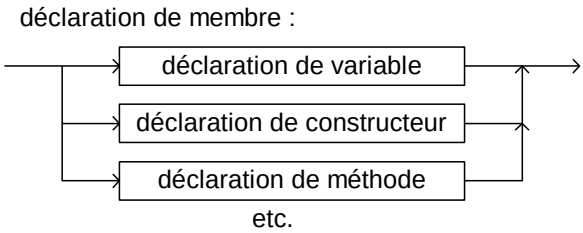
Ensuite, vient le corps de la classe qui contient les déclarations de ses membres.

Membre d'une classe

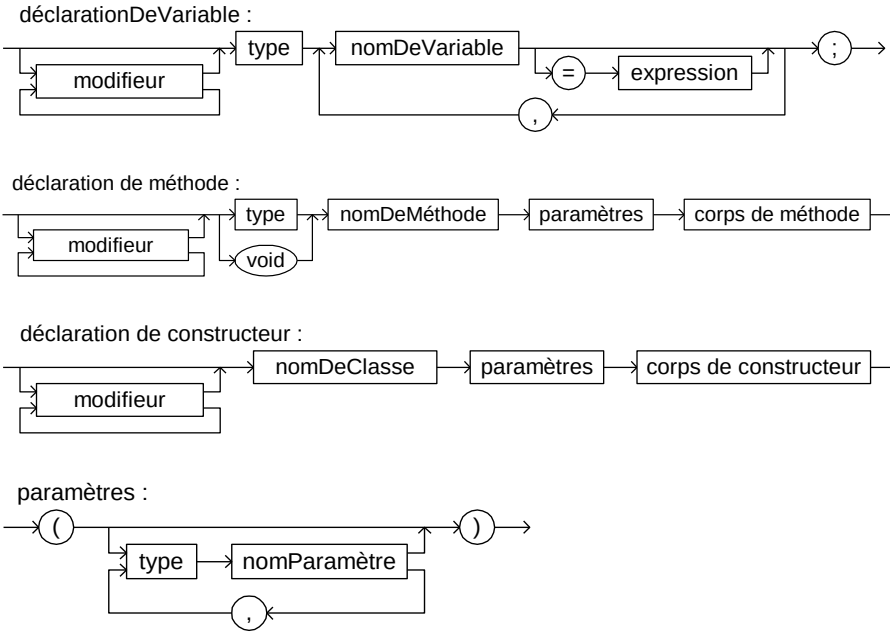
Les variables et méthodes définies au niveau d'une classe sont appelées des membres de cette classe.

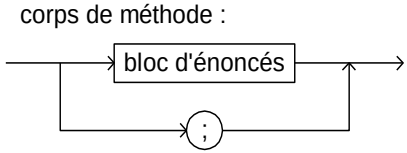


Le corps est une suite de déclarations de membres.



Voici la syntaxe pour chacun des types de membres.





Le modifieur *public* pour un membre signifie qu'il peut être accédé de l'extérieur de la classe par opposition à *private*. Le modifieur *static* spécifie qu'il s'agit d'une variable ou méthode de classe. Une méthode de classe ne peut accéder qu'aux variables de classe alors qu'une méthode d'objet peut accéder aux variables de classes et d'objet tel qu'indiqué dans le tableau suivant :

	Variable d'objet	Variable de classe
Méthode de classe	Accès interdit	Accès permis
Méthode d'objet	Accès permis	Accès permis

Pour une variable, le modifieur *final* signifie qu'elle ne peut être modifiée (constante).

- **Portée des variables**

Lorsqu'un nom de variable est employé dans le corps d'une méthode, ce nom de variable peut correspondre à une des quatre possibilités suivantes :

- variable de classe
- variable d'objet
- variable locale
- paramètre formel

Lorsqu'une variable locale, disons *v*, est déclarée et qu'elle porte le même nom qu'une variable de classe ou d'objet, le nom de variable *v* désigne la variable locale. Ainsi, la variable locale masque en quelque sorte une variable d'objet ou de classe du même nom. Pour désigner la variable de classe qui est masquée, on peut employer la syntaxe *nomClasse.v*. Dans le cas d'une méthode d'objet, la syntaxe *this.v* est aussi permise.

Exemple. JavaPasAPas/chapitre_4/ExempleVariablesLocales.java

Variable de classe masquée par une variable locale

```
public class ExempleVariablesLocales {
 public static int x = 0;

 public ExempleVariablesLocales() {}

 public static void m1() {
  int x = 1;
  System.out.println("Valeur de la variable locale x dans la méthode de classe m1() =" + x);
  System.out.println(
     "Valeur de la variable de classe x dans la méthode de classe m1() ="
       + ExempleVariablesLocales.x);
 }

 public void m2() {
  int x = 2;
  System.out.println("Valeur de la variable locale x dans la méthode d'objet m2() =" + x);
  System.out.println("Valeur de la variable de classe x dans la méthode d'objet m2() =" + this.x);
  System.out.println(
     "Valeur de la variable de classe x dans la méthode d'objet m2() ="
       + ExempleVariablesLocales.x);
 }

 public static void main(String args[]) {

  System.out.println("Appel de la méthode de classe m1() :");
  ExempleVariablesLocales.m1();

  System.out.println("Création d'un objet :");
  ExempleVariablesLocales unObjet = new ExempleVariablesLocales();
  System.out.println("Appel de la méthode de d'objet m2() :");
  unObjet.m2();
 }
}
```

Résultat :

```
Appel de la méthode de classe m1() :
Valeur de la variable locale x dans la méthode de classe m1() =1
Valeur de la variable de classe x dans la méthode de classe m1() =0
Création d'un objet :
Appel de la méthode de d'objet m2() :
Valeur de la variable locale x dans la méthode d'objet m2() =2
Valeur de la variable de classe x dans la méthode d'objet m2() =0
Valeur de la variable de classe x dans la méthode d'objet m2() =0
```

5. Introduction à l'animation 2D

Ce chapitre présente les concepts de base de l'animation graphique 2D. L'idée de base de l'animation est de dessiner une série d'images de manière suffisamment rapide pour donner l'impression d'un mouvement continu. Chacune des images produites est appelée une *scène* de l'animation. La prochaine section illustre le principe de base d'affichage d'une séquence de scènes par une animation élémentaire. Le mécanisme de traitement d'exceptions Java sera en même temps introduit. La section deux ajoute un raffinement à l'animation avec le mécanisme de double tampon.

5.1 Une première tentative d'animation

Le programme suivant est une première tentative d'animation simple du bonhomme Bot. L'objectif est de faire bouger le Bot de gauche à droite dans une fenêtre.

Exemple. JavaPasAPas/chapitre_5/ExempleJFrameAvecAnimationRatee.java

```
// Tentative d'animation par itération d'affichage de gauche à droite
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ExempleJFrameAvecAnimationRatee extends JFrame {

 // Constantes pour la taille de la fenetre et du Bot
 private static final int LARGEURFENETRE = 400;
 private static final int HAUTEURFENETRE = 600;
 private static final int LARGEURBOT = LARGEURFENETRE / 4;
 private static final int HAUTEURBOT = HAUTEURFENETRE / 3;

 public ExempleJFrameAvecAnimationRatee() {
  super("Exemple d'animation ratée");
  this.setDefaultCloseOperation(EXIT_ON_CLOSE);
  this.setSize(LARGEURFENETRE, HAUTEURFENETRE);
  this.setVisible(true);
 }

 // Méthode qui dessine un Bot dans un objet Graphics g
 // à l'échelle dans un rectangle englobant de paramètres x,y,largeur,hauteur
 public void paintBot(Graphics g, int x, int y, int largeur, int hauteur) {
  g.setColor(Color.green);
  g.fillOval(x, y, largeur, hauteur / 2); // La tête

  g.setColor(Color.black);
  g.fillRect(x + largeur / 4, y + hauteur / 8, largeur / 10, hauteur / 20); // L'oeil gauche
  g.fillRect(
    x + largeur * 3 / 4 - largeur / 10,
    y + hauteur / 8,
    largeur / 10,
    hauteur / 20); // L'oeil droit
  g.drawLine(
    x + largeur / 4,
```

```
      y + hauteur * 3 / 8,
      x + largeur * 3 / 4,
      y + hauteur * 3 / 8); // La bouche

    g.setColor(Color.red);
    g.fillRect(x, y + hauteur / 2, largeur, hauteur / 2); // Le corps
  }

  public void paint(Graphics g) {
    super.paint(g);
    for (int x = 0; x <= LARGEURFENETRE - LARGEURBOT; x = x + 5) {
      paintBot(g, x, HAUTEURFENETRE - 2 * HAUTEURBOT, LARGEURBOT, HAUTEURBOT);
      try {
        Thread.sleep(50);
      } catch (InterruptedException uneException) {
        System.out.println(uneException.toString());
      }
    }
  }

  public static void main(String args[]) {
    new ExempleJFrameAvecAnimationRatee();
  }
}
```

La méthode *paint*() dessine le Bot à répétition dans une boucle en variant graduellement la coordonnée *y* du Bot de 0 à LARGEURFENETRE-LARGEURBOT. La ligne suivante de *paint*() est le *for* de la boucle d'animation.

```
for(int x=0; x<=LARGEURFENETRE-LARGEURBOT; x = x + 5){
```

Dans la boucle, chacune des scènes est générée en dessinant le Bot à la coordonnée (x, HAUTEURFENETRE-2*HAUTEURBOT) où *x* varie de 0 à LARGEURFENETRE-LARGEURBOT.

```
paintBot(g,x,HAUTEURFENETRE-2*HAUTEURBOT,LARGEURBOT,HAUTEURBOT);
```

L'appel à *Thread.sleep*(50) introduit un délai de 50 ms afin de ralentir l'animation.

```
Thread.sleep(50);
```

Cet appel est encadré dans un énoncé *try* Java.

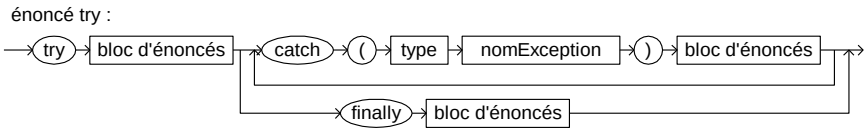
Énoncé try et exceptions

Une méthode Java peut interrompre la séquence normale d'exécution avec un énoncé *throw NomException*. Ceci est utile pour signaler que la méthode a rencontré quelque chose d'anormal pendant son exécution. En Java, on distingue deux catégories d'exceptions : *vérifiées* ou non *vérifiées*. Une exception vérifiée exige que l'appel de la méthode se fasse de manière à prévoir la possibilité d'une exception alors que cette gestion de la condition d'erreur est

optionnelle dans le cas d'une exception non vérifiée. Par exemple, plusieurs fonctions en Java peuvent générer une exception non vérifiée liée à un manque de mémoire (exception de type OutOfMemoryError) et ce type d'exception n'est généralement pas géré explicitement : dans ce cas, le programme se termine avec une erreur.

On gère les exceptions en encadrant l'appel dans un énoncé *try*. Par exemple, la méthode *Thread.sleep*() peut soulever une exception dans la catégorie vérifiée. La méthode qui contient l'appel *Thread.sleep*() attrape l'exception avec un énoncé *try*. Si la fonction ne peut gérer l'exception, elle peut s'en remettre aux fonctions l'appelant en déclarant qu'elle génére des exceptions en ajoutant une clause *throws* à la définition de la fonction.

Pour attraper l'exception avec un énoncé *try*, la syntaxe est :



Il peut y avoir plusieurs *catch* si la méthode soulève plusieurs types d'exceptions. Le type d'une exception est une classe Java. Un certain nombre de classes d'exception sont prédéfinies. Le bloc d'énoncé du *catch* est exécuté si l'exception du type spécifié est levée par la méthode appelée.

Dans notre exemple, si une exception de type *InterruptedException* est levée par l'appel à *Thread.sleep*(), un message est affiché par :

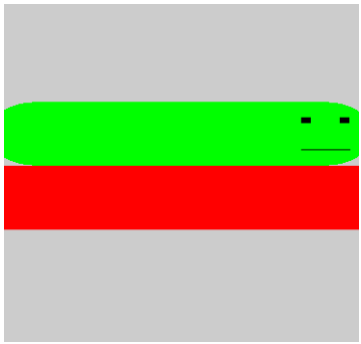
```
    System.err.println(uneException.toString());              }
```

Il est à noter que l'objet *System.err* est employé pour l'affichage plutôt que *System.out*. Cet objet représente une zone de texte réservée aux messages d'erreurs. Dans le cas d'une exécution avec la ligne de commande Windows, *System.err* et *System.out* sont tous les deux assignés à la fenêtre de l'invite de commande. Dans d'autres environnements, ces deux zones sont souvent affichées dans des fenêtres différentes.

Le message est produit par *uneException.toString*() qui retourne un message correspondant à l'exception levée. La partie *finally* est optionnelle et

n'apparaît pas dans notre exemple. Le bloc d'énoncé du *finally* est exécuté que l'exception soit levée ou pas.

L'exécution du programme produit le résultat final suivant qui n'est pas ce qui est visé … Un problème vient du fait qu'avant de dessiner le bonhomme à une nouvelle position, il faut l'effacer de la position précédente.



Exemple. L'effacement du Bot peut être effectué en appelant *clearRect*() en fin de boucle dans la méthode *paint*(). Cette méthode rétablit la couleur de fond d'écran pour le rectangle spécifié par les paramètres.

```
public void paint (Graphics g) {
  super.paint(g);
  for(int x=0; x<=LARGEURFENETRE-LARGEURBOT; x = x + 5){
    paintBot(g,x,HAUTEURFENETRE-2*HAUTEURBOT,LARGEURBOT,HAUTEURBOT);
    try {
      Thread.sleep(50);
    }
    catch(InterruptedException uneException){
      System.out.println(uneException.toString());
    }
    g.clearRect(x,HAUTEURFENETRE-2*HAUTEURBOT,LARGEURBOT,HAUTEURBOT);
  }
}
```

Malheureusement, ceci produit un effet désagréable de scintillement parce que les opérations de dessin ne sont pas instantanées. L'œil perçoit le processus de dessin, ce qui produit cet effet.

5.2 Animation par double tampon

Pour éviter le scintillement, il faut employer la technique de *double tampon*. Plutôt que de dessiner le Bot directement sur le contexte graphique de la fenêtre, on utilise un autre contexte graphique pour le dessin de la nouvelle scène. La figure suivante montre les deux contextes graphiques au début

d'une itération d'animation pour notre exemple de déplacement du Bot de gauche à droite.

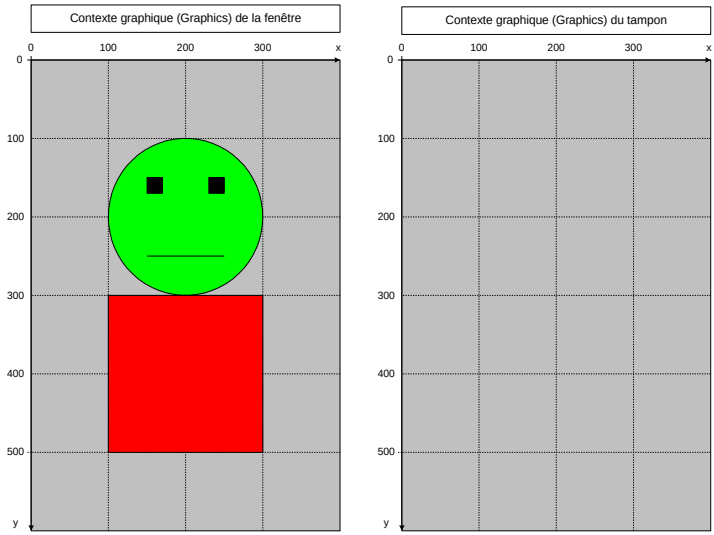


Figure 21. Double tampon.

Les opérations de dessin sont effectuées dans le deuxième contexte graphique. La position du Bot est légèrement décalée vers la droite dans le tampon.

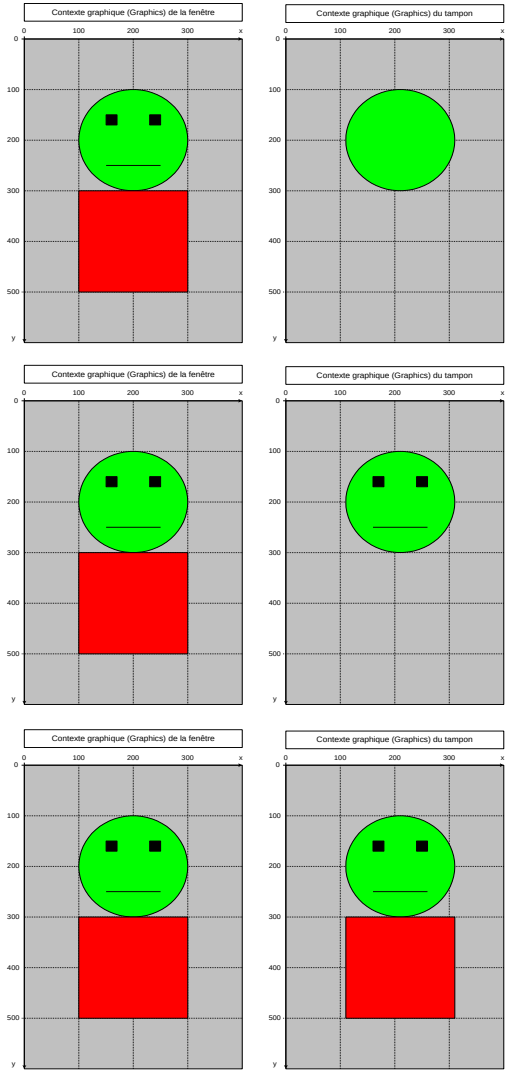


Figure 22. Dessin effectué sur le tampon.

Lorsque le dessin est terminé, l'image du deuxième contexte graphique est copiée directement d'un coup sur le contexte graphique de la fenêtre puis le Bot est effacé dans le tampon pour l'itération suivante. Ceci produit un effet d'animation beaucoup plus convaincant.

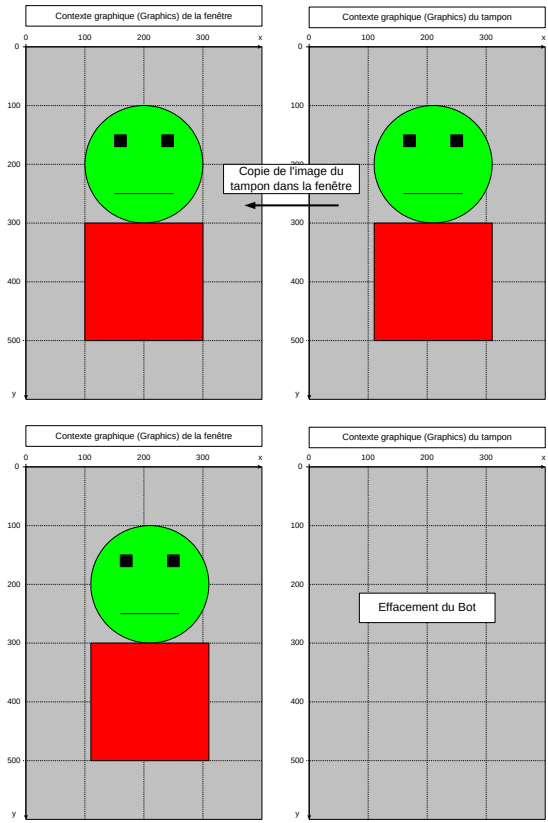


Figure 23. Copie de l'image du tampon dans le contexte graphique de la fenêtre.

***Alternance de tampon* (*page flipping*)**

Dans les applications graphiques à haute performance, en particulier pour les jeux d'ordinateur, il est avantageux d'éviter la copie de tampon en alternant la zone de mémoire associée à l'écran. Le mécanisme d'alternance peut être effectué par la carte graphique afin de le rendre rapide. Les deux contextes graphiques ont ainsi un rôle symétrique. Il est possible de réaliser cette stratégie d'animation en Java si la carte graphique le permet. Nous reviendrons sur ces détails de mise en œuvre par la suite.

Exemple. JavaPasAPas/chapitre_5/
ExempleJFrameAnimationDoubleTampon.java
Le programme suivant produit l'animation du Bot par double tampon.

```java
// Animation par double tampon
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ExempleJFrameAnimationDoubleTampon extends JFrame {

 // Constantes pour la taille de la fenêtre et du Bot
 private static final int LARGEURFENETRE = 400;
 private static final int HAUTEURFENETRE = 400;
 private static final int LARGEURBOT = LARGEURFENETRE / 4;
 private static final int HAUTEURBOT = HAUTEURFENETRE / 3;

 // Tampon pour construire l'image avant d'afficher
 Graphics tamponGraphics;
 Image tamponImage;

 public ExempleJFrameAnimationDoubleTampon() {
  super("Exemple d'animation par double tampon");
  this.setDefaultCloseOperation(EXIT_ON_CLOSE);
  this.setSize(LARGEURFENETRE, HAUTEURFENETRE);
  this.setVisible(true);
 }

 // Méthode qui dessine un Bot dans un objet Graphics g
 // à l'échelle dans un rectangle englobant de paramètres x,y,largeur,hauteur
 public void paintBot(Graphics g, int x, int y, int largeur, int hauteur) {
  g.setColor(Color.green);
  g.fillOval(x, y, largeur, hauteur / 2); // La tête

  g.setColor(Color.black);
  g.fillRect(x + largeur / 4, y + hauteur / 8, largeur / 10, hauteur / 20); // L'oeil gauche
  g.fillRect(
     x + largeur * 3 / 4 - largeur / 10,
     y + hauteur / 8,
     largeur / 10,
     hauteur / 20); // L'oeil droit
  g.drawLine(
     x + largeur / 4,
     y + hauteur * 3 / 8,
     x + largeur * 3 / 4,
     y + hauteur * 3 / 8); // La bouche

  g.setColor(Color.red);
  g.fillRect(x, y + hauteur / 2, largeur, hauteur / 2); // Le corps
 }

 public void paint(Graphics g) {
  super.paint(g);
  tamponImage = createImage(LARGEURFENETRE, HAUTEURFENETRE);
  tamponGraphics = tamponImage.getGraphics();
  for (int x = 0; x <= LARGEURFENETRE - LARGEURBOT; x = x + 5) {
   // Dessine le Bot dans le tampon
   paintBot(tamponGraphics, x, HAUTEURFENETRE - 2 * HAUTEURBOT, LARGEURBOT, HAUTEURBOT);
   // Copie le tampon dans le contexte graphique de la fenetre
   g.drawImage(tamponImage, 0, 0, this);
   try {
    Thread.sleep(50);
   } catch (InterruptedException uneException) {
    System.out.println(uneException.toString());
   }
   // Efface le Bot
   tamponGraphics.clearRect(x, HAUTEURFENETRE - 2 * HAUTEURBOT, LARGEURBOT, HAUTEURBOT);
  }
 }
```

```
 public static void main(String args[]) {
  new ExempleJFrameAnimationDoubleTampon();
 }
}
```

Dans la méthode *paint*() de programme, la ligne suivante crée un objet de la classe *Image* qui sert de tampon :

```
tamponImage = createImage(LARGEURFENETRE,HAUTEURFENETRE);
```

La ligne suivante extrait le contexte graphique *tamponGraphics* qui correspond à l'objet *tamponImage* :

```
tamponGraphics = tamponImage.getGraphics();
```

Les dessins sont par la suite effectués dans ce contexte graphique, par exemple :

```
        paintBot(tamponGraphics,x,HAUTEURFENETRE-
2*HAUTEURBOT,LARGEURBOT,HAUTEURBOT);
```

L'image *tamponImage* est ensuite copiée dans le contexte graphique *g* de la fenêtre à l'écran par :

```
g.drawImage(tamponImage,0,0,this);
```

Exercice. Modifiez le programme précédent de manière à ce que le Bot inverse sa direction de déplacement lorsqu'il atteint le bord de la fenêtre.

Solution. JavaPasAPas/chapitre_5/ExerciceJFrameAvecBotRebondissant.java

```
// Animation par double tampon
// Le Bot rebondit lorsqu'il atteint le bord de la fenêtre
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ExerciceJFrameAvecBotRebondissant extends JFrame {

 // Constantes pour la taille de la fenêtre et du Bot
 private static final int LARGEURFENETRE = 400;
 private static final int HAUTEURFENETRE = 400;
```

```
private static final int LARGEURBOT = LARGEURFENETRE / 4;
private static final int HAUTEURBOT = HAUTEURFENETRE / 3;

// Tampon pour construire l'image avant d'afficher
Graphics tamponGraphics;
Image tamponImage;

public ExerciceJFrameAvecBotRebondissant() {
  super("Bot rebondissant");
  this.setDefaultCloseOperation(EXIT_ON_CLOSE);
  this.setSize(LARGEURFENETRE, HAUTEURFENETRE);
  this.setVisible(true);
}

// Méthode qui dessine un Bot dans un objet Graphics g
// à l'échelle dans un rectangle englobant de paramètres x,y,largeur,hauteur
public void paintBot(Graphics g, int x, int y, int largeur, int hauteur) {
  g.setColor(Color.green);
  g.fillOval(x, y, largeur, hauteur / 2); // La tête

  g.setColor(Color.black);
  g.fillRect(x + largeur / 4, y + hauteur / 8, largeur / 10, hauteur / 20); // L'oeil gauche
  g.fillRect(
      x + largeur * 3 / 4 - largeur / 10,
      y + hauteur / 8,
      largeur / 10,
      hauteur / 20); // L'oeil droit
  g.drawLine(
      x + largeur / 4,
      y + hauteur * 3 / 8,
      x + largeur * 3 / 4,
      y + hauteur * 3 / 8); // La bouche

  g.setColor(Color.red);
  g.fillRect(x, y + hauteur / 2, largeur, hauteur / 2); // Le corps
}

public void paint(Graphics g) {
  super.paint(g);
  tamponImage = createImage(LARGEURFENETRE, HAUTEURFENETRE);
  tamponGraphics = tamponImage.getGraphics();
  int x = 0; // Coordonnée x du Bot
  int directionBot = 1; // +1 vers la droite et -1 vers la gauche
  int vitesseBot = 5; // nombre d'unités de déplacement à chaque itération de la boucle
  while (true) {
    // Dessine le Bot dans le tampon
    paintBot(tamponGraphics, x, HAUTEURFENETRE - 2 * HAUTEURBOT, LARGEURBOT, HAUTEURBOT);
    // Copie le tampon dans le contexte graphique de la fenetre
    g.drawImage(tamponImage, 0, 0, this);
    try {
      Thread.sleep(50);
    } catch (InterruptedException uneException) {
      System.out.println(uneException.toString());
    }
    // Efface le Bot
    tamponGraphics.clearRect(x, HAUTEURFENETRE-2*HAUTEURBOT,LARGEURBOT,HAUTEURBOT);
    // Déplace le Bot
    if (x + LARGEURBOT >= LARGEURFENETRE | x < 0) // Si atteint le bord
    directionBot = -directionBot; // Inverser la direction
    x = x + vitesseBot * directionBot; // Déplacement du Bot
  }
}

public static void main(String args[]) {
  new ExerciceJFrameAvecBotRebondissant();
}
```

```
}
```

Note : la fenêtre ne peut être fermée !

Un problème avec l'animation précédente est le fait que la boucle d'animation est infinie ! Lorsque l'on tente de fermer la fenêtre, rien ne se produit car la méthode *paint*() ne peut être interrompue. La méthode *paint*() ne devrait pas être employée de cette manière dans une application. Nous employons cette approche dans un premier temps afin de simplifier la présentation des concepts d'animation.

Pour arrêter le programme dans une fenêtre de commande Windows, vous pouvez taper <ctrl>-C[24]. Nous verrons par la suite une solution à ce problème qui consiste à effectuer la boucle d'animation à l'extérieur de la méthode *paint*().

Exercice. Animez votre bonhomme préféré en le déplaçant de haut en bas et lorsqu'il touche au bord de la fenêtre, inverser la direction.

Solution. JavaPasAPas/chapitre_5/ExerciceJFrameAvecItiRebondissant.java

```
// Animation par double tampon
// Le Iti se déplace à la verticale
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ExerciceJFrameAvecItiRebondissant extends JFrame {

 // Constantes pour la taille de la fenetre et du Iti
 private static final int LARGEURFENETRE = 400;
 private static final int HAUTEURFENETRE = 400;
 private static final int LARGEURITI = LARGEURFENETRE / 5;
 private static final int HAUTEURITI = HAUTEURFENETRE / 4;

 // Tampon pour construire l'image avant d'afficher
 Graphics tamponGraphics;
 Image tamponImage;

 public ExerciceJFrameAvecItiRebondissant() {
  super("Iti rebondissant");
  this.setDefaultCloseOperation(EXIT_ON_CLOSE);
  this.setSize(LARGEURFENETRE, HAUTEURFENETRE);
  this.setVisible(true);
 }

 // Méthode qui dessine un Iti dans un objet Graphics g
```

[24] Tapez la letter c alors que la touche «ctrl» est enfoncée.

```
// à l'échelle dans un rectangle englobant de paramètres x,y,largeur,hauteur
public static void paintIti(Graphics g, int x, int y, int largeur, int hauteur) {
  // Coordonnées du milieu du rectangle englobant pour faciliter les calculs
  int milieux = x + largeur / 2;
  int milieuy = y + hauteur / 2;
  // La tête
  g.setColor(Color.pink);
  g.fillOval(x + largeur / 3, y, largeur / 3, hauteur / 4);
  // Le sourire
  g.setColor(Color.black);
  g.drawArc(x + largeur / 3, y - hauteur / 12, largeur / 3, hauteur / 4, -125, 70);
  // Les yeux
  g.fillOval(milieux - largeur / 8, y + hauteur / 12, largeur / 12, hauteur / 24);
  g.fillOval(milieux + largeur / 8 - largeur / 12, y + hauteur / 12, largeur / 12, hauteur / 24);
  // Le corps
  g.drawLine(milieux, y + hauteur / 4, milieux, y + hauteur * 3 / 4);
  // Les bras
  g.drawLine(x + 1, y + hauteur / 4, milieux, milieuy);
  g.drawLine(x + largeur - 1, y + hauteur / 4, milieux, milieuy);
  // Les jambes
  g.drawLine(x + 1, y + hauteur - 1, milieux, y + hauteur * 3 / 4);
  g.drawLine(x + largeur - 1, y + hauteur - 1, milieux, y + hauteur * 3 / 4);
}

public void paint(Graphics g) {
  super.paint(g);
  tamponImage = createImage(LARGEURFENETRE, HAUTEURFENETRE);
  tamponGraphics = tamponImage.getGraphics();
  int y = 0; // Coordonnée y du Iti
  int directionIti = 1; // +1 vers la droite et -1 vers la gauche
  int vitesseIti = 5; // nombre d'unités de déplacement à chaque itération de la boucle

  while (true) {
    // Dessine le Iti dans le tampon
    paintIti(tamponGraphics, LARGEURFENETRE - 3 * LARGEURITI, y, LARGEURITI, HAUTEURITI);
    // Copie le tampon dans le contexte graphique de la fenetre
    g.drawImage(tamponImage, 0, 0, this);

    try {
      Thread.sleep(50);
    } catch (InterruptedException uneException) {
      System.out.println(uneException.toString());
    }
    // Efface le Iti
    tamponGraphics.clearRect(LARGEURFENETRE - 3 * LARGEURITI, y, LARGEURITI, HAUTEURITI);
    // Déplace le Iti
    if (y + HAUTEURITI >= HAUTEURFENETRE | y < 0) // Si atteint le bord
    directionIti = -directionIti; // Inverser la direction
    y = y + vitesseIti * directionIti; // Déplacement du Iti
  }
}

public static void main(String args[]) {
  new ExerciceJFrameAvecItiRebondissant();
}
}
```

Note : L'espace 2D du contexte graphique de la fenêtre correspond à toute la surface de la fenêtre incluant les bordures. Ainsi, la partie supérieure du contexte graphique est occupée par la barre de titre de la fenêtre. Les dessins

effectués dans cette portion du contexte graphique sont cachés par la barre de titre. Ceci est apparent lorsque le bonhomme atteint le haut de la fenêtre. Pour éviter ce problème, il faut arrêter le bonhomme à la valeur de *y* qui correspond à la hauteur de la barre de titre.

Exercice. JavaPasAPas/chapitre_5/ ExerciceJFrameAvecPingPongBot.java

Le Bot peut se déplacer en diagonale avec une vitesse de déplacement selon l'axe *x* et une autre selon l'axe *y*. Lorsqu'un bord de fenêtre gauche ou droit est atteint, ceci inverse la direction dans l'axe *x*. Lorsqu'un bord supérieur ou inférieur est atteint, ceci inverse la direction dans l'axe *y*.

```
// Bot se déplace en diagonale
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ExerciceJFrameAvecPingPongBot extends JFrame {

 // Constantes pour la taille de la fenêtre et du Bot
 private static final int LARGEURFENETRE = 400;
 private static final int HAUTEURFENETRE = 400;
 private static final int LARGEURBOT = LARGEURFENETRE / 4;
 private static final int HAUTEURBOT = HAUTEURFENETRE / 3;

 // Tampon pour construire l'image avant d'afficher
 Graphics tamponGraphics;
 Image tamponImage;

 public ExerciceJFrameAvecPingPongBot() {
  super("Ping pong Bot");
  this.setDefaultCloseOperation(EXIT_ON_CLOSE);
  this.setSize(LARGEURFENETRE, HAUTEURFENETRE);
  this.setVisible(true);
 }

 // Méthode qui dessine un Bot dans un objet Graphics g
 // à l'échelle dans un rectangle englobant de paramêtres x,y,largeur,hauteur
 public void paintBot(Graphics g, int x, int y, int largeur, int hauteur) {
  g.setColor(Color.green);
  g.fillOval(x, y, largeur, hauteur / 2); // La tête

  g.setColor(Color.black);
  g.fillRect(x + largeur / 4, y + hauteur / 8, largeur / 10, hauteur / 20); // L'oeil gauche
  g.fillRect(
     x + largeur * 3 / 4 - largeur / 10,
     y + hauteur / 8,
     largeur / 10,
     hauteur / 20); // L'oeil droit
  g.drawLine(
     x + largeur / 4,
     y + hauteur * 3 / 8,
     x + largeur * 3 / 4,
     y + hauteur * 3 / 8); // La bouche

  g.setColor(Color.red);
```

```
    g.fillRect(x, y + hauteur / 2, largeur, hauteur / 2); // Le corps
  }

  public void paint(Graphics g) {
    super.paint(g);
    tamponImage = createImage(LARGEURFENETRE, HAUTEURFENETRE);
    tamponGraphics = tamponImage.getGraphics();

    int xBot = 0; // Coordonnées du Bot
    int yBot = 0;
    int vitesseXBot = 5; // Vitesse du Bot
    int vitesseYBot = 10;

    while (true) {
      // Dessine le Bot dans le tampon
      paintBot(tamponGraphics, xBot, yBot, LARGEURBOT, HAUTEURBOT);
      // Copie le tampon dans le contexte graphique de la fenetre
      g.drawImage(tamponImage, 0, 0, this);
      try {
        Thread.sleep(50);
      } catch (InterruptedException uneException) {
        System.out.println(uneException.toString());
      }
      // Efface le Bot
      tamponGraphics.clearRect(xBot, yBot, LARGEURBOT, HAUTEURBOT);
      // Déplace le Bot
      if (xBot + LARGEURBOT >= LARGEURFENETRE | xBot < 0) // Si atteint le bord
      vitesseXBot = -vitesseXBot; // Inverser la direction selon x
      xBot = xBot + vitesseXBot; // Déplacement du Bot selon x
      if (yBot + HAUTEURBOT >= HAUTEURFENETRE | yBot < 0) // Si atteint le bord
      vitesseYBot = -vitesseYBot; // Inverser la direction selon y
      yBot = yBot + vitesseYBot; // Déplacement du Bot selon y
    }
  }

  public static void main(String args[]) {
    new ExerciceJFrameAvecPingPongBot();
  }
}
```

Exercice. Maintenant combinez quelques Bot et Iti dans la même animation avec des positions initiales, vitesses et tailles différentes.

Solution. JavaPasAPas/chapitre_5/ExerciceJFrameAvecPingPongBotsEtItis.java

```
// Plusieurs Bot et Iti qui bougent
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ExerciceJFrameAvecPingPongBotsEtItis extends JFrame {

  // Constantes pour la taille de la fenetre et du Bot
  private static final int LARGEURFENETRE = 400;
  private static final int HAUTEURFENETRE = 400;

  // Tampon pour construire l'image avant d'afficher
  Graphics tamponGraphics;
  Image tamponImage;
```

```
public ExerciceJFrameAvecPingPongBotsEtItis() {
  super("Ping pong Bots et Itis");
  this.setDefaultCloseOperation(EXIT_ON_CLOSE);
  this.setSize(LARGEURFENETRE, HAUTEURFENETRE);
  this.setVisible(true);
}

// Méthode qui dessine un Bot dans un objet Graphics g
// à l'échelle dans un rectangle englobant de paramètres x,y,largeur,hauteur
public void paintBot(Graphics g, int x, int y, int largeur, int hauteur) {
  g.setColor(Color.green);
  g.fillOval(x, y, largeur, hauteur / 2); // La t◆te

  g.setColor(Color.black);
  g.fillRect(x + largeur / 4, y + hauteur / 8, largeur / 10, hauteur / 20); // L'oeil gauche
  g.fillRect(
      x + largeur * 3 / 4 - largeur / 10,
      y + hauteur / 8,
      largeur / 10,
      hauteur / 20); // L'oeil droit
  g.drawLine(
      x + largeur / 4,
      y + hauteur * 3 / 8,
      x + largeur * 3 / 4,
      y + hauteur * 3 / 8); // La bouche

  g.setColor(Color.red);
  g.fillRect(x, y + hauteur / 2, largeur, hauteur / 2); // Le corps
}

// Méthode qui dessine un Iti dans un objet Graphics g
// à l'échelle dans un rectangle englobant de paramètres x,y,largeur,hauteur
// Méthode qui dessine un Iti dans un objet Graphics g
// à l'échelle dans un rectangle englobant de paramètres x,y,largeur,hauteur
// Méthode qui dessine un Iti dans un objet Graphics g
// à l'échelle dans un rectangle englobant de paramètres x,y,largeur,hauteur
public static void paintIti(Graphics g, int x, int y, int largeur, int hauteur) {
  // Coordonnées du milieu du rectangle englobant pour faciliter les calculs
  int milieux = x + largeur / 2;
  int milieuy = y + hauteur / 2;
  // La tête
  g.setColor(Color.pink);
  g.fillOval(x + largeur / 3, y, largeur / 3, hauteur / 4);
  // Le sourire
  g.setColor(Color.black);
  g.drawArc(x + largeur / 3, y - hauteur / 12, largeur / 3, hauteur / 4, -125, 70);
  // Les yeux
  g.fillOval(milieux - largeur / 8, y + hauteur / 12, largeur / 12, hauteur / 24);
  g.fillOval(milieux + largeur / 8 - largeur / 12, y + hauteur / 12, largeur / 12, hauteur / 24);
  // Le corps
  g.drawLine(milieux, y + hauteur / 4, milieux, y + hauteur * 3 / 4);
  // Les bras
  g.drawLine(x + 1, y + hauteur / 4, milieux, milieuy);
  g.drawLine(x + largeur - 1, y + hauteur / 4, milieux, milieuy);
  // Les jambes
  g.drawLine(x + 1, y + hauteur - 1, milieux, y + hauteur * 3 / 4);
  g.drawLine(x + largeur - 1, y + hauteur - 1, milieux, y + hauteur * 3 / 4);
}

public void paint(Graphics g) {
  super.paint(g);
  tamponImage = createImage(LARGEURFENETRE, HAUTEURFENETRE);
  tamponGraphics = tamponImage.getGraphics();

  int xBot1 = 0; // Coordonnées du Bot1
```

```
int yBot1 = 100;
int vitesseXBot1 = 5; // Vitesse du Bot1
int vitesseYBot1 = 0;
int largeurBot1 = 100; // Taille du Bot1
int hauteurBot1 = 150;

int xBot2 = 100; // Coordonnées du Bot2
int yBot2 = 100;
int vitesseXBot2 = -10; // Vitesse du Bot2
int vitesseYBot2 = 5;
int largeurBot2 = 75; // Taille du Bot2
int hauteurBot2 = 100;

int xIti1 = 200; // Coordonnées du Iti1
int yIti1 = 300;
int vitesseXIti1 = 6; // Vitesse du Iti1
int vitesseYIti1 = 6;
int largeurIti1 = 80; // Taille du Iti1
int hauteurIti1 = 80;

int xIti2 = 200; // Coordonnées du Iti2
int yIti2 = 0;
int vitesseXIti2 = 0; // Vitesse du Iti2
int vitesseYIti2 = 10;
int largeurIti2 = 50; // Taille du Iti2
int hauteurIti2 = 50;

while (true) {
 // Dessine les Bot et Iti
 paintBot(tamponGraphics, xBot1, yBot1, largeurBot1, hauteurBot1);
 paintBot(tamponGraphics, xBot2, yBot2, largeurBot2, hauteurBot2);
 paintIti(tamponGraphics, xIti1, yIti1, largeurIti1, hauteurIti1);
 paintIti(tamponGraphics, xIti2, yIti2, largeurIti2, hauteurIti2);
 // Copie le tampon dans le contexte graphique de la fenetre
 g.drawImage(tamponImage, 0, 0, this);

 try {
  Thread.sleep(50);
 } catch (InterruptedException uneException) {
  System.out.println(uneException.toString());
 }
 // Efface les Bot et Iti
 tamponGraphics.clearRect(xBot1, yBot1, largeurBot1, hauteurBot1);
 tamponGraphics.clearRect(xBot2, yBot2, largeurBot2, hauteurBot2);
 tamponGraphics.clearRect(xIti1, yIti1, largeurIti1, hauteurIti1);
 tamponGraphics.clearRect(xIti2, yIti2, largeurIti2, hauteurIti2);

 // Déplace le Bot1
 if (xBot1 + largeurBot1 >= LARGEURFENETRE | xBot1 < 0) // Si atteint le bord
 vitesseXBot1 = -vitesseXBot1; // Inverser la direction selon x
 xBot1 = xBot1 + vitesseXBot1; // Déplacement du Bot selon x
 if (yBot1 + hauteurBot1 >= HAUTEURFENETRE | yBot1 < 0) // Si atteint le bord
 vitesseYBot1 = -vitesseYBot1; // Inverser la direction selon y
 yBot1 = yBot1 + vitesseYBot1; // Déplacement du Bot selon y

 // Déplace le Bot2
 if (xBot2 + largeurBot2 >= LARGEURFENETRE | xBot2 < 0) // Si atteint le bord
 vitesseXBot2 = -vitesseXBot2; // Inverser la direction selon x
 xBot2 = xBot2 + vitesseXBot2; // Déplacement du Bot selon x
 if (yBot2 + hauteurBot2 >= HAUTEURFENETRE | yBot2 < 0) // Si atteint le bord
 vitesseYBot2 = -vitesseYBot2; // Inverser la direction selon y
 yBot2 = yBot2 + vitesseYBot2; // Déplacement du Bot selon y

 // Déplace le Iti1
 if (xIti1 + largeurIti1 >= LARGEURFENETRE | xIti1 < 0) // Si atteint le bord
 vitesseXIti1 = -vitesseXIti1; // Inverser la direction selon x
```

```
    xIti1 = xIti1 + vitesseXIti1; // Déplacement du Iti selon x
    if (yIti1 + hauteurIti1 >= HAUTEURFENETRE | yIti1 < 0) // Si atteint le bord
    vitesseYIti1 = -vitesseYIti1; // Inverser la direction selon y
    yIti1 = yIti1 + vitesseYIti1; // Déplacement du Iti selon y
    // Déplace le Iti2
    if (xIti2 + largeurIti2 >= LARGEURFENETRE | xIti2 < 0) // Si atteint le bord
    vitesseXIti2 = -vitesseXIti2; // Inverser la direction selon x
    xIti2 = xIti2 + vitesseXIti2; // Déplacement du Iti selon x
    if (yIti2 + hauteurIti2 >= HAUTEURFENETRE | yIti2 < 0) // Si atteint le bord
    vitesseYIti2 = -vitesseYIti2; // Inverser la direction selon y
    yIti2 = yIti2 + vitesseYIti2; // Déplacement du Iti selon y
   }
  }

  public static void main(String args[]) {
   new ExerciceJFrameAvecPingPongBotsEtItis();
  }
}
```

Le programme de l'exercice précédent est assez compliqué et les possibilités d'erreurs de codage se multiplient ! Le prochain chapitre montre comment mieux organiser le programme en exploitant de manière judicieuse la notion d'objet et de classe Java.

6. Développement de classes : conception objet

La *conception objet* désigne le problème de conception d'un programme en utilisant des objets. Un aspect particulièrement important à considérer est le découpage du programme en classes. Ce chapitre passe en revue quelques notions de base de la conception objet : diviser pour régner, encapsulation, interface, cohésion, couplage, et relation d'héritage. Les principes de Java touchant à l'organisation, la compilation et l'exécution d'un programme composé de plusieurs classes sont aussi abordés.

6.1 Découpage d'un programme en classes

Une manière typique de faciliter le développement d'un programme Java complexe est de le découper en plusieurs classes. C'est le principe de la tarte.

Génie logiciel : principe de la tarte (diviser pour régner)

Si la tarte est trop grosse pour être gobée d'un coup, il est préférable de la couper en morceaux[25].

La notion de classe en programmation objet permet de tirer profit de ce principe d'une manière très efficace en regroupant ensemble les variables et méthodes qui sont fortement liées.

Exemple. L'exemple suivant opère une réorganisation du programme du dernier exercice du chapitre précédent avec plusieurs Bot et Iti. Il est recommandé d'étudier la solution *ExerciceJFrameAvecPingPongBotsEtItis* avant de poursuivre ! Plutôt que de tout mettre dans une classe *ExerciceJFrameAvecPingPongBotsEtItis*, deux nouvelles classes sont créées, une pour les Bot, appelée *BotRebondissant*, et une autre pour les Iti, *ItiRebondissant*. Chacun des Bot à animer sera représenté dans le programme par un objet de la classe *BotRebondissant*, et de même pour les Iti.

Voici le code de la classe *BotRebondissant*. Rappelons que l'absence de *extends* est équivalente à *extends java.lang.Object*.

[25] Ce principe est aussi appelé « diviser pour régner ». D'autre part, si le programmeur est une tarte, il est préférable de le changer plutôt que de le découper en morceaux …

JavaPasAPas/chapitre_6/BotRebondissant.java

```
import java.awt.*;

public class BotRebondissant {
  // Variables d'objet qui décrivent l'état d'un objet BotRebondissant
  private int x, y; // Coordonnées x du Bot
  private int largeur, hauteur; // Taille du Bot
  private int vitesseX; // Vitesse de déplacement dans l'axe x
  private int vitesseY; // Vitesse de déplacement dans l'axe y

  // Constructeur pour initialiser l'état du BotRebondissant
  public BotRebondissant(int x, int y, int largeur, int hauteur, int vitesseX, int vitesseY) {
    this.x = x;
    this.y = y;
    this.hauteur = hauteur;
    this.largeur = largeur;
    this.vitesseX = vitesseX;
    this.vitesseY = vitesseY;
  }

  // déplacement pour la prochaine itération
  public void deplacer(int largeurFenetre, int hauteurFenetre) {
    if (x + largeur >= largeurFenetre | x < 0) // Si atteint le bord selon x
    vitesseX = -vitesseX; // Inverser la direction selon x
    x = x + vitesseX; // déplacement selon x
    if (y + hauteur >= hauteurFenetre | y < 0) // Si atteint le bord selon y
    vitesseY = -vitesseY; // Inverser la direction selon y
    y = y + vitesseY; // déplacement selon y
  }

  // Dessin du Bot
  public void paint(Graphics g) {
    g.setColor(Color.green);
    g.fillOval(x, y, largeur, hauteur / 2); // La tête

    g.setColor(Color.black);
    g.fillRect(x + largeur / 4, y + hauteur / 8, largeur / 10, hauteur / 20); // L'oeil gauche
    g.fillRect(
        x + largeur * 3 / 4 - largeur / 10,
        y + hauteur / 8,
        largeur / 10,
        hauteur / 20); // L'oeil droit
    g.drawLine(
        x + largeur / 4,
        y + hauteur * 3 / 8,
        x + largeur * 3 / 4,
        y + hauteur * 3 / 8); // La bouche

    g.setColor(Color.red);
    g.fillRect(x, y + hauteur / 2, largeur, hauteur / 2); // Le corps
  }

  // Effacer le rectangle du Bot dans tamponGraphics
  public void effacer(Graphics tamponGraphics) {
    tamponGraphics.clearRect(x, y, largeur, hauteur);
  }
}
```

La classe *BotRebondissant* contient les variables et méthodes qui concernent un objet Bot. Les variables qui décrivent un objet Bot (coordonnées, taille et vitesse) sont :

```
private int x,y;              // Coordonnées x du Bot
private int largeur,hauteur;  // Taille du Bot
private int vitesseX;         // Vitesse de déplacement dans l'axe x
private int vitesseY;         // Vitesse de déplacement dans l'axe y
```

État d'un objet

Les variables d'objet représentent l'*état* d'un objet. Un objet évolue en passant d'un état à un autre suite aux modifications apportées aux variables de l'objet.

Le constructeur *BotRebondissant*() initialise les variables d'objet. Ici, le constructeur ne fait qu'assigner les valeurs des paramètres aux variables d'objet correspondantes.

```
// Constructeur pour initialiser l'état du BotRebondissant
public BotRebondissant(
  int x, int y, int largeur, int hauteur,
  int vitesseX,int vitesseY) {
    this.x = x; this.y = y;
    this.hauteur = hauteur; this.largeur = largeur;
    this.vitesseX = vitesseX; this.vitesseY = vitesseY;
}
```

Identificateur réservé *this* suivi d'un nom de variable

Le *this* suivi d'un nom de variable désigne la variable d'objet correspondante.

Le *this* est nécessaire ici pour distinguer le paramètre *x* et la variable d'objet *x* car ils portent le même nom ! Il en est de même pour les autres variables.

En plus du constructeur, les méthodes particulières à un objet *BotRebondissant* sont *deplacer*(), *paint*() et *effacer*(). Ce sont des méthodes d'objet car elles accèdent aux variables d'objet de la classe.

Comportement d'un objet

Les méthodes représentent le comportement d'un objet. Elles décrivent les opérations applicables à un objet de la classe.

La méthode *deplacer*() décrit le comportement de déplacement d'un *BotRebondissant* pour passer à la scène suivante.

```
// Déplacement pour la prochaine itération
```

```
    public void deplacer(int largeurFenetre, int hauteurFenetre){
      if (x+largeur>=largeurFenetre | x < 0) // Si atteint le bord
selon x
        vitesseX = -vitesseX; // Inverser la direction selon x
      x = x + vitesseX; // Déplacement selon x
      if (y+hauteur>=hauteurFenetre | y < 0) // Si atteint le bord
selon y
        vitesseY = -vitesseY; // Inverser la direction selon y
      y = y + vitesseY; // Déplacement selon y
    }
```

La méthode utilise toutes les variables d'objet de la classe (*x*, *y*, *largeur*, *hauteur*, *vitesseX* et *vitesseY*). Il est à noter que *largeurFenetre* et *hauteurFenetre* sont passés en paramètre à *deplacer*().

La méthode *paint*() utilise les variables d'objet *x*, *y*, *largeur* et *hauteur* qui ne sont plus passées en paramètre par opposé à la version sans classes. La méthode *effacer*() utilise les mêmes variables.

La classe *ItiRebondissant* est semblable. La différence avec *BotRebondissant* est la méthode *paint*().

JavaPasAPas/chapitre_6/Bonhommes/ItiRebondissant.java

```
import java.awt.*;

public class ItiRebondissant {
  // Variables d'objet qui décrivent l'état d'un objet ItiRebondissant
  private int x, y; // Coordonnées x du Bot
  private int largeur, hauteur; // Taille du Bot
  private int vitesseX; // Vitesse de déplacement dans l'axe x
  private int vitesseY; // Vitesse de déplacement dans l'axe y

  // Constructeur pour initialiser l'état du ItiRebondissant
  public ItiRebondissant(int x, int y, int largeur, int hauteur, int vitesseX, int vitesseY) {
    this.x = x;
    this.y = y;
    this.hauteur = hauteur;
    this.largeur = largeur;
    this.vitesseX = vitesseX;
    this.vitesseY = vitesseY;
  }

  // Déplacement pour la prochaine itération
  public void deplacer(int largeurFenetre, int hauteurFenetre) {
    if (x + largeur >= largeurFenetre | x < 0) // Si atteint le bord selon x
    vitesseX = -vitesseX; // Inverser la direction selon x
    x = x + vitesseX; // Déplacement selon x
    if (y + hauteur >= hauteurFenetre | y < 0) // Si atteint le bord selon y
    vitesseY = -vitesseY; // Inverser la direction selon y
    y = y + vitesseY; // Déplacement selon y
  }

  // Dessin du Iti
  public void paint(Graphics g) {
    // Coordonnées du milieu du rectangle englobant pour faciliter les calculs
    int milieux = x + largeur / 2;
    int milieuy = y + hauteur / 2;
```

```
    // La tête
    g.setColor(Color.pink);
    g.fillOval(x + largeur / 3, y, largeur / 3, hauteur / 4);
    // Le sourire
    g.setColor(Color.black);
    g.drawArc(x + largeur / 3, y - hauteur / 12, largeur / 3, hauteur / 4, -125, 70);
    // Les yeux
    g.fillOval(milieux - largeur / 8, y + hauteur / 12, largeur / 12, hauteur / 24);
    g.fillOval(milieux + largeur / 8 - largeur / 12, y + hauteur / 12, largeur / 12, hauteur / 24);
    // Le corps
    g.drawLine(milieux, y + hauteur / 4, milieux, y + hauteur * 3 / 4);
    // Les bras
    g.drawLine(x + 1, y + hauteur / 4, milieux, milieuy);
    g.drawLine(x + largeur - 1, y + hauteur / 4, milieux, milieuy);
    // Les jambes
    g.drawLine(x + 1, y + hauteur - 1, milieux, y + hauteur * 3 / 4);
    g.drawLine(x + largeur - 1, y + hauteur - 1, milieux, y + hauteur * 3 / 4);
  }

  // Effacer le rectangle du Iti dans tamponGraphics
  public void effacer(Graphics tamponGraphics) {
    tamponGraphics.clearRect(x, y, largeur, hauteur);
  }
}
```

Finalement, voici le code de la classe de la fenêtre *ExempleJFrameAvecClassesPourBotEtIti* qui utilise les deux classes précédentes.

JavaPasAPas/chapitre_6/ExempleJFrameAvecClassesPourBotEtIti.java

```
// Plusieurs Bot et Iti qui bougent
// Utilise les classes BotRebondissant et ItiRebondissant du package Bonhommes
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class ExempleJFrameAvecClassesPourBotEtIti extends JFrame {

   // Constantes pour la taille de la fenetre et du Bot
   private static final int LARGEURFENETRE = 400;
   private static final int HAUTEURFENETRE = 400;

   // Tampon pour construire l'image avant d'afficher
   Graphics tamponGraphics;
   Image tamponImage;


   public ExempleJFrameAvecClassesPourBotEtIti() {
      super("Ping pong avec classes pour Bot et Iti");
      this.setDefaultCloseOperation(EXIT_ON_CLOSE);
      this.setSize(LARGEURFENETRE,HAUTEURFENETRE);
      this.setVisible(true);
   }

   public void paint (Graphics g) {
      tamponImage = createImage(LARGEURFENETRE,HAUTEURFENETRE);
      tamponGraphics = tamponImage.getGraphics();

      Bonhommes.BotRebondissant bot1 = new Bonhommes.BotRebondissant(0,100,100,150,5,0);
      Bonhommes.BotRebondissant bot2 = new Bonhommes.BotRebondissant(100,100,75,100,-10,5);
      Bonhommes.ItiRebondissant iti1 = new Bonhommes.ItiRebondissant(200,300,80,80,6,6);
      Bonhommes.ItiRebondissant iti2 = new Bonhommes.ItiRebondissant(200,0,50,50,0,10);
```

```
    while(true){
      // Dessine les Bot et Iti
      bot1.paint(tamponGraphics); bot2.paint(tamponGraphics);
      iti1.paint(tamponGraphics); iti2.paint(tamponGraphics);

      //Copie le tampon dans le contexte graphique de la fenetre
      g.drawImage(tamponImage,0,0,this);
      try {Thread.sleep(50);}
      catch(InterruptedException uneException){
        System.out.println(uneException.toString());
      }
      // Efface les Bot et Iti du tampon
      bot1.effacer(tamponGraphics); bot2.effacer(tamponGraphics);
      iti1.effacer(tamponGraphics); iti2.effacer(tamponGraphics);

      // Déplace les Bot et Iti
      bot1.deplacer(LARGEURFENETRE, HAUTEURFENETRE);
      bot2.deplacer(LARGEURFENETRE, HAUTEURFENETRE);
      iti1.deplacer(LARGEURFENETRE, HAUTEURFENETRE);
      iti2.deplacer(LARGEURFENETRE, HAUTEURFENETRE);
    }
  }

  public static void main (String args[]) {
    new ExempleJFrameAvecClassesPourBotEtIti();
  }
}
```

La classe *ExempleJFrameAvecClassesPourBotEtIti* utilise les classes *BotRebondissant* et *ItiRebondissant*. On dit que la classe *fait appel aux services* fournis par les classes *BotRebondissant* et *ItiRebondissant*. Le diagramme de Figure 24 montre comment représenter le fait que *ExempleJFrameAvecClassesPourBotEtIti* utilise les classes *BotRebondissant* et *ItiRebondissant* par une flèche pointillée appelée relation de dépendance en UML. Le diagramme montre aussi comment représenter les attributs et les méthodes dans un diagramme de classe UML. Les attributs apparaissent dans le deuxième sous-rectangle à l'intérieur du rectangle de la classe et les méthodes dans le troisième. Le nom d'un attribut peut être suivi de : et du type de l'attribut. Le nom d'une méthode est suivi optionnellement de ses paramètres. Le souligné désigne une propriété (attribut ou méthode) de classe (*static* en Java). Le symbole - avant une propriété correspond à la visibilité *private* de Java et le +, à *public*. Lorsque le nombre de classes devient important, un tel diagramme permet d'en faciliter la compréhension.

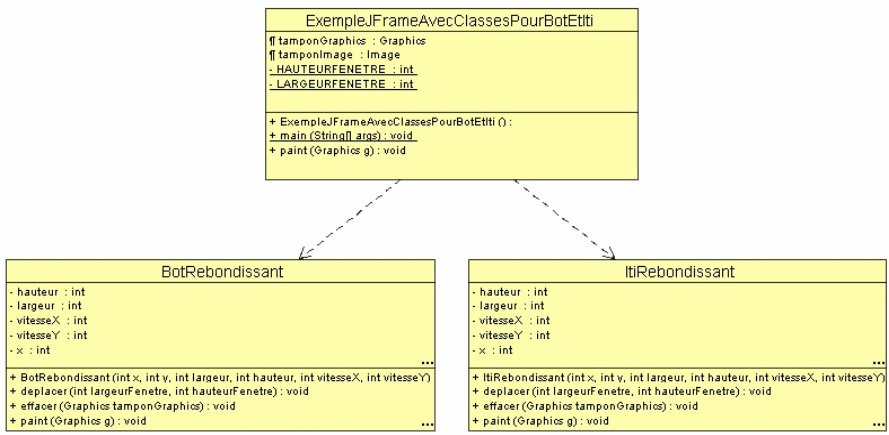


Figure 24. Diagramme UML des classes.

Tous les détails du fonctionnement des Bot et Iti sont cachés dans les classes *BotRebondissant* et *ItiRebondissant* du point de vue de la classe *ExempleJFrameAvecClassesPourBotEtIti*.

***Encapsulation*, *interface programmatique* (*Application Programming Interface* - API), service, client**

Cette manière d'isoler une classe de détails d'une autre classe est une caractéristique de la programmation objet appelée l'*encapsulation*. Tout ce que la classe *ExempleJFrameAvecClassesPourBotEtIti* doit savoir, c'est comment appeler les méthodes appropriées de la classe *BotRebondissant* (*BotRebondissant*(), *deplacer*(), *paint*()). Elle n'a pas besoin de comprendre comment cela se passe à l'intérieur des méthodes appelées. Ainsi la classe *BotRebondissant* fournit une abstraction de mécanismes complexes sous forme d'un ensemble de méthodes simples à appeler. Dans le langage objet, cet ensemble de méthode est appelé une *interface programmatique*. On dit aussi que la classe *BotRebondissant* fournit un *service* à la classe *ExempleJFrameAvecClassesPourBotEtIti* qui est le *client* de ce service. Le client voit l'interface programmatique mais pas la mise en œuvre.

Principes de génie logiciel : cohésion forte et couplage faible entre classes

Une question fondamentale au cœur de la conception d'un programme objet est la manière de répartir les variables et méthodes entre les classes. Deux principes fondamentaux sont de chercher à maximiser la cohésion des classes et de minimiser le couplage entre les classes. La *cohésion* à l'intérieur d'une classe est forte lorsque les variables et méthodes de la classe sont fortement liées. Le couplage entre deux classes est *faible* lorsqu'il y a peu de dépendances entre les classes. Concrètement la dépendance entre classes est déterminée par l'utilisation d'une autre classe en passant par des déclarations, utilisation des variables ou appels de méthodes et passage de paramètres.

Dans notre exemple, les méthodes *deplacer*(), *paint*() et *effacer*() utilisent toutes une grande proportion des variables de classe *BotRebondissant*, ce qui est un signe de grande cohésion de cette classe.

D'autre part, le fait de passer les valeurs *largeurFenetre* et *hauteurFenetre* en paramètres à *deplacer*() est un indice de couplage entre *ExempleJFrameAvecClassesPourBotEtIti* et *BotRebondissant*/*ItiRebondissant* car cette méthode est appelée dans *paint*() de *ExempleJFrameAvecClassesPourBotEtIti*.

Minimiser le couplage signifie, entre autres, de chercher à réduire le nombre de paramètres passés lorsque cela est approprié. Par exemple, une possibilité serait de faire des variables *largeurFenetre* et *hauteurFenetre*, des variables d'objet des classes *BotRebondissant*/*ItiRebondissant*. Comme avantage, on pourrait éviter de passer ces paramètres à chacun des appels à *deplacer*(). Comme inconvénient, il faudrait répéter la même information dans chacun des objets de *BotRebondissant*/*ItiRebondissant*. Généralement, le fait de répéter la même information à plusieurs endroits est à éviter. Pour limiter la répétition des données entre les objets, il serait approprié d'en faire des variables de classe. Ceci éviterait la répétition dans chacun des objets concernés mais il y aurait tout de même une répétition car les données seraient répétées dans les classes *BotRebondissant* et *ItiRebondissant*, et *ExempleJFrameAvecClassesPourBotEtIti*. Cet exemple illustre des enjeux subtils de la conception d'un programme objet.

Pour trouver la solution appropriée, d'une manière abstraite, il faut se demander si la *largeurFenetre* et la *hauteurFenetre* sont des données plus particulièrement caractéristiques de la fenêtre ou d'un *Bot* ? Dans notre exemple, la réponse est que ce sont des caractéristiques de la fenêtre, et de ce point de vue, ces variables devraient plutôt être maintenues dans la classe

fenêtre. Cependant, comme le déplacement d'un *Bot* est contraint par la taille de la fenêtre, la méthode *deplacer*() a besoin de ces informations, d'où la nécessité de passer les paramètres à l'appel. Le choix d'en faire des paramètres de la méthode *deplacer*() est donc approprié dans notre exemple.

Note au sujet de l'initialisation des variables d'objet

Les variables d'objet ou de classe sont automatiquement initialisées avec des valeurs de défaut énumérées dans le tableau suivant.

Type	Valeur de défaut
`boolean`	`false`
`byte`	`0`
`char`	`\u0000`
`short`	`0`
`int`	`0`
`long`	`0`
`float`	`0.0`
`double`	`0.0`
`String or object`	`null`

Il est cependant préférable de ne pas se fier sur cette initialisation implicite afin de faciliter la compréhension du programme.

6.2 Compilation et exécution d'un programme composé de plusieurs classes et de packages

Un programme Java est habituellement composé de plusieurs classes. Un aspect important du développement d'un programme Java est de pouvoir localiser toutes les classes nécessaires à un programme lors de la compilation et de l'exécution. Il y a plusieurs manières de procéder.

- **Cas simple : répertoire courant sans packages**

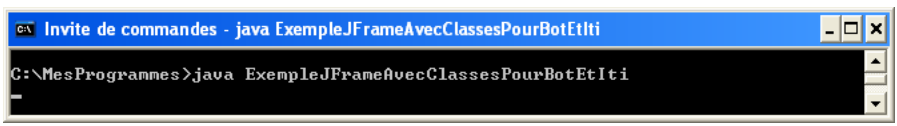
Regardons d'abord un cas simple sans packages où toutes les classes sont dans le répertoire courant.

Exemple. Prenons notre exemple précédent qui inclut trois classes *BotRebondissant* et *ItiRebondissant*, et *ExempleJFrameAvecClassesPourBotEtIti*. Une manière simple de procéder consiste à placer tous les fichiers sources (.java) dans un dossier, par exemple, le dossier *C:\MesProgrammes* et compiler le fichier qui contient la méthode *main*(), soit *ExempleJFrameAvecClassesPourBotEtIti.java*, dans notre exemple, comme pour le cas d'une seule classe :

En compilant *ExempleJFrameAvecClassesPourBotEtIti.java*, le compilateur va automatiquement rechercher les fichiers sources des classes utilisées par la classe *ExempleJFrameAvecClassesPourBotEtIti.java* soit, *BotRebondissant.java ItiRebondissant.java*, dans notre exemple. La recherche est effectuée de la même manière que pour une seule classe. Si la variable d'environnement *classpath* contient le dossier courant[26], le compilateur regarde dans C:\MesProgrammes.

Si ces fichiers source ne sont pas déjà compilés ou si le code compilé (*.class*) n'est pas à jour, le compilateur compile les fichiers source automatiquement et les place dans le même dossier C:\MesProgrammes. Dans notre exemple, les fichiers *BotRebondissant.java* et *ItiRebondissant.java* seraient compilés.

Ensuite, il est possible d'exécuter le fichier *.class* de la classe qui contient le *main*() comme si ce n'était qu'une seule classe.



Le même principe pour la recherche des classes est applicable à l'exécution. En exécutant *ExempleJFrameAvecClassesPourBotEtIti.class*, la JVM va automatiquement rechercher les fichiers *.class* des classes utilisées par

[26] Rappelons que le «.» dans le classpath représente le dossier courant

ExempleJFrameAvecClassesPourBotEtIti.class soit, *BotRebondissant.class ItiRebondissant.class*, dans notre exemple. Si la variable *classpath* contient le dossier courant, le compilateur regarde dans C:\MesProgrammes.

- **Cas général incluant des packages**

Les outils Java incorporent des moyens très flexibles pour la recherche des fichiers. Lorsque le compilateur recherche un fichier, et que le chemin complet n'est pas donné, la recherche se fait de manière relative à un ensemble de répertoires appelé *répertoires racines*. Les répertoires racines sont recherchés dans l'ordre suivant :

1. *Bootclasspath* et *extension directories*. Ces répertoires sont fixés lors de l'installation de JSE. Ils peuvent être modifiés par des options lors de l'appel du compilateur. Consultez la documentation de JSE pour le détail des options du compilateur.

On retrouve dans ces répertoires les packages prédéfinies de Java (java, javax, …). Normalement, ces répertoires ne sont pas utilisés pour les classes du programme d'application en développement.

2. Le *classpath* de l'utilisateur (*user classpath*). C'est ici que le programmeur place les fichiers de ses classes. Ce *classpath* peut être fixé de trois manières.

    a. Par l'option *-classpath* du compilateur

    b. Sinon, par la variable d'environnement *classpath*

    c. Sinon, c'est le répertoire courant

Si une classe est dans un package, la recherche est plus compliquée. Supposons que le compilateur recherche une classe en utilisant le répertoire racine *C:\MesProgrammes* qui est en l'occurrence le répertoire courant. Si le fichier recherché, appelée *Classe1.java* est dans le package *p1.p2.p3* (c'est-à-dire que le nom complet de la classe est *p1.p2.p3.Classe1*), le compilateur recherche *Classe1.java* dans le répertoire suivant :

```
C:\MesProgrammes\p1\p2\p3
```

En d'autres mots, le répertoire racine est utilisé comme point de départ de la recherche et chacun des packages Java correspond effectivement à une suite de répertoires imbriqués sous le répertoire racine. Ainsi le chemin complet du fichier est formé par la concaténation d'un répertoire racine avec la séquence des répertoires correspondant à la séquence des noms de packages, suivi du nom du fichier lui-même. Ceci implique que le programmeur doit créer les répertoires dont le nom correspond au nom du package et y placer les fichiers appropriés.

Cet aspect de la programmation Java devient rapidement très fastidieux et sujet à erreur. C'est pourquoi, lorsque la complexité des programmes augmente et conduit à l'utilisation de packages pour organiser les classes, il est intéressant d'employer un environnement de développement intégré qui s'occupe automatiquement de la gestion des répertoires correspondant aux packages.

Enfin, une autre option peut être employée. Plutôt que de rechercher dans le système de gestion de fichier, la recherche peut se faire directement à l'intérieur d'un fichier d'archive *jar* ou *zip* (extension *.jar* ou *.zip*) qui contient la structure de répertoire des packages.

D'autre part, plutôt que de placer les fichiers compilés (.class) dans le même répertoire que les fichiers sources (.java), il est possible de spécifier un autre répertoire racine pour les fichiers compilés par l'option -d du compilateur. Ainsi, on obtient deux structures de répertoire identiques l'une pour les fichiers sources (.java), l'autre pour les fichiers compilés (.class). Le compilateur crée de nouveaux répertoires au besoin pour y placer le code compilé.

Le même principe est applicable dans le cas de la recherche de fichiers *.class* lors de l'exécution avec *java*.

Exemple. JavaPasAPas/Bonhommes/

Plaçons les deux classes *BotRebondissant* et *ItiRebondissant* dans le package *Bonhommes*. À cet effet, il faut d'une part ajouter un énoncé *package* dans chacun des fichiers *BotRebondissant.java* et *ItiRebondissant.java*. Cet énoncé est placé su début du code Java :

```
package Bonhommes;
```

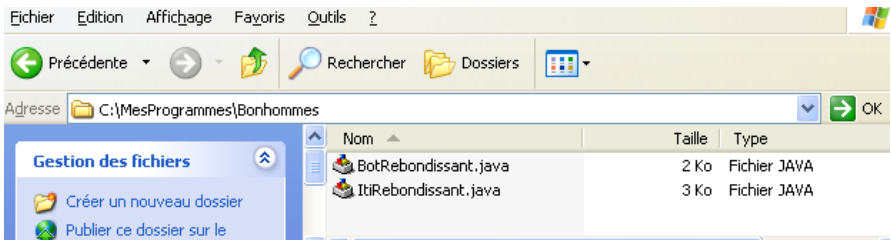
```
import java.awt.*;
public class BotRebondissant {

…
}
```

```
package Bonhommes;
import java.awt.*;
public class ItiRebondissant {

…
}
```

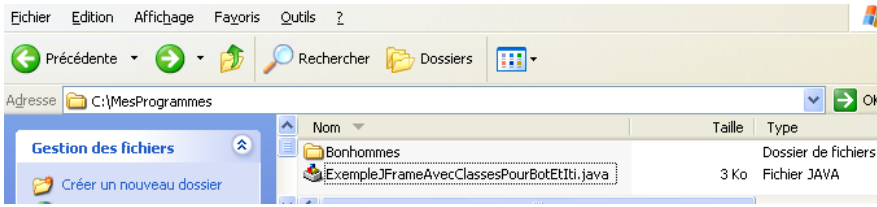
Dans la classe *ExempleJFrameAvecClassesPourBotEtIti*, on peut ajouter un énoncé *import* :

```
// Plusieurs Bot et Iti qui bougent
// Utilise les classes BotRebondissant et ItiRebondissant du package
Bonhommes
import Bonhommes.*;
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class ExempleJFrameAvecClassesPourBotEtIti extends JFrame {
…
}
```

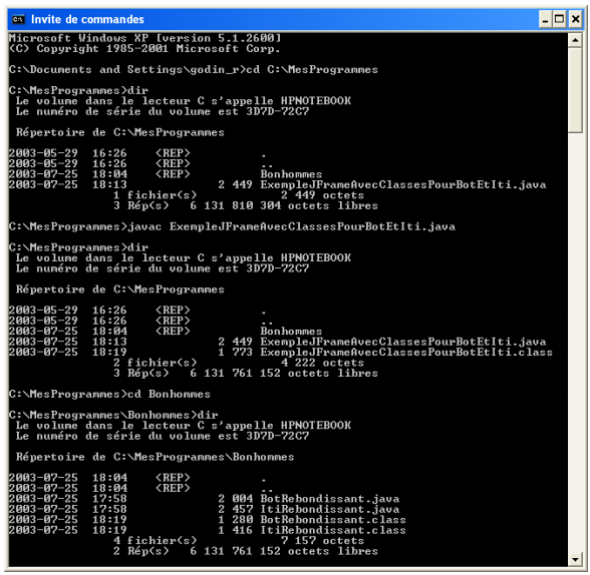
Il faut placer les deux fichiers *BotRebondissant.java* et *ItiRebondissant.java* dans un répertoire appelé aussi *Bonhommes*.



Si le répertoire racine est *C:\MesProgrammes*, il faut placer le répertoire *Bonhommes* sous le répertoire *C:\MesProgrammes* :



La figure suivante montre un scénario de compilation. On peut voir que le compilateur a produit le fichier *ExempleJFrameAvecClassesPourBotEtIti.class* dans le répertoire *C:\MesProgrammes* comme dans le cas simple. Maintenant, les fichiers compilés *BotRebondissant.class* et *ItiRebondissant.class* ont été placés dans *C:\MesProgrammes\Bonhommes*.



On exécute le programme comme dans le cas simple.

Il est à noter que plutôt que d'utiliser un énoncé *import Bonhommes.** dans *ExempleJFrameAvecClassesPourBotEtIti*, on peut spécifier le nom complet *Bonhommes.BotRebondissant* et *Bonhommes.ItiRebondissant*, incluant le nom du package en préfixe :

```
// Plusieurs Bot et Iti qui bougent
// Utilise les classes BotRebondissant et ItiRebondissant du package
Bonhommes
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

…

        Bonhommes.BotRebondissant bot1 = new
Bonhommes.BotRebondissant(0,100,100,150,5,0);
        Bonhommes.BotRebondissant bot2 = new
Bonhommes.BotRebondissant(100,100,75,100,-10,5);
        Bonhommes.ItiRebondissant iti1 = new
Bonhommes.ItiRebondissant(200,300,80,80,6,6);
        Bonhommes.ItiRebondissant iti2 = new
Bonhommes.ItiRebondissant(200,0,50,50,0,10);

…
}
```

Le nom du package peut être spécifié avant le nom de classe dans le diagramme UML tel qu'illustré à la Figure 25.

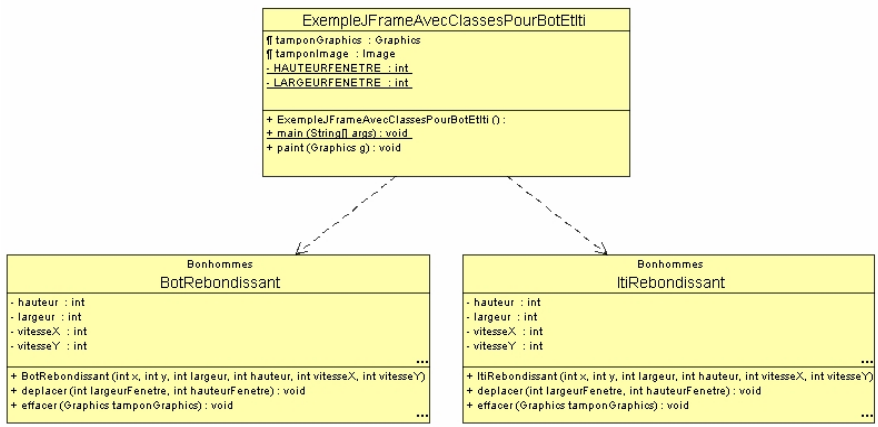


Figure 25. Package *Bonhommes*

Exercice. Ajoutez à l'exemple précédent une nouvelle classe pour votre bonhomme préféré que vous placerez aussi dans le package *Bonhommes*. Créez quelques objets de cette classe dans l'animation.

Exercice. Supposons que le *Iti* se déplace dans l'axe *z*. Coder une méthode *deplacer*() spécifique au *Iti* afin de donner cette impression. Ceci peut être effectué en modifiant la taille du *Iti* plutôt que sa position !

6.3 Limiter la répétition de code par la création d'une super-classe

L'utilisation de méthodes pour limiter la répétition des mêmes groupes d'énoncés dans le code du programme a déjà été expliquée. Un autre moyen qui permet de limiter la répétition de code est l'utilisation de super-classes.

Exemple.

Dans l'exemple précédent, il y a beaucoup de répétition de code entre les deux classes *BotRebondissant* et *ItiRebondissant*. Les variables sont les mêmes ainsi que les méthodes *deplacer*() et *effacer*(). Une manière d'éviter cette redondance de code est de créer une super-classe des deux classes qui isole les aspects communs des deux classes dans la super-classe commune. Appelons cette super-classe *EntiteRebondissante*. La classe *EntiteRebondissante* déclare les variables et méthodes communes aux deux classes *BotRebondissant* et *ItiRebondissant* de l'exemple précédent (voir Figure 26).

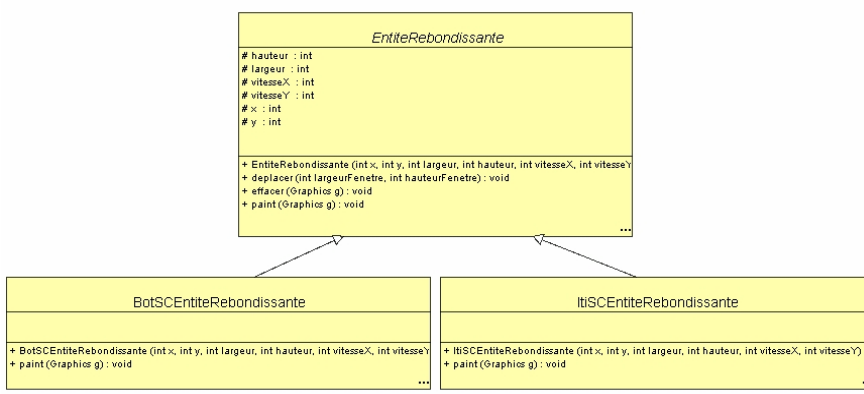


Figure 26. Abstraction des aspects communs par création d'une super-classe.

JavaPasAPas/chapitre_6/EntiteRebondissante.java

```
import java.awt.*;

public abstract class EntiteRebondissante {
 // Variables d'objet qui décrivent l'état d'un objet EntiteRebondissante
 // Protected permet aux sous-classes d'utiliser les variables
 protected int x, y; // Coordonnées x du Bot
 protected int largeur, hauteur; // Taille du Bot
 protected int vitesseX; // Vitesse de déplacement dans l'axe x
 protected int vitesseY; // Vitesse de déplacement dans l'axe y

 // Constructeur
 public EntiteRebondissante(int x, int y, int largeur, int hauteur, int vitesseX, int vitesseY) {
  this.x = x;
  this.y = y;
  this.hauteur = hauteur;
  this.largeur = largeur;
  this.vitesseX = vitesseX;
  this.vitesseY = vitesseY;
 }

 // Déplacement pour la prochaine itération
 public void deplacer(int largeurFenetre, int hauteurFenetre) {
  if (x + largeur >= largeurFenetre | x < 0) // Si atteint le bord selon x
  vitesseX = -vitesseX; // Inverser la direction selon x
  x = x + vitesseX; // Déplacement selon x
  if (y + hauteur >= hauteurFenetre | y < 0) // Si atteint le bord selon y
  vitesseY = -vitesseY; // Inverser la direction selon y
  y = y + vitesseY; // Déplacement selon y
 }

 // Méthode abstraite : corps doit être précisé dans la sous-classe concr?te
 public abstract void paint(Graphics g);

 // Effacer le rectangle dans tamponGraphics
 public void effacer(Graphics g) {
  g.clearRect(x, y, largeur, hauteur);
 }
}
```

Modifieur *abstract* pour une classe, classe abstraite, classe concrète

Le modifieur *abstract* dans la déclaration d'une classe signifie qu'il n'est pas possible de créer directement un objet de cette classe (avec *new*). La *classe* dite *abstraite* n'est utile que comme super-classe d'autres *classes* non abstraites dites *concrètes* qui, elles, sont utilisées pour créer des objets.

La classe *EntiteRebondissante* est abstraite car ce sont les sous-classes, *BotSCEntiteRebondissante* et *ItiSCEntiteRebondissante*, de la classe *EntiteRebondissante* qui seront utilisées pour créer les objets.

```
public abstract class EntiteRebondissante {
```

En UML, le modifieur *abstract* est représenté en mettant le nom en italique (voir Figure 26).

Modifieur *abstract* pour une méthode

Le modifieur abstract pour une méthode signifie que la méthode n'a pas de corps. Le corps d'une méthode abstraite doit être précisé une sous-classe concrète.

Le modifieur *abstract* de la méthode *paint*() signifie que la méthode n'a pas de corps dans la classe *EntiteRebondissante*.

```
public abstract void paint (Graphics g) ;
```

Donc, cette déclaration précise qu'une sous-classe de *EntiteRebondissante* doit pouvoir répondre à un appel de la méthode *paint*(). Cette déclaration indique que les sous-classes concrètes doivent spécifier un corps de la méthode abstraite *paint*().

Modifieur *protected* pour une variable

Le modifieur *protected* pour une variable signifie que la variable n'est pas accessible aux autres classes (comme *private*) mais en revanche, elle peut être accédée par une méthode d'une sous-classe (par opposé à *private*).

Maintenant, les classes pour *Bot* et *Iti* sont les sous-classes *BotSCEntiteRebondissante* et *ItiSCEntiteRebondissante* de la classe *EntiteRebondissante*.

```
import java.awt.*;

public class BotSCEntiteRebondissante extends EntiteRebondissante {

 // Constructeur
 public BotSCEntiteRebondissante(
    int x, int y, int largeur, int hauteur, int vitesseX, int vitesseY) {
  super(x, y, largeur, hauteur, vitesseX, vitesseY);
 }

 // Corps de la méthode abstraite héritée de la super-classe
 public void paint(Graphics g) {
  g.setColor(Color.green);
  g.fillOval(x, y, largeur, hauteur / 2); // La tête

  g.setColor(Color.black);
  g.fillRect(x + largeur / 4, y + hauteur / 8, largeur / 10, hauteur / 20); // L'oeil gauche
  g.fillRect(
     x + largeur * 3 / 4 - largeur / 10,
     y + hauteur / 8,
```

```
      largeur / 10,
      hauteur / 20); // L'oeil droit
    g.drawLine(
      x + largeur / 4,
      y + hauteur * 3 / 8,
      x + largeur * 3 / 4,
      y + hauteur * 3 / 8); // La bouche

    g.setColor(Color.red);
    g.fillRect(x, y + hauteur / 2, largeur, hauteur / 2); // Le corps
  }
}
```

Dans la déclaration de la classe, la clause *extends* indique la super-classe.

```
public class BotSCEntiteRebondissante extends EntiteRebondissante {
```

La classe *ExempleJFrameAvecSuperClassePourBotEtIti* est identique à la version sans super-classe sauf pour les noms des sous-classes *BotSCEntiteRebondissante* et *ItiSCEntiteRebondissante* qui remplacent *BotRebondissant* et *ItiRebondissant*. NB On aurait pu conserver exactement les mêmes noms pour les sous-classes. Le programme qui utilise une classe n'a pas besoin de connaître ses super-classes.

JavaPasAPas/chapitre_6/ ExempleJFrameAvecSuperClassePourBotEtIti.java

```
// Plusieurs Bot et Iti qui bougent
// Version avec super-classe EntiteRebondissante
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ExempleJFrameAvecSuperClassePourBotEtIti extends JFrame {

  // Constantes pour la taille de la fenetre et du Bot
  private static final int LARGEURFENETRE = 400;
  private static final int HAUTEURFENETRE = 400;

  // Tampon pour construire l'image avant d'afficher
  Graphics tamponGraphics;
  Image tamponImage;

  public ExempleJFrameAvecSuperClassePourBotEtIti() {
    super("Ping pong avec classes pour Bot et Iti");
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    this.setSize(LARGEURFENETRE, HAUTEURFENETRE);
    this.setVisible(true);
  }

  public void paint(Graphics g) {
    tamponImage = createImage(LARGEURFENETRE, HAUTEURFENETRE);
    tamponGraphics = tamponImage.getGraphics();
```

```
  BotSCEntiteRebondissante bot1 = new BotSCEntiteRebondissante(0, 100, 100, 150, 5, 0);
  BotSCEntiteRebondissante bot2 = new BotSCEntiteRebondissante(100, 100, 75, 100, -10, 5);
  ItiSCEntiteRebondissante iti1 = new ItiSCEntiteRebondissante(200, 300, 80, 80, 6, 6);
  ItiSCEntiteRebondissante iti2 = new ItiSCEntiteRebondissante(200, 0, 50, 50, 0, 10);

  while (true) {
    // Dessine les Bot et Iti
    bot1.paint(tamponGraphics);
    bot2.paint(tamponGraphics);
    iti1.paint(tamponGraphics);
    iti2.paint(tamponGraphics);

    // Copie le tampon dans le contexte graphique de la fenetre
    g.drawImage(tamponImage, 0, 0, this);
    try {
      Thread.sleep(50);
    } catch (InterruptedException uneException) {
      System.out.println(uneException.toString());
    }
    // Efface les Bot et Iti du tampon
    bot1.effacer(tamponGraphics);
    bot2.effacer(tamponGraphics);
    iti1.effacer(tamponGraphics);
    iti2.effacer(tamponGraphics);

    // Déplace les Bot et Iti
    bot1.deplacer(LARGEURFENETRE, HAUTEURFENETRE);
    bot2.deplacer(LARGEURFENETRE, HAUTEURFENETRE);
    iti1.deplacer(LARGEURFENETRE, HAUTEURFENETRE);
    iti2.deplacer(LARGEURFENETRE, HAUTEURFENETRE);
  }
 }

 public static void main(String args[]) {
   ExempleJFrameAvecSuperClassePourBotEtIti laFenetre =
      new ExempleJFrameAvecSuperClassePourBotEtIti();
 }
}
```

***Refactorisation de code* (*refactoring*)**

Dans l'exemple *ExempleJFrameAvecSuperClassePourBotEtIti*, le code de *ExempleJFrameAvecClassesPourBotEtIti* a été transformé en déplaçant des méthodes et variables dans une super-classe mais sans en changer le comportement. Ce processus de transformation qui préserve le même comportement est appelé une *refactorisation* du code (*refactoring*)

Exercice. Reprenez les deux exercices de la section précédente en employant la super-classe *EntiteRebondissante*.

7. Animation 2D et développement d'un jeu simple

Ce chapitre étudie le développement d'un jeu simple qui combine l'animation 2D et une interactivité de base au moyen de la souris. Ce faisant, l'approche d'animation introduite au chapitre 6 sera raffinée. D'autre part, les acteurs seront raffinés en ajoutant des sons et des gestes. Le code des acteurs de notre animation sera réorganisé d'une manière plus cohérente en regroupant les éléments qui touche au monde du jeu. Ceci permet de réutiliser les classes du jeu dans une variété de contextes.

7.1 Animation avec un *Timer* dans une sous-classe de *JPanel*

Dans les chapitres précédents, une boucle explicite contrôle l'animation et le principe du double tampon est exploité pour la production de la séquence de scènes d'une animation. Ce mécanisme d'animation est raffiné ici en utilisant de manière plus judicieuse les classes de Java.

Plutôt que d'employer une boucle pour produire la séquence des scènes de l'animation, un objet de la classe *javax.swing*.**Timer** va générer un événement à intervalles réguliers afin de déclencher la production de la prochaine scène. Cet événement provoque l'appel d'une méthode prédéfinie, la méthode *actionPerformed*() qui servira à dessiner la prochaine scène et à préparer la scène suivante. L'objet **Timer** joue ainsi le rôle d'un chronomètre d'animation. Un avantage d'un chronomètre d'animation est de mieux contrôler l'intervalle de temps entre l'affichage des scènes en le rendant indépendant du temps de calcul nécessaire à la production de la prochaine scène.

Note : paramétrage de la prochaine scène par le délai écoulé

Une autre approche de contrôle de la séquence d'animation serait de paramétrer la prochaine scène (e.g. le déplacement des objets) en fonction du délai écoulé entre les scènes évitant ainsi de dépendre d'un délai fixé à l'avance.

D'autre part, plutôt que d'exploiter directement une sous-classe de **JFrame**, une sous-classe de javax.swing.**JPanel** contiendra le contexte graphique de l'animation. Ceci a comme avantage de favoriser la réutilisabilité de

l'animation car l'objet **JPanel** peut être inclus dans diverses formes d'interfaces graphiques tel qu'une fenêtre **JFrame**, ou encore une application Web.

Exemple. L'exemple suivant reprend l'animation du chapitre précédent avec plusieurs Bot et Iti. La classe *ExempleJPanelAvecAnimationParTimer* est une sous-classe de *javax.swing.* **JPanel** qui contient le contexte graphique de l'animation. Un **JPanel** est une zone d'affichage simple qui doit être incluse dans un autre objet qui fournit le support à l'affichage sur l'écran de l'ordinateur. Ici, le **JPanel** est inclus dans la fenêtre *ExempleJFrameIncluantJPanelAvecAnimationParTimer* sous-classe de **JFrame**. Nous verrons plus loin que le **JPanel** peut aussi être inclus dans une application Web sans modification.

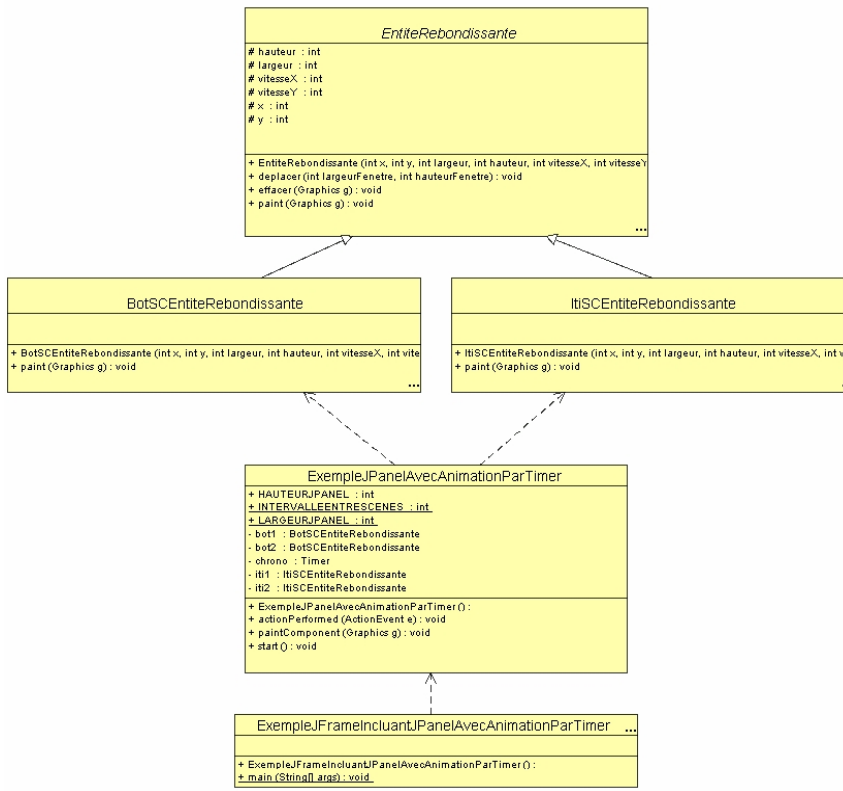


Figure 27. Nouvelle organisation des classes qui passe par un JPanel.

Le délai entre deux scènes de la séquence d'animation est contrôlé par un objet de la classe **Timer** qui génère un événement à toutes les 50 *ms*. Voici le code de la classe *ExempleJPanelAvecAnimationParTimer*, sous-classe de *javax.swing.JPanel*.

JavaPasAPas/chapitre_7/
ExempleJPanelAvecAnimationParTimer.java

```
// Plusieurs Bot et Iti qui bougent dans un JPanel avec Timer
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ExempleJPanelAvecAnimationParTimer extends JPanel implements ActionListener {

 public static final int INTERVALLEENTRESCENES = 50; // En ms

 // Le chrono génère un événement à chaque intervalle
 private Timer chrono;
 // Entités du monde à animer
 private BotSCEntiteRebondissante bot1;
 private BotSCEntiteRebondissante bot2;
 private ItiSCEntiteRebondissante iti1;
 private ItiSCEntiteRebondissante iti2;

 // Taille du JPanel
 public static final int LARGEURJPANEL = 400;
 public static final int HAUTEURJPANEL = 400;

 // Conctructeur initialise les entit�s � animer
 public ExempleJPanelAvecAnimationParTimer() {
  bot1 = new BotSCEntiteRebondissante(0, 100, 100, 150, 5, 0);
  bot2 = new BotSCEntiteRebondissante(100, 100, 75, 100, -10, 5);
  iti1 = new ItiSCEntiteRebondissante(200, 300, 80, 80, 6, 6);
  iti2 = new ItiSCEntiteRebondissante(200, 0, 50, 50, 0, 10);
 }

 public void start() {
  if (chrono == null) {
   chrono = new Timer(INTERVALLEENTRESCENES, this);
   chrono.start();
  }
 }

 // Le chrono appelle actionPerformed périodiquement (boucle d'animation)
 public void actionPerformed(ActionEvent e) {
  // Affiche la scène
  repaint();

  // Déplace les entités à animer pour la prochaine scène
  bot1.deplacer(LARGEURJPANEL, HAUTEURJPANEL);
  bot2.deplacer(LARGEURJPANEL, HAUTEURJPANEL);
  iti1.deplacer(LARGEURJPANEL, HAUTEURJPANEL);
  iti2.deplacer(LARGEURJPANEL, HAUTEURJPANEL);
 }

 // paintComponent() est appelée indirectement par repaint()
 // N.B. Swing utilise le double tampon : pas besoin d'effacer !
 public void paintComponent(Graphics g) {
  super.paintComponent(g);

  // Dessine les entités de l'animation
```

```
  bot1.paint(g);
  bot2.paint(g);
  iti1.paint(g);
  iti2.paint(g);
 }
}
```

Regardons maintenant le détail de la classe. Le constructeur initialise les entités à animer :

```
public ExempleJPanelAvecAnimationParTimer() {
  bot1 = new BotSCEntiteRebondissante(0,100,100,150,5,0);
  bot2 = new BotSCEntiteRebondissante(100,100,75,100,-10,5);
  iti1 = new ItiSCEntiteRebondissante(200,300,80,80,6,6);
  iti2 = new ItiSCEntiteRebondissante(200,0,50,50,0,10);
 }
```

La méthode *start*() doit être appelée pour démarrer l'animation. Dans notre exemple, elle sera appelée par la méthode *main*() de la classe *ExempleJFrameIncluantJPanelAvecAnimationParTimer* que nous verrons plus loin. La méthode *start*() crée le chronomètre (objet de la classe *Timer*) et le démarre.

```
public void start(){
    if (chrono == null){
        chrono = new Timer(intervalleEntreScenes,this);
        chrono.start();
    }
}
```

Si la variable *chrono* est *null*, un nouvel objet *chrono* de la classe *Timer* est créé. Le premier paramètre du constructeur est l'intervalle de temps et le second est l'objet qui doit répondre à l'événement généré par le *Timer*. Dans notre exemple, le second paramètre est *this*, ce qui signifie que la méthode de réponse à l'événement du *Timer*, la méthode *actionPerformed*(), doit se trouver dans la classe *ExempleJPanelAvecAnimationParTimer*.

Lorsqu'un événement est généré par l'objet *Timer*, la méthode *actionPerformed*() est appelée automatiquement sur l'objet de la classe *ExempleJPanelAvecAnimationParTimer* qui doit répondre à l'événement.

```
public void actionPerformed( ActionEvent e){
    // Affiche la scène
    repaint();

    // Déplace les entités à animer pour la prochaine scène
    bot1.deplacer(LARGEURJPANEL, HAUTEURJPANEL);
    bot2.deplacer(LARGEURJPANEL, HAUTEURJPANEL);
```

```
        iti1.deplacer(LARGEURJPANEL, HAUTEURJPANEL);
        iti2.deplacer(LARGEURJPANEL, HAUTEURJPANEL);
    }
```

Nous avons décrit au chapitre 5, le concept d'écouteur d'événement de souris qui doit implémenter l'interface *java.awt.event.MouseListener*. Par analogie pour répondre à l'événement du **Timer**, la classe *ExempleJPanelAvecAnimationParTimer* doit implémenter l'interface *java.awt.event.ActionListener* :

```
public class ExempleJPanelAvecAnimationParTimer extends JPanel implements ActionListener{
```

La méthode *actionPerformed*() appelle d'abord la méthode *repaint*() qui provoque l'appel de la méthode *paintComponent*() pour afficher la prochaine scène. La méthode *paintComponent*() de **JPanel** joue un rôle analogue à la méthode *paint*() de **JFrame** que nous avons employé jusqu'à présent. Ensuite, la méthode *actionPerformed*() mets les entités à jour pour la prochaine scène. En effectuant l'appel à *repaint*() avant de produire la prochaine scène, l'intervalle entre l'affichage de deux scènes est indépendant du temps de calcul nécessaire à la production de la prochaine scène.

Comme vu précédemment pour *paint*(), la méthode *paintComponent*() appelle d'abord la méthode correspondante de la super-classe. Ensuite, la scène est dessinée. Il est à noter qu'il n'est pas nécessaire de dessiner dans un tampon car l'approche du double tampon est automatiquement employée lorsque l'on passe par *repaint*() pour dessiner la prochaine scène ! D'autre part, il n'est pas nécessaire d'effacer les entités de la scène précédente car le fond de l'écran est rétabli par *repaint*() avant l'appel à *paintComponent*().

```
    public void paintComponent (Graphics g) {
        super.paintComponent(g);

        // Dessine les entités de l'animation
        bot1.paint(g); bot2.paint(g); iti1.paint(g); iti2.paint(g);
    }
```

La classe *ExempleJFrameIncluantJPanelAvecAnimationParTimer* est une sous-classe de **JFrame** qui représente la fenêtre contenant l'objet *ExempleJPanelAvecAnimationParTimer*.

```
import javax.swing.JFrame;
```

```
public class ExempleJFrameIncluantJPanelAvecAnimationParTimer extends
JFrame{

    public ExempleJFrameIncluantJPanelAvecAnimationParTimer() {
        super("Animation dans JPanel avec Timer");
        ExempleJPanelAvecAnimationParTimer leJPanelAnimation =
            new ExempleJPanelAvecAnimationParTimer();
        this.getContentPane().add(leJPanelAnimation);
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);
        this.setSize(
          ExempleJPanelAvecAnimationParTimer.LARGEURJPANEL,
          ExempleJPanelAvecAnimationParTimer.HAUTEURJPANEL+30);
        this.setVisible(true);
        leJPanelAnimation.start();
    }

    public static void main (String args[]) {
        new ExempleJFrameIncluantJPanelAvecAnimationParTimer();
    }
}
```

La ligne suivante crée l'objet d'animation *leJPanelAnimation*:

```
        ExempleJPanelAvecAnimationParTimer leJPanelAnimation =
            new ExempleJPanelAvecAnimationParTimer();
```

La ligne suivante ajoute l'objet d'animation *leJPanelAnimation* comme une composante de la fenêtre **JPanel**. Par conséquent, l'objet *leJPanelAnimation* sera affiché dans la fenêtre.

```
        this.getContentPane().add(leJPanelAnimation);
```

L'appel à la méthode *leJPanelAnimation.start*() démarre l'animation :

```
        leJPanelAnimation.start();
```

Exercice. Reprenez l'application précédente en incluant votre entité préférée.

7.2 Isoler le monde à animer du mécanisme d'animation

Cette section présente une refactorisation de l'exemple précédent (animation de plusieurs Bot et Iti) en isolant dans une classe *MondeAnime* les aspects du programme qui concerne le monde que nous voulons animer, indépendamment du mécanisme d'animation lui-même. Cette classe peut ainsi être utilisée soit dans une animation par boucle ou encore dans une animation avec *Timer*. Ceci illustre un des aspects les plus puissants de la

programmation objet. En regroupant de manière judicieuse les variables et méthodes sous forme de classe, on obtient une composante logicielle facile à comprendre et à réutiliser à travers différents contextes. Java inclut d'ailleurs un grand nombre de classes réutilisables. En développant ses propres classes, il est important de chercher à favoriser la réutilisabilité par un découpage judicieux des éléments du programme en classes.

JavaPasAPas/chapitre_7/MondeAnime.java

```
// Monde à animer
import java.awt.*;

public class MondeAnime {
  // Taille du monde
  public static int LARGEURMONDE = 400;
  public static int HAUTEURMONDE = 400;
  // Entités du monde à animer
  private BotSCEntiteRebondissante bot1 = new BotSCEntiteRebondissante(0, 100, 100, 150, 5, 0);
  private BotSCEntiteRebondissante bot2 = new BotSCEntiteRebondissante(100, 100, 75, 100, -10, 5);
  private ItiSCEntiteRebondissante iti1 = new ItiSCEntiteRebondissante(200, 300, 80, 80, 6, 6);
  private ItiSCEntiteRebondissante iti2 = new ItiSCEntiteRebondissante(200, 0, 50, 50, 0, 10);

  public MondeAnime() {
    // Initialise les entités à animer pour la première scène
    bot1 = new BotSCEntiteRebondissante(0, 100, 100, 150, 5, 0);
    bot2 = new BotSCEntiteRebondissante(100, 100, 75, 100, -10, 5);
    iti1 = new ItiSCEntiteRebondissante(200, 300, 80, 80, 6, 6);
    iti2 = new ItiSCEntiteRebondissante(200, 0, 50, 50, 0, 10);
  }

  public void prochaineScene() {
    // Modifie les entités à animer pour la prochaine scène
    bot1.deplacer(LARGEURMONDE, HAUTEURMONDE);
    bot2.deplacer(LARGEURMONDE, HAUTEURMONDE);
    iti1.deplacer(LARGEURMONDE, HAUTEURMONDE);
    iti2.deplacer(LARGEURMONDE, HAUTEURMONDE);
  }

  public void paint(Graphics g) {
    // Dessine la scène
    bot1.paint(g);
    bot2.paint(g);
    iti1.paint(g);
    iti2.paint(g);
  }
}
```

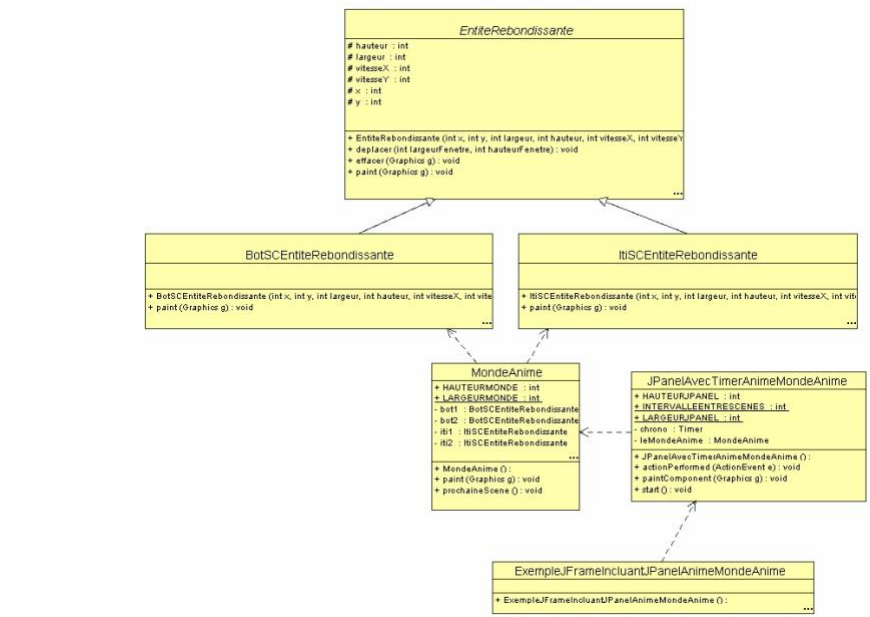


Figure 28. Isolation du monde à animer dans la classe *MondeAnime*.

La classe *JPanelAvecTimerAnimeMondeAnime* contient le mécanisme d'animation par *Timer*. Elle fait appel aux méthodes de la classe *MondeAnime* qui contient les détails du monde à animer. L'interaction entre la classe du monde à animer et le mécanisme d'animation est limité à l'appel de trois méthodes : le constructeur *MondeAnime*(), *prochaineScene*() et *paint*(). Les variables de classe constantes *MondeAnime*.LARGEURMONDE et *MondeAnime*.HAUTEURMONDE définissent la taille du monde 2D dans lequel les entités sont animées.

JavaPasAPas/chapitre_7/ JPanelAvecTimerAnimeMondeAnime.java

```
// JPanel qui anime un objet de MondeAnime
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JPanelAvecTimerAnimeMondeAnime extends JPanel implements ActionListener {

 public static final int INTERVALLEENTRESCENES = 50; // En ms

 // Le chrono génère un événement à chaque intervalle
 private Timer chrono;
 // Le monde à animer
 private MondeAnime leMondeAnime;
```

```
 // Taille du JPanel
 public static final int LARGEURJPANEL = MondeAnime.LARGEURMONDE;
 public static final int HAUTEURJPANEL = MondeAnime.HAUTEURMONDE;

 // Conctructeur initialise le monde à animer
 public JPanelAvecTimerAnimeMondeAnime() {
  leMondeAnime = new MondeAnime();
 }

 public void start() {
  if (chrono == null) {
   chrono = new Timer(INTERVALLEENTRESCENES, this);
   chrono.start();
  }
 }
 // Le chrono appelle actionPerformed périodiquement (boucle d'animation)
 public void actionPerformed(ActionEvent e) {
  repaint();
  // Produire la prochaine scène du monde à animer
  leMondeAnime.prochaineScene();
 }

 // paintComponent() est appelée indirectement par repaint()
 // N.B. Swing utilise le double tampon : pas besoin d'effacer !
 public void paintComponent(Graphics g) {
  super.paintComponent(g);

  // Dessine les entités de l'animation
  leMondeAnime.paint(g);
 }
}
```

Enfin, l'objet de la classe *JPanelAvecTimerAnimeMondeAnime* est inclus dans une fenêtre de la classe *ExempleJFrameIncluantJPanelAnimeMondeAnime*.

JavaPasAPas/chapitre_7/
ExempleJFrameIncluantJPanelAnimeMondeAnime.java

```
// JFrame qui inclue le JPanelAnimeMondeAnime
import javax.swing.JFrame;

public class ExempleJFrameIncluantJPanelAnimeMondeAnime extends JFrame {

  public ExempleJFrameIncluantJPanelAnimeMondeAnime() {
    super("Animation dans JPanel avec Timer");
    JPanelAvecTimerAnimeMondeAnime leJPanelAnimation = new
        JPanelAvecTimerAnimeMondeAnime();
    this.getContentPane().add(leJPanelAnimation);
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    this.setSize(
        JPanelAvecTimerAnimeMondeAnime.LARGEURJPANEL,
        JPanelAvecTimerAnimeMondeAnime.HAUTEURJPANEL + 30);
    this.setVisible(true);
    leJPanelAnimation.start();
  }

  public static void main(String args[]) {
    new ExempleJFrameIncluantJPanelAnimeMondeAnime();
  }
}
```

Il est important d'examiner attentivement la différence entre cette version du programme et la précédente (section 7.1). Le regroupement des aspects spécifiques au monde à animer, de manière indépendante du mécanisme d'animation, facilite la compréhension du programme. Les différents aspects du monde à animer ont été identifiés et isolés de manière précise. Ce partage des responsabilités permet de changer le mécanisme d'animation en réutilisant la même classe *MondeAnime*.

- **Animation par boucle explicite**

Dans l'exemple suivant, un **JPanel** utilise une boucle explicite pour animer le monde de la classe *MondeAnime* plutôt qu'un *Timer*. La classe *MondeAnime* est réutilisée telle quelle sans changement !

Exemple. Voici le code de la sous-classe de **JPanel** qui fournit le mécanisme d'animation sous forme d'une boucle dans la méthode *start*().

JavaPasAPas/chapitre_7/
JPanelAvecBoucleAnimeMondeAnime.java

```
// JPanel avec boucle qui anime un objet de MondeAnime
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JPanelAvecBoucleAnimeMondeAnime extends JPanel {

 public static final int INTERVALLEENTRESCENES = 50; // En ms

 // Le chrono génère un évènement à chaque intervalle
 //   private Timer chrono;
 // Le monde à animer
 private MondeAnime leMondeAnime;

 // Taille du JPanel
 public static final int LARGEURJPANEL = MondeAnime.LARGEURMONDE;
 public static final int HAUTEURJPANEL = MondeAnime.HAUTEURMONDE;

 // Conctructeur initialise le monde à animer
 public JPanelAvecBoucleAnimeMondeAnime() {
  leMondeAnime = new MondeAnime();
 }

 public void start() {
  while (true) {
   repaint();
   try {
    Thread.sleep(INTERVALLEENTRESCENES);
   } catch (InterruptedException exception) {
    System.err.println(exception.toString());
   }
   leMondeAnime.prochaineScene();
  }
 }

 // paintComponent() est appelée indirectement par repaint()
 // N.B. Swing utilise le double tampon : pas besoin d'effacer !
 public void paintComponent(Graphics g) {
  super.paintComponent(g);

  // Dessine les entités de l'animation
  leMondeAnime.paint(g);
 }
}
```

Comme pour la version avec *Timer*, l'animation est démarrée avec la méthode *start*() du **JPanel**. Contrairement aux exemples de boucle d'animation vus précédemment où la boucle était placée dans la méthode *paint*(), la fermeture de fenêtre fonctionne correctement car la boucle infinie dans *start*() peut être interrompue par un événement de souris. Dans la boucle, *repaint*() est appelée pour afficher la scène courante, ensuite *leMondeAnime.prochaineScene*() produit les changements du monde pour la prochaine scène.

```
    public void start(){
        while (true){
            repaint();
            try {
                Thread.sleep(intervalleEntreScenes);
            }
            catch(InterruptedException exception){
                System.err.println(exception.toString());
            }
        leMondeAnime.prochaineScene();
        }
    }
```

La sous-classe *JFrameIncluantJPanelAvecBoucleAnimeMondeAnime* de **JFrame** inclut l'objet de la classe *JPanelAvecBoucleAnimeMondeAnime* pour afficher l'animation dans une fenêtre :

JavaPasAPas/chapitre_7/

JFrameIncluantJPanelAvecBoucleAnimeMondeAnime.java

```
import javax.swing.JFrame;
public class JFrameIncluantJPanelAvecBoucleAnimeMondeAnime extends
JFrame{

    public JFrameIncluantJPanelAvecBoucleAnimeMondeAnime() {
        super("Animation dans JPanel avec boucle");
        JPanelAvecBoucleAnimeMondeAnime leJPanelAnimation =
            new JPanelAvecBoucleAnimeMondeAnime();
        this.getContentPane().add(leJPanelAnimation);
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);
        this.setSize(
          JPanelAvecBoucleAnimeMondeAnime.LARGEURJPANEL,
          JPanelAvecBoucleAnimeMondeAnime.HAUTEURJPANEL+30);
        this.setVisible(true);
        leJPanelAnimation.start();
    }

    public static void main (String args[]) {
        new JFrameIncluantJPanelAvecBoucleAnimeMondeAnime();
    }
}
```

7.3 Développement du jeu

Cette section développe un jeu simple 2D en raffinant les classes développées précédemment. Le jeu démarre en animant une série d'entités animées. Le but de l'utilisateur est de détruire ces entités en cliquant dessus avec la souris. Lorsque l'entité est touchée, elle disparaît en poussant un cri de désarroi. Nous allons d'abord raffiner la hiérarchie des entités du monde à animer en ajoutant des comportements plus sophistiqués (gestes, cris). La figure suivante montre les classes à développer.

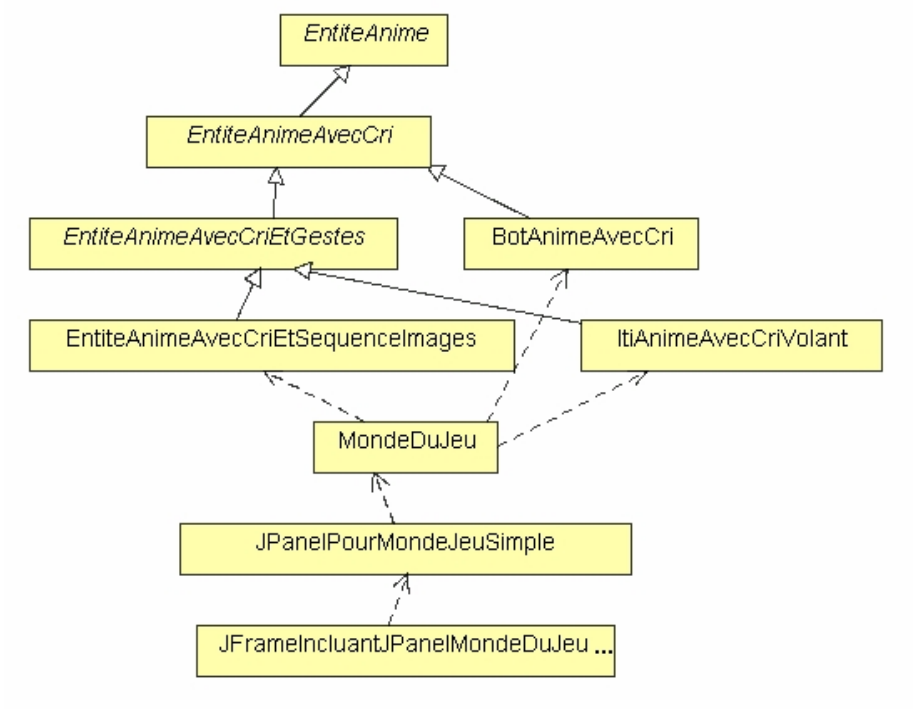


Figure 29. Classes du jeu simple.

La classe *EntiteAnime* du package *JeuSimple* ajoute les aspects suivants à la classe *EntiteAnime* développée au chapitre 7 :

- Visibilité : une entité peut être visible ou pas (variable d'objet *visible*). Si elle est invisible, un appel à *paintSiVisible*() ne la dessine pas.

- La méthode *touche*() : cette méthode retourne vrai si la coordonnée (*unX*,*unY*) est à l'intérieur du rectangle englobant de l'entité. Elle sert à vérifier si l'utilisateur a touché l'entité avec la souris dans le contexte du jeu.

- Méthodes de lecture (*lecteurs*/*getters*) et d'écriture (*modifieurs*/*setters*) de variables d'objet. Il est souvent nécessaire de pouvoir extraire ou modifier une variable membre d'une classe à partir de l'extérieur de cette classe. Il est possible de spécifier que la variable est *public* à cet effet. Cependant, cette approche est généralement à éviter pour

minimiser le couplage entre classes. Il est préférable d'utiliser des méthodes pour extraire (lecteur) ou modifier (modifieur) le contenu des variables. Une convention souvent employée en Java est de nommer par *getNomVariable*() et *setNomVariable*() les méthodes qui vont extraire le contenu et modifier le contenu de la variable appelée *nomVariable*. Des méthodes de lecture et d'écriture ont été ajoutées pour la plupart des variables d'objet de la classe *EntiteAnime* car ceci est nécessaire dans notre programme.

JavaPasAPas/JeuSimple/EntiteAnime.java

```
package JeuSimple;

import java.awt.*;

public abstract class EntiteAnime {
  // Variables d'objet qui décrivent l'état d'un objet EntiteAnime
  protected int x, y; // Coordonnées de l'entité
  protected int largeur, hauteur; // Taille du rectangle englobant
  protected int vitesseX; // Vitesse de déplacement dans l'axe x
  protected int vitesseY; // Vitesse de déplacement dans l'axe y
  protected boolean visible; // Indique si l'entité doit être affichée

  // Constructeur
  public EntiteAnime(
      int x, int y, int largeur, int hauteur, int vitesseX, int vitesseY, boolean visible) {
    this.x = x;
    this.y = y;
    this.hauteur = hauteur;
    this.largeur = largeur;
    this.vitesseX = vitesseX;
    this.vitesseY = vitesseY;
    this.visible = visible;
  }
  // Méthode de lecture et d'écriture des variables d'objet
  public void setX(int x) {
    this.x = x;
  }

  public void setY(int y) {
    this.y = y;
  }

  public void setLargeur(int largeur) {
    this.largeur = largeur;
  }

  public void setHauteur(int hauteur) {
    this.hauteur = hauteur;
  }

  public void setVisible(boolean visible) {
    this.visible = visible;
  }

  public void setVitesseX(int vitesseX) {
    this.vitesseX = vitesseX;
  }
  public void setVitesseY(int vitesseY) {
    this.vitesseY = vitesseY;
```

```
  }

  public int getX() {
    return x;
  }
  public int getY() {
    return y;
  }

  public int getLargeur() {
    return largeur;
  }

  public int getHauteur() {
    return hauteur;
  }

  public boolean getVisible() {
    return visible;
  }
  public int getVitesseX() {
    return vitesseX;
  }
  public int getVitesseY() {
    return vitesseY;
  }
  // Modifier les Coordonnées pour la prochaine scène
  public void prochaineScene(int largeurMonde, int hauteurMonde) {
    if (x + largeur >= largeurMonde | x < 0) // Si atteint le bord selon x
    vitesseX = -vitesseX; // Inverser la direction selon x
    x = x + vitesseX; // déplacement selon x
    if (y + hauteur >= hauteurMonde | y < 0) // Si atteint le bord selon y
    vitesseY = -vitesseY; // Inverser la direction selon y
    y = y + vitesseY; // déplacement selon y
  }

  // Détermine si la coordonnée unX,unY touche au rectangle englobant de l'entité
  public boolean touche(int unX, int unY) {
    return ((unX >= x) && (unX <= x + largeur) && (unY >= y) && (unY <= y + hauteur));
  }

  // Méthode abstraite de dessin de l'entité
  public abstract void paint(Graphics g);
  // Dessine seulement si visible
  public void paintSiVisible(Graphics g) {
    if (visible) {
      paint(g);
    }
  }

  // Effacer l'entité
  public void effacer(Graphics g) {
    g.clearRect(x, y, largeur, hauteur);
  }
}
```

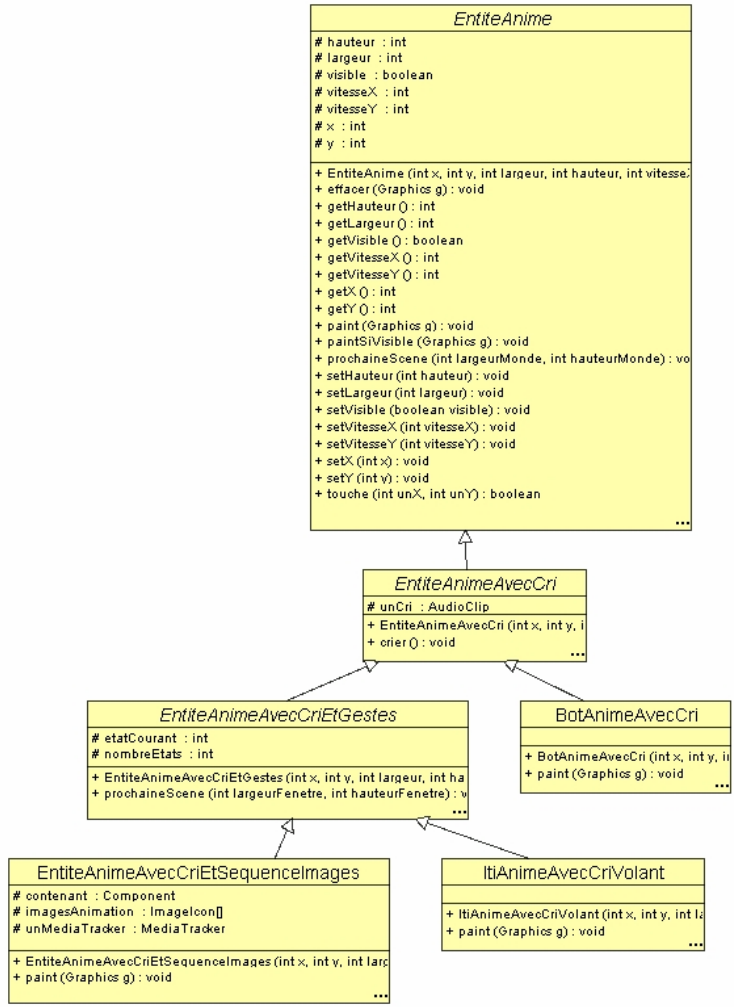


Figure 30. Hiérarchie des entités animées.

- **Traitement du son avec *AudioClip***

La classe *EntiteAnimeAvecCri*, sous-classe de *EntiteAnime*, vise à associer un son à l'entité qui servira dans le contexte du jeu. Lorsqu'une entité est touchée par la souris elle pousse un cri de désarroi.

JavaPasAPas/JeuSimple/EntiteAnimeAvecCri.java

```
package JeuSimple;

import java.applet.*;
import java.net.URL;

public abstract class EntiteAnimeAvecCri extends EntiteAnime {
 protected AudioClip unCri; // Cri de l'Entité

 public EntiteAnimeAvecCri(
    int x,
    int y,
    int largeur,
    int hauteur,
    int vitesseX,
    int vitesseY,
    boolean visible,
    String nomFichierAudio) {
  super(x, y, largeur, hauteur, vitesseX, vitesseY, visible);
  // Charge le son dans le fichier nomFichierAudio
  // Le fichier est dans le dossier de EntiteAnimeAvecCri.class
  // Cherche l'URL du fichier
  URL url = EntiteAnimeAvecCri.class.getResource(nomFichierAudio);
  // Charge le clip audio à partir de l'URL
  unCri = Applet.newAudioClip(url);
 }

 public void crier() {
  unCri.play();
 }
}
```

La variable d'objet *unCri* de la classe java.applet.**AudioClip** contient un clip audio :

```
protected AudioClip unCri; //Cri de l'entité
```

Le clip est lu d'un fichier dont le nom, *nomFichierAudio*, est passé en paramètre au constructeur. L'appel suivant construit une adresse sous forme d'URL qui fait référence au fichier qui contient le clip audio. Sans entrer dans les détails de la notion d'URL, mentionnons que dans notre contexte, l'URL représente le chemin du fichier. On suppose que le fichier se trouve dans le même dossier que le fichier *EntiteAnimeAvecCri.class*.

```
URL = EntiteAnimeAvecCri.class.getResource(nomFichierAudio);
```

L'appel suivant construit l'objet *AudioClip* à partir du fichier audio.

```
unCri = Applet.newAudioClip(url);
```

La méthode *crier*() appelle tout simplement la méthode *play*() de l'objet *unCri* pour jouer le clip audio.

```
public void crier (){unCri.play();}
```

La méthode *paint()* de la classe *BotAnimeAvecCri* sous-classe de *EntiteAnimeAvecCri* précise la manière de dessiner un Bot.

JavaPasAPas/JeuSimple/BotAnimeAvecCri.java

```
package JeuSimple;

import java.awt.*;

public class BotAnimeAvecCri extends EntiteAnimeAvecCri {

 public BotAnimeAvecCri(
    int x,
    int y,
    int largeur,
    int hauteur,
    int vitesseX,
    int vitesseY,
    boolean visible,
    String fichierAudio) {
  super(x, y, largeur, hauteur, vitesseX, vitesseY, visible, fichierAudio);
 }

 public void paint(Graphics g) {
  g.setColor(Color.green);
  g.fillOval(x, y, largeur, hauteur / 2); // La tête

  g.setColor(Color.black);
  g.fillRect(x + largeur / 4, y + hauteur / 8, largeur / 10, hauteur / 20); // L'oeil gauche
  g.fillRect(
     x + largeur * 3 / 4 - largeur / 10,
     y + hauteur / 8,
     largeur / 10,
     hauteur / 20); // L'oeil droit
  g.drawLine(
     x + largeur / 4,
     y + hauteur * 3 / 8,
     x + largeur * 3 / 4,
     y + hauteur * 3 / 8); // La bouche

  g.setColor(Color.red);
  g.fillRect(x, y + hauteur / 2, largeur, hauteur / 2); // Le corps
 }
}
```

- **Animation vectorielle de gestes (par opérations de dessin)**

Pour rendre l'animation plus intéressante, une entité peut non seulement se déplacer mais aussi effectuer des gestes. Pour simplifier, supposons que l'entité effectue toujours la même série de gestes en répétant la même séquence d'images. Généralement, on distingue deux techniques de production des images simulant les gestes :

- *Animation vectorielle*. Chaque image est produite en invoquant des opérations de dessins, par exemple, avec Java 2D.

- *Animation bitmap*. Chaque image est préalablement créée. Cette approche évite le calcul répété des images à dessiner mais nécessite le stockage des images. Dans le cas d'images très complexes, il peut être très difficile de produire le résultat recherché avec des opérations de dessin.

Pour produire la séquence d'animation des gestes, il faut un moyen de déterminer la forme graphique appropriée pour la prochaine scène. La classe *EntiteAnimeAvecCriEtGestes* sous-classe de *EntiteAnimeAvecCri* ajoute deux variables d'objets à cet effet. La variable d'objet *nombreEtats* représente le nombre de formes graphiques de l'entité et *etatCourant* détermine la prochaine forme graphique de l'entité. La méthode *prochaineScene*() fait passer à la forme suivante en ajoutant 1 à *etatCourant* (modulo le *nombreEtats* afin de produire un effet de répétition du mouvement). L'opération modulo (reste après division entière) est représentée par le symbole %.

JavaPasAPas/JeuSimple/EntiteAnimeAvecCriEtGestes.java

```
package JeuSimple;

public abstract class EntiteAnimeAvecCriEtGestes extends EntiteAnimeAvecCri {
 protected int etatCourant = 0;
 protected int nombreEtats = 1;

 public EntiteAnimeAvecCriEtGestes(
    int x,
    int y,
    int largeur,
    int hauteur,
    int vitesseX,
    int vitesseY,
    boolean visible,
    String nomFichierAudio,
    int nombreEtats) {
  super(x, y, largeur, hauteur, vitesseX, vitesseY, visible, nomFichierAudio);
  this.nombreEtats = nombreEtats;
 }

 public void prochaineScene(int largeurFenetre, int hauteurFenetre) {
  super.prochaineScene(largeurFenetre, hauteurFenetre);
  etatCourant = (etatCourant + 1) % nombreEtats;
 }
}
```

La méthode *paint*() de la sous-classe *ItiAnimeAvecCriVolant* tient compte de la variable *etatCourant* dans le dessin du *Iti* en faisant bouger ses bras pour donner l'impression d'un battement d'ailes. La position des bras dépend de la valeur de *etatCourant*.

JavaPasAPas/JeuSimple/ItiAnimeAvecCriVolant.java

```
package JeuSimple;

import java.awt.*;

public class ItiAnimeAvecCriVolant extends EntiteAnimeAvecCriEtGestes {
 public ItiAnimeAvecCriVolant(
    int x,
    int y,
    int largeur,
    int hauteur,
    int vitesseX,
    int vitesseY,
    boolean visible,
    String fichierAudio) {
  super(x, y, largeur, hauteur, vitesseX, vitesseY, visible, fichierAudio, 3);
 }

 public void paint(Graphics g) {
  int milieux = x + largeur / 2;
  int milieuy = y + hauteur / 2;

  // La tête
  g.setColor(Color.pink);
  g.fillOval(x + largeur / 3, y, largeur / 3, hauteur / 4);
  // Le sourire
  g.setColor(Color.black);
  g.drawArc(x + largeur / 3, y - hauteur / 12, largeur / 3, hauteur / 4, -125, 70);
  // Les yeux
  g.fillOval(milieux - largeur / 8, y + hauteur / 12, largeur / 12, hauteur / 24);
  g.fillOval(milieux + largeur / 8 - largeur / 12, y + hauteur / 12, largeur / 12, hauteur / 24);

  // Le corps
  g.drawLine(milieux, y + hauteur / 4, milieux, y + hauteur * 3 / 4);
  // Les bras
  g.drawLine(x, y + hauteur / 4 + (hauteur / 4) * etatCourant, milieux, milieuy);
  g.drawLine(x + largeur, y + hauteur / 4 + (hauteur / 4) * etatCourant, milieux, milieuy);
  // Les jambes
  g.drawLine(x, y + hauteur, milieux, y + hauteur * 3 / 4);
  g.drawLine(x + largeur, y + hauteur, milieux, y + hauteur * 3 / 4);
 }
}
```

- **Animation bitmap de gestes**

La sous-classe *EntiteAnimeAvecCriEtSequenceImages* illustre une autre manière de produire l'animation :

JavaPasAPas/JeuSimple/
EntiteAnimeAvecCriEtSequenceImages.java

```
package JeuSimple;

import java.awt.*;
import java.net.URL;
import javax.swing.*;

public class EntiteAnimeAvecCriEtSequenceImages extends EntiteAnimeAvecCriEtGestes {

 protected Image imagesAnimation[];
 protected Component contenant;

 public EntiteAnimeAvecCriEtSequenceImages(
    int x,
    int y,
    int largeur,
    int hauteur,
    int vitesseX,
    int vitesseY,
    boolean visible,
    String fichierAudio,
    int nombreEtats,
    String nomDossier) {
  super(x, y, largeur, hauteur, vitesseX, vitesseY, visible, fichierAudio, nombreEtats);

  // Charge les images de l'animation
  // On suppose que les fichiers .gif se trouvent dans un dossier nomm□ nomDossier
  // dans le répertoire du code compilé et que les noms de fichiers gif sont
  // de la forme nomDossiern.gif, n = 1 .. nombreEtats
  this.imagesAnimation = new Image[nombreEtats];
  for (int i = 0; i < nombreEtats; i++) {
   URL url = getClass().getResource(nomDossier + "/" + nomDossier + (i + 1) + ".gif");
   this.imagesAnimation[i] = new ImageIcon(url).getImage();
  }
 }

 public void paint(Graphics g) {
  g.drawImage(imagesAnimation[etatCourant], x, y, largeur, hauteur, null);
 }
}
```

Plutôt que d'employer les méthodes de dessin de *Graphics*, une séquence d'images pré-enregistrées est utilisée. Java inclut des classes pour la manipulation des images selon différentes normes telles GIF (*Graphics Interchange Format*), JPEG (*Joint Photographic Experts Group*) et PNG (*Portable Network Graphics*). Notre classe suppose que les images sont des fichiers de type GIF.

Exemple. La figure suivante montre le contenu de 9 fichiers GIF qui représentent une séquence d'images utilisées pour produire une animation d'un *coq* qui bat des ailes pour voler approximativement. Le nom du fichier

est montré sous l'image. L'extension *.gif* est une convention pour les fichiers d'images encodés selon la norme GIF.

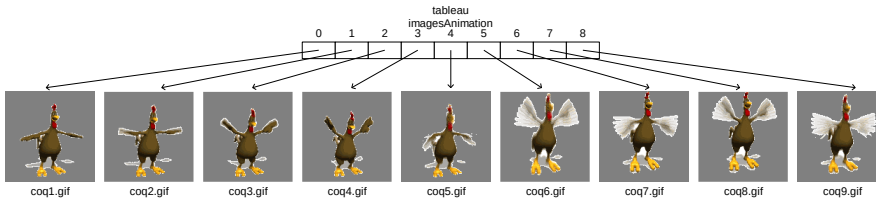


Figure 31. Série d'images pour l'animation du coq qui bât des ailes pour voler approximativement.

- **Notion de tableau**

Dans notre programme, une image est représentée par un objet de la classe *Image*. Il faut un objet pour chacune des images (9 pour le coq). Lorsqu'il est nécessaire de manipuler une série d'objets du même type, il est possible de déclarer une variable pour chacun des objets mais ceci devient rapidement lourd lorsque le nombre croît. Une solution consiste à employer une structure de donnée telle qu'un *tableau*.

La ligne suivante déclare un tableau nommé *imagesAnimation* illustré à la figure précédente. Les crochets [ ] qui suivent le nom de la variable indique que c'est un tableau.

```
protected Image imagesAnimation[];
```

Le nom de classe *Image* qui précède le nom du tableau signifie que chacune des cases du tableau représente en quelque sorte une variable de type *Image* et donc fait référence à un objet de la classe *Image*.

En Java, un tableau est considéré comme un objet. Comme pour un objet, le *new* crée l'objet tableau. Le nombre d'éléments du tableau est spécifié entre crochets (ici *nombreEtats*):

```
this.imagesAnimation = new Image[nombreEtats];
```

La boucle suivante crée les objets *Image* à partir des fichiers GIF et les affectent aux cases du tableau.

```
for (int i = 0; i < nombreEtats; i++){
```

L'URL est créé comme précédemment pour le fichier audio avec *getResource*() :

```
            URL =
getClass().getResource(nomDossier+"/"+nomDossier+(i+1)+".gif");
```

Le paramètre

```
nomDossier+"/"+nomDossier+(i+1)+".gif"
```

est le chemin du fichier relativement au dossier de la classe *EntiteAnimeAvecCriEtSequenceImages.class*. Pour notre exemple de coq, les fichiers sont nommés coq*n*.gif où *n* est un entier de 1 à 9. L'entier désigne l'ordre de l'image dans la séquence d'animation. De plus, les images sont dans un dossier nommé *coq* dans le répertoire du fichier compilé *EntiteAnimeAvecCriEtSequenceImages.class*.

La ligne suivante crée l'objet de la classe *Image* à partir de l'URL du fichier et l'affecte à la case *i* du tableau *imagesAnimation* :

```
this.imagesAnimation[i] = new ImageIcon(url).getImage();
```

La notation avec crochets *imagesAnimation*[*i*] désigne la case d'indice *i* du tableau. Il est à noter que, en Java , les cases d'un tableau sont numérotées de 0 à *n*-1 où *n* est le nombre de cases. Le numéro de case est appelé un *indice*. Par exemple, pour le coq, le tableau a 9 cases dont les indices vont de 0 à 8 comme illustré à la figure précédente. Dans la figure, il faut comprendre qu'une case fait référence à un objet de la classe *Image* qui contient l'image produite à partir du fichier. Par exemple, *imagesAnimation*[3] fait référence à un objet de la classe *Image* qui contient l'image du fichier *coq4.gif*.

Dans la classe *EntiteAnimeAvecCriEtSequenceImages*, la méthode *paint*() affiche successivement les images du tableau en se servant de *etatCourant* comme indice :

```
g.drawImage(imagesAnimation[etatCourant],x,y,largeur,hauteur,null);
```

Modes de transparence

Lorsqu'une image est affichée, chacun des pixels peut être affiché selon trois modes de transparence :

Opaque. Le pixel apparaît sans altération en remplaçant la couleur existante.

Transparent. Le pixel n'est pas affiché. On voit donc ce qui est déjà affiché. Dans notre exemple de coq, le gris foncé qui entoure le coq correspond à des pixels transparents.

Translucide. Le pixel est affiché en laissant transparaître partiellement ce qui est déjà affiché. Ceci est réalisé en combinant la couleur existante avec la couleur du pixel à afficher.

Le mode de transparence du pixel est spécifié dans l'encodage de l'image. Le format GIF permet à chacun des pixels d'être transparent ou opaque. Le format PNG permet les trois modes alors que le format JPEG ne permet que le mode opaque. Les éditeurs d'images permettent de préciser le mode de transparence de chacun des pixels.

Caractéristiques des tableaux Java

Un tableau Java est considéré comme un objet et chacun des éléments du tableau doit être du même type. Le type des éléments primitif ou objet. Le nombre d'éléments d'un tableau ne peut changer après sa création. Les Collections Java, introduites plus loin, permettent de traiter des groupes de taille variable au besoin. A la création, les éléments d'un tableau sont initialisés avec une valeur de défaut. Il est aussi possible de préciser des valeurs initiales à la création du tableau en affectant au tableau une suite d'éléments entre accolades.

Exemple. L'exemple suivant illustre l'utilisation d'un tableau.

```
public class ExempleTableau {
    public static void main(String[] args) {
        int tableauDe5Int[] = {12, 3, 154, -5, 17};
        for (int i = 0; i < tableauDe5Int.length; i++)
            System.out.println(i+" "+ tableauDe5Int[i]);
        for (int unInt : tableauDe5Int)
             System.out.println(unInt);
    }
}
```

Le tableau *tableauDe5Int* est initialisé avec les cinq entiers entre accolades :

```
int tableauDe5Int[] = {12, 3, 154, -5, 17};
```

Le premier *for* permet d'itérer sur chacun des éléments du tableau à l'aide du compteur *i* qui joue le rôle d'indice qui prend les valeurs de 0 à la taille du tableau contenue dans *tableauDe5Int.length*. Le corps du *for* imprime l'indice *i* avec l'élément correspondant du tableau désigné par *tableauDe5Int*[*i*].

```
for (int i = 0; i < tableauDe5Int.length; i++)
    System.out.println(i+" "+ tableauDe5Int[i]);
```

Le deuxième *for* montre une forme spéciale du *for* adaptée à l'itération des éléments du tableau sans avoir à spécifier un compteur explicite.

```
for (int unInt : tableauDe5Int)
     System.out.println(unInt);
```

Le *for* permet de spécifier une variable, ici *unInt*, qui représente un élément du tableau à l'intérieur du *for*. Le type de la variable est déclaré en entête devant le nom de la variable. La variable est suivie de « : » et du nom du tableau à parcourir. Le *for* parcoure ainsi les éléments du tableau du premier au dernier et affecte l'élément courant de la boucle à la variable spécifiée, soit *unInt*, dans l'exemple.

Exercice. Ajoutez à l'exemple précédent le calcul de la moyenne des éléments du tableau et la différence de chacun des éléments par rapport à la moyenne.

Tableaux multi-dimensionnels

En Java il est possible que chacun des éléments d'un tableau soit lui-même un tableau (par nécessairement de même dimension).

Exemple. L'exemple suivant illustre l'utilisation d'un tableau à deux dimensions (matrice).

```
public class ExempleTableau {
   public static void main(String[] args) {
      int tableauDe5Int[] = {12, 3, 154, -5, 17};
      for (int i = 0; i < tableauDe5Int.length; i++)
         System.out.println(i+" "+ tableauDe5Int[i]);
      for (int unInt : tableauDe5Int)
         System.out.println(unInt);
   }
}
```

La déclaration du type multi-dimensionnel est faite par une séquence de [] pour désigner chacune des dimensions. Le code suivant déclare un tableau *matrice2par3* à deux dimensions, 2 lignes par 3 colonnes.

```
int matrice2par3 [][] = {{11, 3, 2}, {-5, 7, 2 }};
```

Les éléments sont énumérés ligne par ligne. La notation *matrice2par3*[2][3] représente l'élément de la deuxième ligne et de la troisième colonne.

Deux boucles imbriquées avec compteur parcourent les éléments ligne par ligne, une boucle pour chacune des deux dimensions. Le compteur *i* correspond à la ligne et *j* à la colonne.

```
for (int i = 0; i < 2 ; i++){
    for (int j = 0; j < 3 ; j++)
        System.out.print(matrice2par3 [i][j] + " ");
    System.out.println();
}
```

La forme étendue du *for* peut aussi être employée :

```
for (int[] ligne : matrice2par3) {
    for (int unInt : ligne)
        System.out.print(unInt + " ");
    System.out.println();
}
```

Exercice. Afficher le produit de deux matrices.

- **Classe *MondeDuJeu***

La classe *MondeDuJeu* reproduit la même organisation que la classe *MondeAnime* développée précédemment. En plus des méthodes *prochaineScene*() et *paint*(), elle ajoute une nouvelle méthode *mousePressed*() qui doit être appelée pour répondre au click de la souris. La classe utilise des entités animées créées à partir des classes précédentes.

JavaPasAPas/JeuSimple/MondeDuJeu.java

```
package JeuSimple;
/*
 * MondeDuJeu.java
 * Plusieurs bonhommes dans un Vector
 * Le Iti vole (changement du dessin à chaque état)
 * Ajoute un coq (animation par séquence d'images)
 */
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.util.*;
import javax.swing.*;

public class MondeDuJeu {

 // Taille du monde
 public static final int LARGEURMONDE = 1000;
 public static final int HAUTEURMONDE = 1000;

 protected Vector vecteurEntites;

 public MondeDuJeu() {
  vecteurEntites = new Vector();
  vecteurEntites.addElement(new BotAnimeAvecCri(10, 100, 100, 200, 3, 3, true, "Son2.wav"));
  vecteurEntites.addElement(new ItiAnimeAvecCriVolant(200, 50, 100, 200, 3, 0, true, "Son3.wav"));
  vecteurEntites.addElement(new KennyAnimeAvecCri(100, 50, 100, 200, 3, 9, true, "Son1.wav"));
  vecteurEntites.addElement(
     new EntiteAnimeAvecCriEtSequenceImages(
        50, 100, 300, 300, 5, 5, true, "Son4.wav", 9, "coq"));
  vecteurEntites.addElement(
     new EntiteAnimeAvecCriEtSequenceImages(
        175, 800, 200, 200, 0, 0, true, "invince.wav", 6, "homer"));
 }

 public void prochaineScene() {
  for (Iterator unIterator = vecteurEntites.iterator(); unIterator.hasNext(); ) {
   ((EntiteAnime) unIterator.next()).prochaineScene(LARGEURMONDE, HAUTEURMONDE);
  }
 }

 public void paint(Graphics g) {
  for (Iterator unIterator = vecteurEntites.iterator(); unIterator.hasNext(); ) {
   ((EntiteAnime) unIterator.next()).paintSiVisible(g);
  }
 }

 // Si une entité est cliquée, elle disparait en poussant un cri
 public void mousePressed(MouseEvent e) {
  for (Iterator unIterator = vecteurEntites.iterator(); unIterator.hasNext(); ) {
   EntiteAnimeAvecCri uneEntiteAnimee = (EntiteAnimeAvecCri) unIterator.next();
   if (uneEntiteAnimee.touche(e.getX(), e.getY())) {
    uneEntiteAnimee.setVisible(false);
```

```
        uneEntiteAnimee.crier();
      }
    }
  }
}
```

- **Collections en Java**

L'ensemble des entités du jeu est représenté par un objet *vecteurEntités* de la classe java.util.**Vector** :

```
vecteurEntités = new Vector();
```

Un tel objet correspond au concept mathématique de vecteur et il est semblable à un tableau au sens où il permet de remplacer plusieurs variables par une seule. Comme pour un tableau, il est possible d'accéder à chacun des objets dans un **Vector**. Cependant, un **Vector** se distingue d'un tableau par deux aspects. La manière d'accéder aux objets est différente et le nombre d'objets d'un **Vector** n'est pas fixe. À noter que pour notre exemple de jeu, on aurait aussi bien pu employer un tableau Java étant donné que le nombre d'entités du jeu est connu à l'avance. Un **Vector** est employé afin d'illustrer le mécanisme de base des collections en le contrastant avec le mécanisme de tableau vu précédemment.

La classe **Vector** fait partie d'un ensemble de classes Java qui implémentent l'interface java.util.**Collection**. Ces classes permettent toutes de manipuler des collections d'objets mais elles se distinguent surtout par différentes possibilités pour accéder aux objets.

La méthode **addElement**(**E** obj) sert à ajouter un objet à un **Vector**. Le constructeur de *MondeDuJeu* fait une série d'appels à *addElement*() afin d'ajouter les objets correspondant aux entités à animer dans le **Vector** *vecteurEntités*. Par exemple, l'appel suivant ajoute un nouvel objet de la classe *BotAnimeAvecCri* :

```
vecteurEntités.addElement(new
BotAnimeAvecCri(10,100,20,40,3,3,true,"Son2.wav"));
```

L'objet est toujours ajouté à la suite des autres objets déjà contenus dans le **Vector**. Les objets sont ainsi ordonnés en fonction de l'ordre des appels à *addElement*(). Contrairement au tableau, on ne précise pas à quelle position

exacte l'objet est ajouté. Comme pour un tableau, l'utilisation d'un **Vector** simplifie énormément le codage lorsqu'il y a une collection d'entités à traiter.

On peut parcourir les objets du **Vector**, un par un, en passant par un objet **Iterator** produit à partir du **Vector**. Par exemple, dans la méthode *prochaineScene*() de *MondeDuJeu*, on appelle la méthode *prochaineScene*(*LARGEURMONDE*,*HAUTEURMONDE*) sur chacun des objets (entités) de *vecteurEntités* de la manière suivante :

```
        for(Iterator unIterator = vecteurEntités.iterator();
unIterator.hasNext();){

((EntiteAnime)unIterator.next()).prochaineScene(LARGEURMONDE,HAUTEURM
ONDE);
        }
```

Pour accéder aux objets, un objet **Iterator** est créé à partir du **Vector** dans la partie initialisation du *for*. L'objet **Iterator** maintient une position courante dans le **Vector**. La méthode **hasNext**() vérifie s'il reste encore un objet à parcourir. La méthode **next**() retourne le prochain objet en modifiant la position courante. Au départ, la position courante est avant le premier objet. Par opposition à un tableau, il n'est pas nécessaire de préciser la position de l'objet par un indice. La méthode **next**() retourne l'objet suivant en fonction de la position courante dans le **Vector** qui est maintenue dans l'objet **Iterator**.

A noter l'utilisation de la conversion de type:

```
(EntiteAnime)unIterator.next())
```

La méthode **next**() retourne un objet de la classe *java.lang.Object* super-classe de toutes les classes Java. Pour appeler la méthode *prochaineScene*() sur cet objet, il faut d'abord convertir l'objet en un *EntiteAnime*. Rappelons que c'est le principe de surcharge dynamique qui détermine la méthode *prochaineScene*() qui est effectivement exécutée. Dans le cas d'un objet de la classe *EntiteAnimeAvecCriEtSequenceImages* ou *ItiAnimeAvecCriVolant*, c'est la méthode définie dans la super-classe *EntiteAnimeAvecCriEtGestes* qui est exécutée. Dans le cas d'un objet *BotAnimeAvecCri*, c'est la méthode de la super-classe *EntiteAnime* qui est exécutée. Cet exemple illustre un aspect très puissant de la programmation objet. En regroupant un ensemble d'objets possiblement de classes différentes dans une collection, on peut par la suite

appeler une méthode sur tous les objets de la collection par une itération même si le traitement effectué varie en fonction de la classe de l'objet.

Le même principe est employé dans la méthode *paint*() qui appelle *paintSiVisible*() sur chacun des objets du *Vector*.

```
    public void paint(Graphics g){
        for(Iterator unIterator = vecteurEntités.iterator();
unIterator.hasNext();){
            ((EntiteAnime)unIterator.next()).paintSiVisible(g);
        }
    }
```

La méthode *mousePressed*() est invoquée lors d'un click de la souris. Elle vérifie, pour chacune des entités, si le curseur de la souris la touche. Si c'est le cas, l'entité devient invisible et la méthode *crier*() est appelée :

```
    public void mousePressed(MouseEvent e){
        for(Iterator unIterator = vecteurEntités.iterator();
unIterator.hasNext();){
            EntiteAnimeAvecCri uneEntitéAnimée =
(EntiteAnimeAvecCri)unIterator.next();
            if (uneEntitéAnimée.touche(e.getX(),e.getY())) {
                uneEntitéAnimée.setVisible(false);
                uneEntitéAnimée.crier();
            }
        }
    }
```

Voici le code de la classe *JPanelPourMondeJeuSimple*.

JavaPasAPas/JeuSimple/JPanelPourMondeJeuSimple.java

```
package JeuSimple;

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

public class JPanelPourMondeJeuSimple extends JPanel implements ActionListener, MouseListener {

 public static final int INTERVALLEENTRESCENES = 50; // En ms

 // Le chrono génère un évènement a chaque intervalle
 private Timer chrono;
 // Le monde a animer
 private MondeDuJeu leMondeDuJeu;

 // Taille du JPanel
 public static final int LARGEURJPANEL = MondeDuJeu.LARGEURMONDE;
 public static final int HAUTEURJPANEL = MondeDuJeu.HAUTEURMONDE;
```

```
// Conctructeur initialise le monde à animer
public JPanelPourMondeJeuSimple() {
  leMondeDuJeu = new MondeDuJeu();
  addMouseListener(this);
}

public void start() {
  if (chrono == null) {
    chrono = new Timer(INTERVALLEENTRESCENES, this);
    chrono.start();
  }
}

// Le chrono appelle actionPerformed périodiquement (boucle d'animation)
public void actionPerformed(ActionEvent e) {
  repaint();
  // Produire la prochaine scène du monde à animer
  leMondeDuJeu.prochaineScene();
}

// paintComponent() est appelée indirectement par repaint()
// N.B. Swing utilise le double tampon : pas besoin d'effacer !
public void paintComponent(Graphics g) {
  super.paintComponent(g);

  // Dessine les entités de l'animation
  leMondeDuJeu.paint(g);
}

public void mousePressed(MouseEvent leMouseEvent) {
  leMondeDuJeu.mousePressed(leMouseEvent);
}
// Il faut absolument définir les autres méthodes pour les autres
// événements de souris même s'il ne font rien
public void mouseClicked(MouseEvent leMouseEvent) {}

public void mouseEntered(MouseEvent leMouseEvent) {}

public void mouseExited(MouseEvent leMouseEvent) {}

public void mouseReleased(MouseEvent leMouseEvent) {}
}
```

Tel qu'illustré à la section précédente, la classe *JPanelPourMondeJeuSimple* anime le *MondeDuJeu* par la stratégie du chronomètre. Elle implémente donc l'interface java.awt.event.**ActionListener** et la méthode **actionPerformed**(**ActionEvent** e) pour répondre aux événement du *Timer* :

```
public void actionPerformed( ActionEvent e){
    repaint();
    // Produire la prochaine scène du monde à animer
    leMondeDuJeu.prochaineScene();
}
```

La classe *JPanelPourMondeJeuSimple* est désignée comme écouteur de l'événement du click de la souris. Nous avons déjà illustré le mécanisme d'écouteur au chapitre 5. Nous avons vu qu'il faut désigner un objet qui implémente l'interface java.awt.event.**MouseListener** et les méthodes correspondantes. La ligne suivante désigne l'objet de la classe *JPanelPourMondeJeuSimple* comme écouteur des événements de la souris :

```
addMouseListener (this);
```

La méthode **mousePressed**(**MouseEvent** e) spécifie la réponse à l'événement de click de la souris :

```
public void mousePressed(MouseEvent leMouseEvent){
  leMondeDuJeu.mousePressed(leMouseEvent);
}
```

Cette méthode délègue le travail à effectuer au *MondeDuJeu* en appelant la méthode de même nom. La classe *JFrameIncluantJPanelMondeDuJeu* est la fenêtre qui contient le **JPanel** à animer.

JavaPasAPas/JeuSimple/ JFrameIncluantJPanelMondeDuJeu.java

```
package JeuSimple;

import javax.swing.JFrame;

public class JFrameIncluantJPanelMondeDuJeu extends JFrame {

 public JFrameIncluantJPanelMondeDuJeu() {
  super("Jeu simple");
  JPanelPourMondeJeuSimple leJPanelAnimation = new JPanelPourMondeJeuSimple();
  this.getContentPane().add(leJPanelAnimation);
  this.setDefaultCloseOperation(EXIT_ON_CLOSE);
  this.setSize(
    JPanelPourMondeJeuSimple.LARGEURJPANEL, JPanelPourMondeJeuSimple.HAUTEURJPANEL + 60);
  this.setVisible(true);
  leJPanelAnimation.start();
 }

 public static void main(String args[]) {
  new JFrameIncluantJPanelMondeDuJeu();
 }
}
```

7.4 Génériques

Introduite à la version 5 de Java, la généricité permet d'employer un type comme valeur de paramètre. Cette possibilité est exploitée en particulier avec les collections génériques. Il est ainsi possible de préciser le type des éléments d'une collection à sa déclaration. Dans l'exemple suivant, un vecteur est déclaré en spécifiant que les éléments sont des objets de la classe *EntiteAnimeAvecCri*.

```
Vector<EntiteAnimeAvecCri> vecteurEntites;
```

Le constructeur peut aussi spécifier le type des éléments :

```
vecteurEntites = new Vector<EntiteAnimeAvecCri>();
```

Le *for* permet l'itération sur les éléments de la collection sans avoir à passer explicitement par un *Iterator* et une conversion de type tel qu'illustré par l'exemple suivant :

```
for(EntiteAnimeAvecCri uneEntite : vecteurEntites){
       uneEntite.prochaineScene(LARGEURMONDE,HAUTEURMONDE);
}
```

L'exemple suivant est une version du monde du jeu qui exploite une collection générique d'entités animées.

```
package JeuSimple;
/*
 * MondeDuJeuVectorGen.java
 * Plusieurs bonhommes dans un Vector<EntiteAnime>
 * Le Iti vole (changement du dessin a chaque etat)
 * Ajoute un coq (animation par séquence d'images)
 */
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.util.*;
import javax.swing.*;

public class MondeDuJeuVectorGen {

 // Taille du monde
 public static final int LARGEURMONDE = 1000;
 public static final int HAUTEURMONDE = 1000;

 protected Vector<EntiteAnimeAvecCri> vecteurEntites;
```

```
public MondeDuJeuVectorGen() {
 vecteurEntites = new Vector<EntiteAnimeAvecCri>();
 vecteurEntites.addElement(new BotAnimeAvecCri(10, 100, 100, 200, 3, 3, true, "Son2.wav"));
 vecteurEntites.addElement(new ItiAnimeAvecCriVolant(200, 50, 100, 200, 3, 0, true, "Son3.wav"));
 vecteurEntites.addElement(new KennyAnimeAvecCri(100, 50, 100, 200, 3, 9, true, "Son1.wav"));
 vecteurEntites.addElement(
    new EntiteAnimeAvecCriEtSequenceImages(
       50, 100, 300, 300, 5, 5, true, "Son4.wav", 9, "coq"));
 vecteurEntites.addElement(
    new EntiteAnimeAvecCriEtSequenceImages(
       175, 800, 200, 200, 0, 0, true, "invince.wav", 6, "homer"));
}

public void prochaineScene() {
 for (EntiteAnimeAvecCri uneEntiteAnime : vecteurEntites) {
  uneEntiteAnime.prochaineScene(LARGEURMONDE, HAUTEURMONDE);
 }
}

public void paint(Graphics g) {
 for (EntiteAnimeAvecCri uneEntiteAnime : vecteurEntites) {
  uneEntiteAnime.paintSiVisible(g);
 }
}
// Si une entité est cliquée, elle disparait en poussant un cri
public void mousePressed(MouseEvent e) {
 for (EntiteAnimeAvecCri uneEntiteAnime : vecteurEntites) {
  if (uneEntiteAnime.touche(e.getX(), e.getY())) {
   uneEntiteAnime.setVisible(false);
   uneEntiteAnime.crier();
  }
 }
}
```

7.5 Autres collections

En plus de la classe **Vector**, Java offre plusieurs autres classes pour manipuler des collections d'objets : *Set*, *ArrayList*, *LinkedList*, *Map*, *Queue*, etc. Chacune des classes possèdent des avantages particuliers pour la manipulation de groupes d'objets. La classe **Vector** a la particularité d'être synchronisée (*Synchronized*), ce qui signifie qu'elle peut être partagée entre plusieurs fils (*Thread*) parallèles. Les mécanismes de contrôle de concurrence employés à cet effet entraînent une surcharge de calcul dans le cas où ce partage n'est pas nécessaire. Java offre plusieurs classes Collection au-delà de Vector qui ne sont pas synchronisées et qui évitent cette surcharge de travail. Dans notre exemple de jeu, il aurait été plus judicieux d'employer la classe **ArrayList**.

Exercice. Reprendre le jeu avec une collection **ArrayList** générique.

8. Traitement de fichiers

Les données en mémoire Java ne sont pas conservées après la fin du programme. Pour conserver des données de manière persistante à long terme, il faut employer les mémoires secondaires. Les langages de programmation fournissent des interfaces programmatiques pour la manipulation des fichiers en mémoire secondaire. Ces interfaces réalisent des abstractions simples qui isolent le client de plusieurs des détails de bas niveau des mécanismes des mémoires secondaires. La simplicité de ces interfaces est pertinente pour le développement d'applications basiques nécessitant la persistance des données. Par contre, ces interfaces sont souvent trop limitées pour des applications complexes nécessitant des services plus sophistiqués. Les systèmes de gestion de bases de données sont prévus à cet effet.

Dans le cas de Java, le package java.io contient une hiérarchie élaborée de classes dédiées aux entrées-sorties sur fichier et sur d'autres types de flux de données. Dans ce chapitre, l'emphase est mise sur l'utilisation des classes pour le traitement de fichiers. Les classes de java.io permettent d'utiliser les fichiers selon deux modes de base :

- par *accès sériel* comme des flux de données (*stream*). Les octets du fichier sont lus ou écrits en série les uns après les autres.

- par *accès direct* (*random access*). Les octets peuvent être lus ou écrits dans un ordre quelconque.

Dans le cas de l'accès par flux, des méthodes permettent d'itérer sur les données de manière sérielle les unes après les autres. Un flux d'entrée correspondant à la classe abstraite **InputStream** permet de lire une suite d'octets. Un flux de sortie correspondant à la classe abstraite **OutputStream** permet d'écrire une suite d'octets. Comme illustré à la figure suivante, la méthode **read**() de **InputStream** lit le prochain octet du flux d'entrée et la méthode **write**(int b) de **OutputStream** ajoute un octet au flux de sortie.

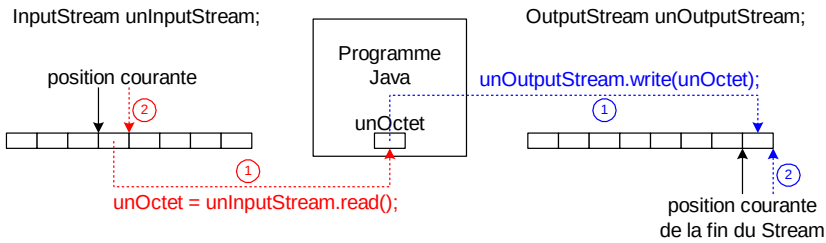


Figure 32. Concept d'*InputStream* et d'*OutputStream*.

Cette abstraction cache plusieurs détails de réalisation du stockage des données sous forme de fichiers. Ainsi le programme Java n'a pas à se préoccuper de la structure physique des unités périphériques, de l'allocation d'espace au fichier, etc. Le fichier apparaît tout simplement comme une série d'octets. Le package *java.io* fournit ainsi une abstraction de base qui permet l'accès au niveau octet en traduisant les opérations sur les octets en termes des opérations du niveau de la structure physique de l'unité périphérique.

Les flux sont utilisés non seulement pour l'accès aux mémoires secondaires mais aussi pour d'autres unités périphériques tel que le clavier, l'écran, l'imprimante, le réseau, etc. L'accès à un flux est toujours sériel. Ceci est parfois insuffisant, par exemple, dans le cas d'une application qui doit pouvoir accéder directement à une donnée sans devoir itérer sur tout le fichier. Pour obtenir cette souplesse d'accès, il faut employer la classe **RandomAccessFile** qui permet un accès direct à n'importe quel octet d'un fichier.

La Figure 33 montre les classes pour les flux d'entrée d'octets. La Figure 34 montre les classes des flux de sortie d'octets. La Figure 35 montre la classe **RandomAccessFile** qui sert à la fois d'entrée et de sortie. Les flux de base **InputStream** et **OutputStream** agissent au niveau binaire en terme d'octets. Le programme voit l'unité d'entrée ou de sortie comme une série d'octets. Dans le cas d'un fichier, le programme ne voit donc pas de données structurées mais uniquement une suite de bits regroupés en octets.

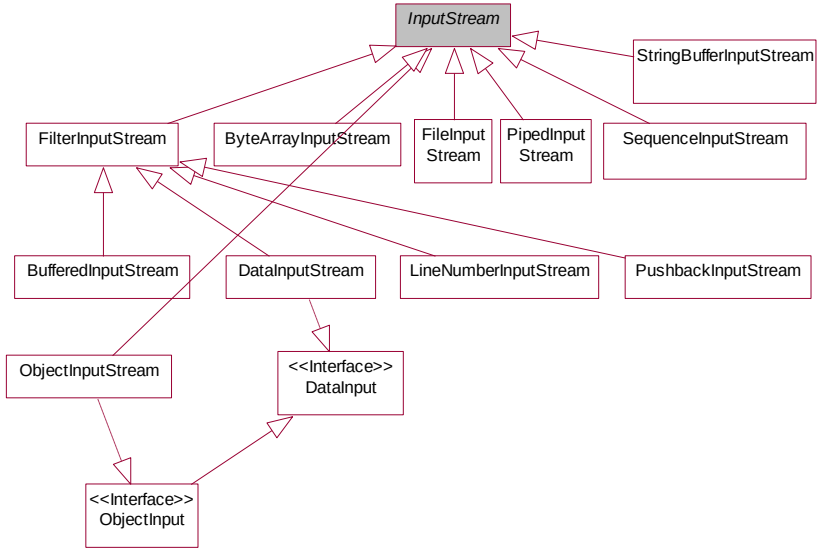


Figure 33. Classes de java.io pour les flux d'entrée d'octets (*InputStream*).

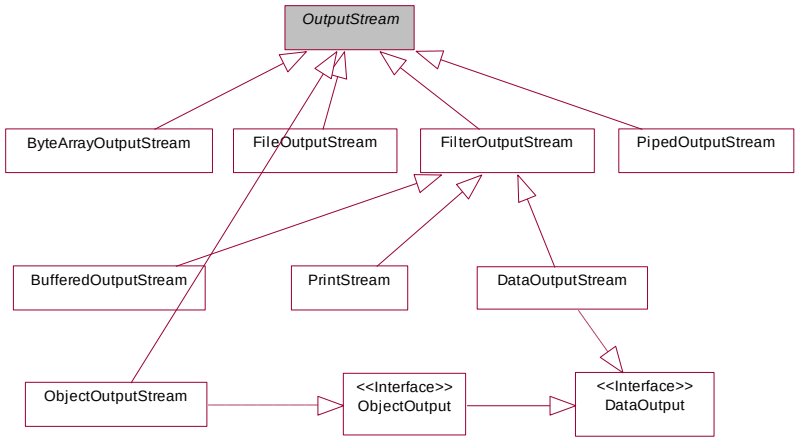


Figure 34. Classes de java.io pour les flux de sortie d'octets.

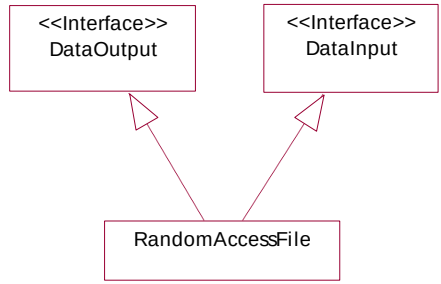


Figure 35. Classe *RandomAccessFile*.

- Lecture d'un flux d'octets provenant d'un fichier

L'exemple suivant introduit les concepts de base concernant la lecture d'un fichier vu comme une suite d'octets. Le programme utilise la classe **FileInputStream**, sous-classe de **InputStream**, dédiée au traitement des fichiers.

Exemple. Le programme Java suivant lit le contenu du fichier en consommant les octets, un octet à la fois, par l'utilisation de la classe **FileInputStream** et compte le nombre d'octets contenus dans le fichier.

JavaPasAPas/chapitre_8/CompterOctetsFichier.java

```
/* Lire un fichier et en compter le nombre d'octets */

import java.io.*;

public class CompterOctetsFichier {
 public static void main(String args[]) {
   int unOctet;
   int compteurOctet;
   FileInputStream unFichier;
   try {
     unFichier = new FileInputStream("Fichier1.txt");
     compteurOctet = 0;
     while ((unOctet = unFichier.read()) != -1) compteurOctet++;
     unFichier.close();
     System.out.println("Nombre d'octets du fichier Fichier1.txt : " + compteurOctet);
   } catch (IOException e) {
     System.err.println("Exception\n" + e.toString());
   }
 }
}
```

L'énoncé suivant crée un objet de la classe **FileInputStream** en ouvrant le fichier dont le chemin est passé en paramètre au constructeur :

```
            unFichier =
new FileInputStream("Fichier1.txt");
```

Si le fichier est inexistant, une exception est levée. Le constructeur de la classe **FileInputStream** retourne une référence à un objet qui représente le fichier[27]. Ainsi la variable *unFichier* est par la suite utilisée pour faire référence au fichier comme dans :

```
        unOctet = unFichier.read()
```

La méthode **read**() de la classe **FileInputStream** retourne le prochain octet lu sous forme d'un entier. Pour lire tous les octets du fichier un par un, la lecture est incluse dans une boucle. Le fichier apparaît ainsi au programme comme un flux d'octets. Par convention, lorsque la fin du fichier est atteinte, la valeur –1 est retournée[28] par **read**(). La boucle incrémente le compteur *compteurOctet* à chacune des itérations.

Enfin, la méthode **close**() ferme le fichier :

```
        unFichier.close();
```

Après avoir fermé le fichier, les ressources associées sont libérées et le fichier n'est plus accessible par le programme.

Exercice. Comptez le nombre d'occurrences de la lettre « a » dans le fichier.

L'exemple suivant étend l'exemple de lecture d'un fichier en écrivant les octets lus dans un nouveau fichier.

[27] . A noter que le«/» est utilisé plutôt que le «\» dans le chemin de fichier même si la plate-forme est Windows. Les deux formes de séparateur sont acceptées.

[28] Cependant, la fin de fichier est habituellement représentée par un code particulier du jeu de caractères qui n'est pas réellement la valeur entière –1. Ce code peut varier selon la plate-forme sous-jacente. Le *read*() retourne –1 afin de faciliter l'indépendance du code vis-à-vis la plate-forme.

Exemple.

JavaPasAPas/chapitre_8/CopierFichier.java

```
/* Copier un fichier octet par octet */

import java.io.*;

public class CopierFichier {
  public static void main(String args[]) {
    int unOctet;
    FileInputStream unFileInputStream;
    FileOutputStream unFileOutputStream;
    try {
      unFileInputStream = new FileInputStream("Fichier1.txt");
      unFileOutputStream = new FileOutputStream("Fichier2.txt");
      while ((unOctet = unFileInputStream.read()) != -1) unFileOutputStream.write(unOctet);
      unFileInputStream.close();
      unFileOutputStream.close();
    } catch (IOException e) {
      System.err.println("Exception\n" + e.toString());
    }
  }
}
```

L'énoncé

```
new FileOutputStream("Fichier2.txt");
```

ouvre le fichier désigné par son chemin en vue de l'écriture. Si le fichier n'existe pas, il est créé. S'il existe déjà, le contenu précédent sera détruit et remplacé par les octets écrits par la suite.

La méthode **write**(int b) de **FileOutputStream** écrit l'octet lu dans le fichier :

```
unFileOutputStream.write(unOctet);
```

Seul l'octet le moins significatif de l'entier est employé.

8.1 Fichier binaire (FileOutputStream, FileInputStream)

Cette section montre comment conserver et récupérer une donnée dans un fichier en employant les opérations au niveau octet.

Exemple. Supposons que l'on veuille écrire un nombre entier (*int*) dans un fichier afin de le récupérer par la suite. Malheureusement, la classe **FileOutputStream** ne permet d'écrire que des octets qui ne sont pas interprétés comme des données d'un type de plus haut niveau comme un

entier, un réel ou une chaîne de caractères[29]. Comme l'interface est en termes d'octets, il faut d'abord convertir l'entier à écrire sous forme du tableau *tampon* de quatre octets. Le tableau *tampon* est ensuite écrit octet par octet par *write*(*tampon*).

JavaPasAPas/chapitre_8/EcrireEntierEnOctets.java

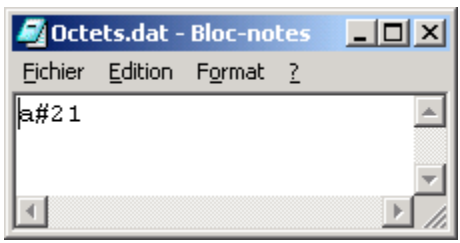
```
/* Creation d'un fichier et écriture d'un entier sous forme d'une suite d'octets dans le fichier */
import java.io.*;

public class EcrireEntierEnOctets {
  public static void main(String args[]) {
    FileOutputStream unFichier;
    try {
      unFichier = new FileOutputStream("Octets.dat");

      int unEntier = 1629696561; // (97*2^24)+(35*2^16)+(50<<2^8)+49 = "a#21" en String;
      // Convertir unEntier en un tableau de 4 octets
      byte[] tampon = new byte[4];
      for (int i = 3; i >= 0; i--) {
        tampon[i] = (byte) (unEntier & 0XFF); // Extrait l'octet le moins significatif
        unEntier >>>= 8; // Décalage de 8 bits (remplissage à 0)
      }
      unFichier.write(tampon);
      unFichier.close();
    } catch (IOException e) {
      System.err.println("Exception\n" + e.toString());
    }
  }
}
```

L'énoncé

```
            unFichier = new FileOutputStream("Octets.dat");
```

ouvre le fichier.

Par ailleurs, si ce fichier est ouvert avec l'éditeur de texte *Bloc-notes* de *Windows*, le contenu semble incompréhensible parce que le programme *Bloc-notes* interprète le contenu du fichier comme une suite caractères et non un entier de 4 octets :



[29] Il n'y a pas de méthode *writeInt*(*int* unEntier) supportée par *FileOutputStream*.

On dit souvent de ce genre de fichier qu'il est *binaire* par opposition à un fichier de type *texte* que nous étudierons plus loin. Pour lire le fichier avec **FileInputStream**, et l'interpréter correctement, il faut convertir les octets lus en entier.

Exemple.

JavaPasAPas/chapitre_8/LireEntierEnOctets.java

```
/* Lecture dans le fichier d'un entier sous forme d'une suite d'octets et conversion en int */

import java.io.*;

public class LireEntierEnOctets {
 public static void main(String args[]) {
  FileInputStream unFichier;
  try {
   unFichier = new FileInputStream("Octets.dat");

   byte[] tampon = new byte[4];
   unFichier.read(tampon); // Lecture des 4 octets

   // Convertir le tableau d'octets tampon en int unEntier
   int unEntier = 0;
   for (int i = 0; i <= 3; i++) {
    unEntier <<= 8;
    unEntier += ((int) tampon[i]) & 0XFF;
   }
   unFichier.close();
   System.out.println("Valeur décimale de l'entier : " + unEntier);
  } catch (IOException e) {
   System.err.println("Exception\n" + e.toString());
  }
 }
}
```

Résultat :

```
Valeur décimale de l'entier : 1629696561
```

Exercice. Etendre l'exemple précédent en écrivant une suite de 3 entiers dans un fichier. Ensuite relire les trois entiers du fichier dans un autre programme.

Les classes **InputStream** et **OutputStream** sont rarement utilisées directement à cause des conversions fastidieuses à effectuer comme l'illustre les exemples précédents. D'autres sous-classes fournissent des abstractions de plus haut niveau. Les classes **DataInputStream** et **DataOutputStream** permettent de lire et d'écrire directement des types de base (*int*, *long*, *float*,…) sans avoir à les convertir en suite d'octets. Les classes **ObjectInputStream** et **ObjectOutputStream** permettent de lire et d'écrire directement des objets.

D'autre part, les classes **Reader** et **Writer** permettent de traiter les fichiers de type texte qui sont directement lisibles par les humains.

8.2 DataInputStream et DataOutputStream

Les classes **DataInputStream** et **DataOutputStream** permettent de lire et d'écrire les types de base. La figure suivante montre les méthodes supportées.

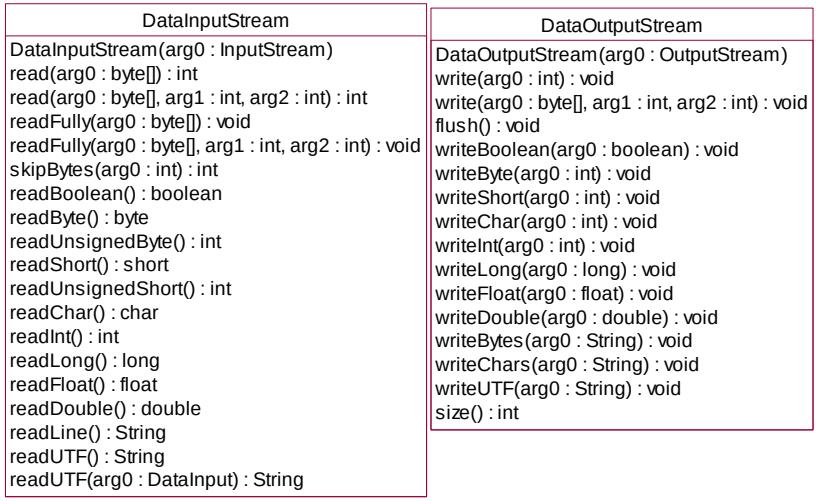


Figure 36. Méthodes des classes *DataInputStream* et *DataOutputStream*.

Pour illustrer l'utilisation de ces classes, reprenons l'exemple précédent d'écriture et de lecture d'un entier.

Exemple. Comme dans l'exemple précédent, le programme suivant stocke un entier dans un fichier, mais cette fois-ci, en utilisant la méthode **writeInt**(int v) de la classe **DataOutputStream** évitant ainsi la conversion en suite d'octets.

JavaPasAPas/chapitre_8/EcrireEntier.java

```
/* création d'un DataOutputStream à partir d'un fichier et écriture d'un entier dans le fichier */
package LivreJava;

import java.io.*;
```

```
public class EcrireEntier {
  public static void main(String args[]) {
    DataOutputStream unFichier;
    try {
      unFichier = new DataOutputStream(new FileOutputStream("UnEntier.dat"));
      int unEntier = 1629696561;
      // (97*2^24)+(35*2^16)+(50<<2^8)+49 = "a#21" en String;

      unFichier.writeInt(unEntier);
      unFichier.close();

    } catch (IOException e) {
      System.err.println("Exception\n" + e.toString());
    }
  }
}
```

L'objet **DataOutputStream** est construit à partir d'un **FileOutputStream** avec l'instruction[30] :

```
                new DataOutputStream(
        new FileOutputStream("UnEntier.dat"));
```

Il est ensuite possible d'écrire directement l'entier dans le fichier avec :

```
            unFichier.writeInt(unEntier);
```

Exemple. Le programme suivant lit l'entier avec **readInt**().

JavaPasAPas/chapitre_8/LireEntier.java

```
/* Lecture dans le fichier d'un entier à l'aide d'un DataInputStream */
import java.io.*;

public class LireEntier {
  public static void main(String args[]) {
    DataInputStream unFichier;
    try {
      unFichier = new DataInputStream(new FileInputStream("UnEntier.dat"));
      int unEntier = unFichier.readInt();
      unFichier.close();
      System.out.println("Valeur décimale de l'entier : " + unEntier);
    } catch (IOException e) {
      System.err.println("Exception\n" + e.toString());
    }
  }
}
```

[30] Cette manière d'ajouter des fonctionnalités correspond au patron bien connu *décorateur*.

Exercice. Etendre l'exemple précédent en écrivant une suite de 3 entiers dans un fichier. Ensuite relire les trois entiers du fichier dans un autre programme.

8.3 Fichier texte

Les classes abstraites **Reader** et **Writer** sont analogues aux classes **InputStream** et **OutputStream** sauf qu'elles interprètent les flux d'octets comme des suites de caractères d'un jeu de caractère particulier. Elles gèrent plusieurs jeux de caractères standards (ASCII, ISO-Latin-1, Unicode, etc.). Par défaut, le jeu de caractère de la plate-forme sous-jacente est utilisé. La conversion des octets en caractères est effectuée par ces classes.

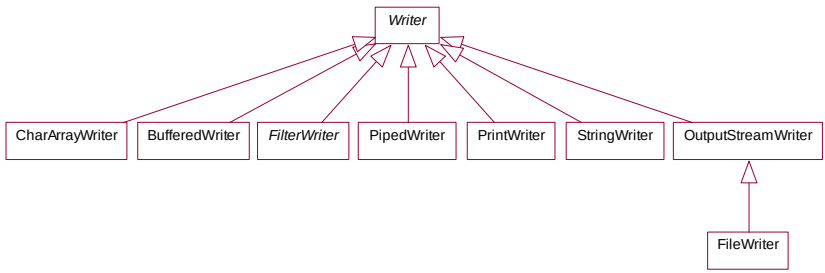


Figure 37. Sous-hiérarchie des classes Writer.

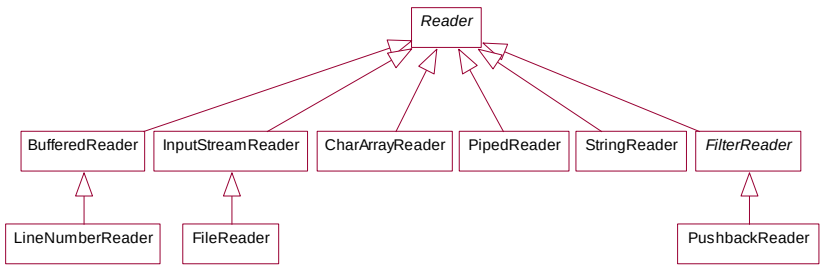


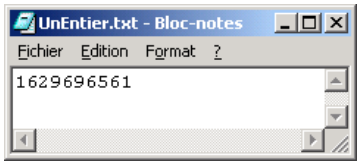
Figure 38. Sous-hiérarchie des classes *Reader*.

Exemple. L'exemple suivant écrit une chaîne de caractère qui représente un entier dans un fichier.

JavaPasAPas/chapitre_8/EcrireEntierTexte.java

```
/* création d'un FileWriter à partir d'un fichier et écriture d'un entier dans le fichier
sous forme d'une chaîne de caractères */
import java.io.*;

public class EcrireEntierTexte {
  public static void main(String args[]) {
    FileWriter unFichier;
    try {
      unFichier = new FileWriter("UnEntier.txt");
      unFichier.write("1629696561");
      unFichier.close();
    } catch (IOException e) {
      System.err.println("Exception\n" + e.toString());
    }
  }
}
```

Comme le contenu du fichier *UnEntier.txt* est sous forme de texte, il peut être consulté avec un éditeur de texte. En ouvrant ce fichier avec l'éditeur Notepad, on voit donc l'entier sous une forme lisible :



Le même effet est obtenu par le programme suivant qui utilise plutôt un **PrintWriter**. Ceci permet d'écrire un *int* qui sera automatiquement converti sous forme d'une chaîne de caractère. Le *print*() accepte tous les types de base.

Exemple.

JavaPasAPas/chapitre_8/EcrireEntierTextePrintWriter.java

```
/* création d'un PrintWriter à partir d'un fichier et écriture d'un entier dans le fichier
sous forme d'une chaîne de caractères */

import java.io.*;
```

```
public class EcrireEntierTextePrintWriter {
  public static void main(String args[]) {
    FileWriter unFichier;
    PrintWriter unPrintWriter;
    try {
      unFichier = new FileWriter("UnEntierPW.txt");
      unPrintWriter = new PrintWriter(unFichier);
      unPrintWriter.print(1629696561);
      System.out.println(unFichier.getEncoding());
      unFichier.close();
    } catch (IOException e) {
      System.err.println("Exception\n" + e.toString());
    }
  }
}
```

PrintStream

La classe **PrintStream** permet aussi de faire un *print* des types de base. Pour des raisons historiques, les objets prédéfinis *System.out* (sortie standard, habituellement l'écran) et *System.err* sont des **PrintStream** plutôt que des **PrintWriter**.

Exemple. Le programme suivant lit l'entier sous forme de texte dans un tableau de caractères. Ensuite le tableau est converti en un *int* en passant par la méthode **Integer.parseInt(String** s).

JavaPasAPas/chapitre_8/LireEntierTexte.java

```
/* Lecture dans le fichier d'un entier sous forme de texte à l'aide d'un FileReader */

import java.io.*;

public class LireEntierTexte {
  public static void main(String args[]) {
    FileReader unFichier;
    try {
      char[] tableauChar = new char[10];
      unFichier = new FileReader("UnEntier.txt");
      unFichier.read(tableauChar, 0, 10);
      int unEntier = Integer.parseInt(new String(tableauChar, 0, 10));
      unFichier.close();
      System.out.println("Valeur décimale de l'entier : " + unEntier);
    } catch (IOException e) {
      System.err.println("Exception\n" + e.toString());
    }
  }
}
```

Une difficulté importante de la lecture de données sous forme de texte est que la taille de la donnée à lire n'est pas toujours connue à l'avance et peut

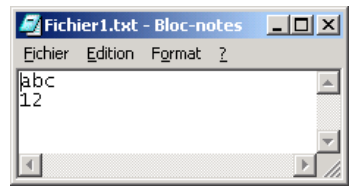
être variable. Dans l'appel à **read**(char[] cbuf, int off, int len), il faut spécifier l'indice du début de la chaîne et nombre de caractères à lire :

```
        unFichier.read(tableauChar,0,10);
```

Si le nombre de caractères utilisé pour représenter l'entier est différent de 10, le programme ne donnera pas le bon résultat. Habituellement, lorsque des données sont placées dans un fichier texte, la fin d'une donnée est spécifiée par un délimiteur déterminé par une convention prédéfinie. Dans notre petit exemple, on pourrait spécifier que la fin de l'entier est déterminée par la fin du fichier. Le programme pourrait alors lire caractère par caractère jusqu'à ce que la fin du fichier soit atteinte. Le cas général du découpage d'un texte en éléments délimités est étudié plus loin.

8.3.1 Représentation interne des caractères et traitement des fins de ligne

Supposons que le fichier *Fichier1.txt* soit édité avec l'éditeur de texte *Notepad* et que le contenu soit celui affiché à la figure suivante :



L'éditeur de texte nous affiche le contenu du fichier sous forme de caractères. En réalité, le contenu du fichier est une séquence de bits. Avec *Notepad* sous *Windows*, par défaut, le jeu de caractère utilisé est le code ASCII. Chacun des caractères du texte est représenté par un code ASCII de huit bits (un octet). La première ligne de l'exemple suivant montre le contenu réel du fichier. La deuxième ligne indique les caractères correspondants.

Exemple. Contenu du fichier *Fichier1.txt* en binaire.

```
01100001 01100010 01100011 00001101 00001010 00110001 00110010
00001101 00001010

a        b        c        \r       \n       1        2        \r
\n
```

Sous Windows, la fin de ligne est représentée par la séquence des caractères spéciaux ASCII, *retour de chariot* (noté <CR> ou '\r' en Java) et *saut de ligne*

(<LF> ou '\n' en Java)[31]. Un code supplémentaire, absent dans l'exemple, est ajouté à la fin de la séquence pour indiquer la fin du fichier (0X0A sur Windows).

Avec cet exemple, le programme *CompterOctetsFichier* afficherait le résultat suivant à l'écran :

```
Nombre d'octets du fichier Fichier1.txt : 9
```

Pour un **FileInputStream**, les octets lus n'ont pas de signification particulière. Les octets représentant les caractères de contrôle, de retour de chariot ('\r') et de saut de ligne ('\n') sont des octets comme les autres et ils sont donc comptés par le programme. C'est pourquoi même si à l'écran de *Notepad*, on ne voit que 5 caractères, chacun étant représenté par un octet ASCII, le programme compte 9 octets. Un **InputStream** voit donc un fichier sous forme binaire, c'est-à-dire sans interpréter les octets lus.

Exemple. Le fichier *Fichier1.txt* pourrait être produit en écrivant directement les octets à l'aide d'un **FileOutputStream.**

JavaPasAPas/chapitre_8/EcrireOctetsFichier.java

```
/* création d'un fichier et écriture d'un suite d'octets dans le fichier */
import java.io.*;

public class EcrireOctetsFichier {
 public static void main(String args[]) {
   FileOutputStream unFichier;
   try {
     unFichier = new FileOutputStream("Fichier1.txt");

     unFichier.write(0X61);
     unFichier.write(0X62);
     unFichier.write(0X63);
     unFichier.write(0X0D);
     unFichier.write(0X0A);
     unFichier.write(0X31);
     unFichier.write(0X32);
     unFichier.write(0X0D);
     unFichier.write(0X0A);

     unFichier.close();
   } catch (IOException e) {
     System.err.println("Exception\n" + e.toString());
```

[31] La manière de représenter les fins de ligne peut différer en fonction de la plate-forme. Unix, par exemple, emploie uniquement le saut de ligne (<LF>).

```
  }
 }
}
```

On peut produire le même effet avec les classes **PrintWriter** et *PrintStream* qui possèdent des méthodes pour traiter les fins de ligne. En particulier, les méthodes *println*() sont analogues aux méthodes *print*() et ajoutent une fin de ligne après la donnée.

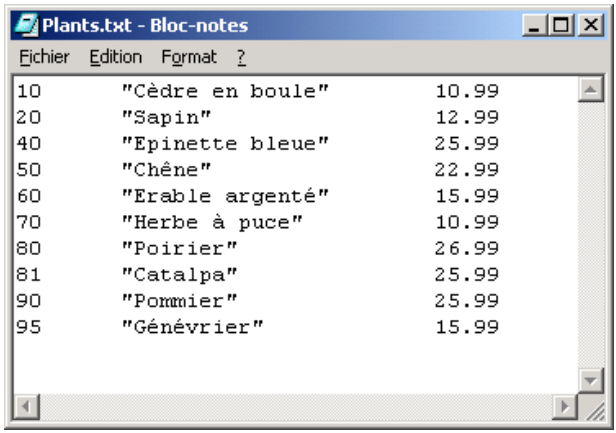
Exemple. Production de *Fichier1.txt* avec un **PrintWriter**.

JavaPasAPas/chapitre_8/EcrireTexteabc12.java

```
/* création d'un PrintWriter à partir d'un fichier et écriture de texte avec println */

import java.io.*;

public class EcrireTexteabc12 {
 public static void main(String args[]) {
  FileWriter unFichier;
  PrintWriter unPrintWriter;

  try {
   unFichier = new FileWriter("Fichier1.txt");
   unPrintWriter = new PrintWriter(unFichier);
   unPrintWriter.println("abc");
   unPrintWriter.println(12);
   unFichier.close();
  } catch (IOException e) {
   System.err.println("Exception\n" + e.toString());
  }
 }
}
```

8.3.2 Analyse lexicale avec la classe StreamTokenizer

Un fichier texte est souvent utilisé pour saisir des données ou pour échanger des données entre applications. Les données sont alors placées en séquence selon un format prédéfini. Par exemple, supposons que le fichier *plants.txt* contienne des données sur les plants du catalogue de la pépinière *PleinDeFoin* sous la forme suivante :



On peut imaginer que les données ont été saisies à l'aide d'un éditeur de texte conventionnel. Ces données correspondent au numéro de catalogue, nom et prix d'un ensemble de plants. Supposons que les conventions suivantes ont été établies. Une donnée se termine par un ou plusieurs caractères parmi les suivants : espaces (' '), tabulation ('\t'), retour de chariot('\r'), fin de ligne ('\n'). Ces caractères jouent le rôle de *délimiteurs*. Un cas particulier est souvent nécessaire pour les chaînes de caractères qui peuvent contenir des délimiteurs. C'est le cas des descriptions de plants dans notre exemple. Par convention, elles sont encadrées par des guillemets (") qui jouent ainsi le rôle de *délimiteur de chaîne*. Le découpage du texte en ses éléments constitutifs, appelés *jetons* (*token*) est un problème bien connu, appelé analyse lexicale, qui fait appel à des techniques éprouvées. La classe **StreamTokenizer** incorpore les algorithmes nécessaires.

Exemple. Le programme suivant illustre l'utilisation d'un **StreamTokenizer** qui permet de découper un texte en jetons. Les données lues sont stockées dans un vecteur d'objets de la classe *Plant* et sont affichées à l'écran.

JavaPasAPas/chapitre_8/ExempleStreamTokenizer.java

```
/* Illustration du StreamTokenizer
 * Lit le fichier plants.txt, affiche à l'écran chacun des jetons (noPlant,description,prixUnitaire) et
 * stocke le contenu dans un vecteur d'objets Plant */

import java.io.*;
import java.util.*;

public class ExempleStreamTokenizer {
  public static void main(String args[]) {
    try {
      FileReader unFichier = new FileReader("Plants.txt");
      StreamTokenizer unStreamTokenizer = new StreamTokenizer(unFichier);
```

```
    // Les 5 lignes suivantes ne sont pas nécessaires car les paramètres
    // donnés sont les valeurs de défaut
    unStreamTokenizer.quoteChar((int) '"');
    unStreamTokenizer.whitespaceChars((int) '\r', (int) '\r');
    unStreamTokenizer.whitespaceChars((int) '\n', (int) '\n');
    unStreamTokenizer.whitespaceChars((int) '\t', (int) '\t');
    unStreamTokenizer.whitespaceChars((int) ' ', (int) ' ');

    Vector vecteurDePlants = new Vector();
    int noPlant = 0;
    String description = "";
    double prixUnitaire = 0.0;

    while (unStreamTokenizer.nextToken() != StreamTokenizer.TT_EOF) { // fin du fichier ?
     // Lecture du noPlant
     if (unStreamTokenizer.ttype == StreamTokenizer.TT_NUMBER) { // Est-ce bien un nombre ?
      noPlant = (int) unStreamTokenizer.nval; // nval est un double !
     } else {
      System.out.println("Le format du fichier est incorrect : noPlant attendu");
      System.exit(1);
     }

     // Lecture de la description
     unStreamTokenizer.nextToken();
     if (unStreamTokenizer.ttype == (int) '"') { // Est-ce bien une chaîne encadrée par " ?
      description = unStreamTokenizer.sval;
     } else {
      System.out.println("Le format du fichier est incorrect : description attendue");
      System.exit(1);
     }
     // Lecture du prixUnitaire
     unStreamTokenizer.nextToken();
     if (unStreamTokenizer.ttype == StreamTokenizer.TT_NUMBER) { // Est-ce bien un nombre ?
      prixUnitaire = unStreamTokenizer.nval;
     } else {
      System.out.println("Le format du fichier est incorrect : prix attendu");
      System.exit(1);
     }

     // création de l'objet Plant
     Plant unPlant = new Plant(noPlant, description, prixUnitaire);
     System.out.println(noPlant + " " + description + " " + prixUnitaire);
     vecteurDePlants.addElement(unPlant);
    }
    unFichier.close();
   } catch (IOException e) {
    System.err.println("Exception\n" + e.toString());
   }
  }
}
```

JavaPasAPas/chapitre_8/Plant.java

```
import java.io.*;

public class Plant implements Serializable {
  private int noPlant; // numéro de catalogue du plant
  private String description; // description du plant
  private double prixUnitaire; // prix unitaire du plant

  public Plant(int noPlant, String description, double prixUnitaire) {
    this.noPlant = noPlant;
```

```
  this.description = description;
  this.prixUnitaire = prixUnitaire;
 }

 public void setNoPlant(int noPlant) {
  this.noPlant = noPlant;
 }

 public int getNoPlant() {
  return noPlant;
 }

 public void setDescription(String description) {
  this.description = description;
 }

 public String getDescription() {
  return description;
 }

 public void setPrixUnitaire(double prixUnitaire) {
  this.prixUnitaire = prixUnitaire;
 }

 public double getPrixUnitaire() {
  return prixUnitaire;
 }

 public void ecrireEnregistrementTailleMax(RandomAccessFile unFichier) throws Exception {
  unFichier.writeInt(noPlant); // 4 octets
  if (description.length() > 38) {
   System.exit(1);
  }
  unFichier.writeInt(description.length()); // 4octets
  unFichier.writeBytes(description); // max 38 octets
  unFichier.writeDouble(prixUnitaire); // 8 octets
 }

 public void lireEnregistrementTailleMax(RandomAccessFile unFichier) throws Exception {
  noPlant = unFichier.readInt();
  int tailleDescription = unFichier.readInt();
  byte[] tampon = new byte[tailleDescription];
  unFichier.readFully(tampon);
  description = new String(tampon);
  prixUnitaire = unFichier.readDouble();
 }

 public static int tailleMaxEnregistrement() {
  return 50;
 }
}
```

La méthode **nextToken()** retourne le type du prochain jeton rencontré et stocke cette valeur dans la variable publique *ttype*. Les types possibles sont les suivants :

- *StreamTokenizer.TT_WORD* : une chaîne a été rencontré. La valeur est stockée dans la variable String *sval*.

- *StreamTokenizer.TT_NUMBER* : un nombre a été rencontré. La valeur est stockée dans la variable double *nval*.

- *StreamTokenizer.TT_EOF* : la fin du fichier a été rencontrée.

- *StreamTokenizer.TT_EOL* : une fin de ligne a été rencontrée. Par défaut les fins de ligne ne sont pas traitées. La méthode *eolIsSignificant*(boolean flag) permet d'activer la détection des fins de ligne.

- Lorsqu'une chaîne délimitée par un caractère délimiteur de chaîne (*quoteChar*) est rencontrée, le type retourné est la valeur entière du caractère délimiteur de chaîne.

- Dans le cas de caractères spéciaux, c'est la valeur entière du caractère lui-même qui est retourné.

Plusieurs paramètres qui contrôlent le *StreamTokenizer* peuvent être modifiés au besoin par les méthodes suivantes entre autres :

- **whitespaceChars**(int low, int hi). Spécifie que les caractères de l'intervalle [*low*, *hi*] sont des délimiteurs.

- **quoteChar**(int ch). Spécifie que le caractère *ch* est un délimiteur de chaîne.

Un problème important lors du décodage d'un fichier de texte est la validation de son format. Dans notre petit exemple, le format est assez simple. La validation est rudimentaire et ne tient pas compte de toutes les possibilités. Par exemple, si un nombre réel est rencontré à la place d'un entier, la conversion tronque le réel en entier sans aucun avertissement.

Exercice. Reprendre l'exemple précédent pour un fichier qui contient les noms et numéros de téléphone de contacts fictifs.

Avec la popularité du Web qui est un medium de transmission basé sur le texte, l'échange de données sous forme de fichier de texte prend une importance croissante. Ceci a conduit à l'établissement de la norme XML (*eXtensible Markup Language*). La section suivante introduit XML et des outils Java permettant de manipuler les fichiers XML.

8.3.3 Traitement d'un document XML avec SAX et DOM

XML permet de spécifier la grammaire d'un document textuel par un *Data Type Definition* – DTD ou par un schéma XML.

Exemple. Le contenu du fichier *Plants.xml* suivant est un exemple de document XML avec DTD incluse représentant le catalogue des plants de la pépinière *PleinDeFoin*.

JavaPasAPas/chapitre_8/Plants.xml

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?> <!DOCTYPE Catalogue [
  <!ELEMENT Catalogue (Plant+)>
  <!ELEMENT Plant (noPlant,description,prixUnitaire)>
  <!ELEMENT noPlant (#PCDATA)>
  <!ELEMENT description (#PCDATA)>
  <!ELEMENT prixUnitaire (#PCDATA)>
]>

<Catalogue>
  <Plant>
            <noPlant>10</noPlant>
            <description>Cèdre en boule</description>
            <prixUnitaire>10.99</prixUnitaire>
  </Plant>
  <Plant>
            <noPlant>20</noPlant>
            <description>Sapin</description>
            <prixUnitaire>12.99</prixUnitaire>
  </Plant>
  <Plant>
            <noPlant>40</noPlant>
            <description>Epinette bleue</description>
            <prixUnitaire>25.99</prixUnitaire>
  </Plant>
  <Plant>
            <noPlant>50</noPlant>
            <description>Chêne</description>
            <prixUnitaire>22.99</prixUnitaire>
  </Plant>

  <Plant>
            <noPlant>60</noPlant>
            <description>Erable argenté</description>
            <prixUnitaire>15.99</prixUnitaire>
  </Plant>

  <Plant>
            <noPlant>70</noPlant>
            <description>Erable argenté</description>
            <prixUnitaire>15.99</prixUnitaire>
  </Plant>
  <Plant>
            <noPlant>80</noPlant>
            <description>Herbe à puce</description>
            <prixUnitaire>10.99</prixUnitaire>
  </Plant>

  <Plant>
            <noPlant>81</noPlant>
            <description>Catalpa</description>
            <prixUnitaire>25.99</prixUnitaire>
```

```
  </Plant>
  <Plant>
            <noPlant>90</noPlant>
            <description>Pommier</description>
            <prixUnitaire>25.99</prixUnitaire>
  </Plant>
  <Plant>
            <noPlant>95</noPlant>
            <description>Génévrier</description>
            <prixUnitaire>15.99</prixUnitaire>
  </Plant>
</Catalogue>
```

L'entête

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
```

indique la version utilisée de la norme XML, le jeu de caractère choisi et si le document fait référence à des documents externes. Le DOCTYPE définit un nom de type pour la structure du document :

```
<!DOCTYPE Catalogue [
```

Les clauses ELEMENT décrivent la grammaire du document. Chaque clause

```
!ELEMENT nomElement syntaxeElement
```

définit un élément qui est encodé dans le texte à l'aide d'une paire de *balises* de la forme

```
<nomElement> contenu </nomElement>
```

dont le contenu suit la syntaxe définie par *syntaxeElement*. Par opposition à HTML, les balises XML n'ont pas pour objectif de préciser le style d'affichage des données mais plutôt d'en indiquer la signification.
La clause

```
        <!ELEMENT Catalogue (Plant+)>
```

de la DTD définit l'élément nommé *Catalogue* dont le contenu est un ensemble (au moins un à cause du +) d'éléments *Plant* (au moins un à cause du +).
On peut ainsi observer que le contenu du document lui-même débute par la balise ouvrante

```
        <Catalogue>
```

et se termine par la balise fermante

```
        </Catalogue>
```

Entre les deux se trouvent une suite d'éléments *Plant*.
La clause

```
        <!ELEMENT Plant (noPlant,description,prixUnitaire)>
```

définit qu'un élément *Plant* est composé de trois éléments nommés : *noPlant*, *description* et *prixUnitaire*. On peut observer cette structure dans l'exemple suivant du contenu du document :

```
<Plant>
   <noPlant>10</noPlant>
   <description>Cèdre en boule</description>
   <prixUnitaire>10.99</prixUnitaire>
</Plant>
```

La présence des balises avec les données permet d'interpréter le texte du document soit pour un programme ou un humain. La clause

```
<!ELEMENT noPlant (#PCDATA)>
```

définit qu'un *noPlant* est une chaîne de caractère. Le symbole #PCDATA indique une feuille de la grammaire dont la syntaxe est une suite quelconque de caractères. XML ne permet pas de définir de types pour les données elle-même. C'est à l'application de vérifier que la chaîne est du bon type (un entier, un réel, etc.). Cette limitation de XML fait d'ailleurs l'objet d'une autre norme, *XML Schema* qui vise en particulier à combler cette lacune. D'autre part, la grammaire en XML *Schema* est elle-même codée en suivant la norme XML.

Contrairement à notre petit exemple, la DTD d'un document n'est habituellement pas incluse directement dans le document lui-même mais est plutôt placée dans un fichier à part auquel le document XML fait référence. Ceci permet de partager la même DTD entre plusieurs documents XML sans avoir à la répéter.

Des outils, appelés parseurs XML, permettent de décoder un fichier XML, de vérifier la validité du contenu par rapport à la grammaire et de stocker le contenu en mémoire du programme sous une forme objet standard, le *Document Object Model* (DOM). Des parseurs Java sont disponibles et suivent eux-mêmes une norme pour l'interface programmatique appelée JAXP (*Java API for XML Processing Specification*). En fournissant la grammaire du document avec son contenu, il devient possible de vérifier la validité du contenu du document, ce qui garantit une meilleure intégrité des documents échangés.

- **Interface programmatique JAXP pour le parsage d'un document XML**

L'interface programmatique JAXP permet de traiter un document XML. L'exemple suivant lit le fichier *Plants.xml*, le parse avec un parseur de JAXP. Deux interfaces sont disponibles pour le traitement d'un

document XML. L'interface DOM (*Document Object Model*) construit une représentation objet du document XML sous forme d'un arbre. Des méthodes permettent de naviguer dans la structure d'arbre DOM afin d'en extraire les éléments. L'interface SAX (*Simple API for XML*) permet d'itérer sur les éléments de manière sérielle selon une approche de programmation par événement analogue à l'approche utilisée dans les interfaces graphiques.

Exemple. Le programme *ExempleJAXPPlants* analyse le fichier *Plants.xml* et en extrait les données avec l'interface DOM. Les données du fichier XML sont extraites par navigation dans l'arbre DOM, puis stockées dans un vecteur d'objets de la classe *Plant* et affichées à l'écran. Le résultat est donc le même que pour *ExempleStreamTokenizer* de la section précédente.

JavaPasAPas/chapitre_8/ExempleJAXPPlants.java

```
/**
 * création d'un arbre DOM avec JAXP Parcours de l'arbre pour extraire les données et les insérer
 * dans le vecteurs d'objets Plant
 */

// Packages de JAXP
import java.io.*;
import java.util.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

public class ExempleJAXPPlants {

  public static void main(String[] args) throws Exception {
    // création d'un DocumentBuilderFactory et configuration des paramètres
    DocumentBuilderFactory unDocBuildFact = DocumentBuilderFactory.newInstance();
    unDocBuildFact.setValidating(true);
    unDocBuildFact.setIgnoringElementContentWhitespace(true);

    // création d'un DocumentBuilder
    DocumentBuilder unDocumentBuilder = unDocBuildFact.newDocumentBuilder();

    // Parsage du document
    File leFile = new File("Plants.xml");
    Document unDocument = unDocumentBuilder.parse(leFile);
    Vector vecteurDePlants = new Vector();
    Node unElementCatalogue =
      unDocument.getDocumentElement(); // Cherche l'élément racine <catalogue>

    // Itérer sur les noeuds <Plant> qui sont les enfants de <Catalogue>
    NodeList listeNodePlants = unElementCatalogue.getChildNodes();
    int tailleListe = listeNodePlants.getLength();
    for (int i = 0; i < tailleListe; i++) {
      Node unNodePlant = listeNodePlants.item(i); // ELEMENT <Plant>

      Node unNodeNoPlant = unNodePlant.getFirstChild(); // ELEMENT <noPlant>
      // la valeur est dans le premier enfant
```

```
    int noPlant = Integer.parseInt(unNodeNoPlant.getFirstChild().getNodeValue());

    Node unNodeDescription = unNodeNoPlant.getNextSibling(); // ELEMENT <description>
    String description = unNodeDescription.getFirstChild().getNodeValue();

    Node unNodePrixUnitaire = unNodeDescription.getNextSibling(); // ELEMENT <prixUnitaire>
    double prixUnitaire = Double.parseDouble(unNodePrixUnitaire.getFirstChild().getNodeValue());

    Plant unPlant = new Plant(noPlant, description, prixUnitaire);
    System.out.println(noPlant + " " + description + " " + prixUnitaire);
    vecteurDePlants.addElement(unPlant);
   }
  }
}
```

Si le fichier n'est pas valide par rapport à la DTD, une exception est levée par le parseur et retournée par le programme. Bien que notre exemple n'en fasse pas la démonstration, il est possible de fournir au besoin une classe chargée de répondre aux exceptions. La structure exacte de l'arbre DOM est assez compliquée et nous ne présentons pas une description exhaustive du sujet.

Une bonne manière de se familiariser avec DOM est d'afficher la structure d'un document en utilisant le programme *DOMEcho*. Le programme montre la structure d'arbre du DOM produit à partir d'un document XML. Le résultat partiel pour *Plants.xml* est le suivant :

```
DOC: nodeName="#document"
  DOC_TYPE: nodeName="Catalogue"
  ELEM: nodeName="Catalogue" local="Catalogue"
    ELEM: nodeName="Plant" local="Plant"
      ELEM: nodeName="noPlant" local="noPlant"
        TEXT: nodeName="#text" nodeValue="10"
      ELEM: nodeName="description" local="description"
        TEXT: nodeName="#text" nodeValue="Cèdre en boule"
      ELEM: nodeName="prixUnitaire" local="prixUnitaire"
        TEXT: nodeName="#text" nodeValue="10.99"
    ELEM: nodeName="Plant" local="Plant"
      ELEM: nodeName="noPlant" local="noPlant"
        TEXT: nodeName="#text" nodeValue="20"
      ELEM: nodeName="description" local="description"
        TEXT: nodeName="#text" nodeValue="Sapin"
      ELEM: nodeName="prixUnitaire" local="prixUnitaire"
        TEXT: nodeName="#text" nodeValue="12.99"
...
```

Chacun des nœuds de l'arbre selon la norme DOM est un objet de la classe **Node**. Chacun des **Node** possède des attributs tel que le type du **Node**, son nom, sa valeur, etc. La racine de l'arbre est un **Node** de type DOC nommé "#document". Dans notre exemple, ce **Node** a deux enfants de type DOC_TYPE et ELEM. Le ELEM nommé "Catalogue"

correspond à l'élément <*Catalogue*> du document XML. Sous lui, on retrouve la liste des éléments <*Plant*>, etc. On peut se servir de cette information afin de déterminer la manière exacte de cheminer dans l'arbre.

Exercice. Reprendre l'exemple précédent pour un fichier xml qui contient les noms et numéros de téléphone de contacts fictifs.

8.4 Gestion de fichiers et répertoires avec java.io.File

La classe java.io.**File** permet de manipuler la hiérarchie des répertoires de fichiers. Elle permet de vérifier si un fichier ou répertoire existe ou non, de créer des fichiers ou des répertoires, de parcourir la hiérarchie, de renommer un fichier ou un répertoire, etc.

- **Vérifier l'existence d'un fichier**

Exemple. L'exemple suivant reprend l'exemple *EcrireEntierEnOctets* d'écriture d'octets dans le fichier *Octets.dat* en vérifiant d'abord si le fichier existe. Si c'est bien le cas, une confirmation est exigée de l'utilisateur avant d'écraser le contenu actuel du fichier.

JavaPasAPas/chapitre_8/VerifierExistenceFichier.java

```
package LivreJava;

import java.io.*;
import javax.swing.JOptionPane;

public class VerifierExistenceFichier {
  public static void main(String args[]) {
    FileOutputStream unFichier;

    try {
      File leFile = new File("Octets.dat");
      if (leFile.exists()) {
        String reponse =
            JOptionPane.showInputDialog("Voulez-vous détruire le contenu existant (oui ou non)?");
        if (reponse.equals("non")) {
          System.out.println("Le fichier demeure tel quel");
          System.exit(0);
        }
      }
      unFichier = new FileOutputStream(leFile);

      int unEntier = 1629696561; // (97*2^24)+(35*2^16)+(50<<2^8)+49 = "a#21" en String;
      // Convertir unEntier en un tableau de 4 octets
      byte[] tampon = new byte[4];
      for (int i = 3; i >= 0; i--) {
        tampon[i] = (byte) (unEntier & 0XFF); // Extrait l'octet le moins significatif
        unEntier >>>= 8; // Décalage de 8 bits (remplissage à 0)
      }
```

```
    unFichier.write(tampon);
    unFichier.close();

   } catch (IOException e) {
     System.err.println("Exception\n" + e.toString());
   }
   System.exit(0);
  }
}
```

Les différences à noter par rapport à l'exemple *EcrireFichierEnOctets* sont les suivantes. L'énoncé

```
File leFile =
new File("Octets.dat");
```

crée un nouvel objet *unFile* de la classe *File* qui correspond à un chemin spécifié en paramètre. La création d'un objet **File** n'ouvre pas le fichier en vue de manipuler son contenu. La classe **File** ne permet que de manipuler les informations du répertoire de fichier, et non pas le contenu des fichiers.

L'énoncé

```
if (leFile.exists()){
```

vérifie si le fichier existe.

L'énoncé

```
unFichier = new FileOutputStream(leFile);
```

ouvre le fichier en écriture à partir de l'objet *leFile* de la classe *File*.

- **Créer de nouveaux dossiers**

La classe **File** peut aussi être utilisée pour créer de nouvelles branches dans la hiérarchie de répertoires.

Exemple. L'exemple suivant crée de nouveaux dossiers pour obtenir un chemin spécifié (ici C:/ DossierA/DossierB/).

```
package LivreJava;
import java.io.*;
public class CreerRepertoire{
    public static void main (String args[]) throws
Exception {
            File unFile = new
File("C:/DossierA/DossierB/");
            unFile.mkdirs();
            if(unFile.exists()){System.out.println("Il
a été créé");}
```

```
                else{System.out.println("Il   n'a   pas   été
créé");}
            }
        }
```

La ligne

```
        File unFile = new File("C:/DossierA/DossierB/");
```

spécifie le chemin désiré.

La méthode **mkdirs**() de la classe *File* est utilisé pour créer les dossiers nécessaires :

```
        unFile.mkdirs();
```

Tous les dossiers manquants du chemin spécifié sont créés.

8.4.1 Dialogue de sélection de fichier avec la classe *JFileChooser*

Souvent, l'utilisateur doit choisir un fichier de manière interactive en naviguant dans un répertoire. L'outil **JFileChooser** est une classe GUI dédiée à ce traitement.

Exemple. L'exemple suivant reprend l'exemple *EcrireEntierEnOctets* d'écriture d'octets dans un fichier. Cependant, ici, un **JFileChooser** permet à l'utilisateur de spécifier le fichier à créer.

JavaPasAPas/chapitre_8/CreerFichierFileChooser.java

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

public class CreerFichierFileChooser extends JFrame {
  public CreerFichierFileChooser() throws Exception {
    JFileChooser unFileChooser = new JFileChooser();
    unFileChooser.setFileSelectionMode(JFileChooser.FILES_ONLY);
    int resultat = unFileChooser.showSaveDialog(this);
    if (resultat != JFileChooser.CANCEL_OPTION) {
      File leFile = unFileChooser.getSelectedFile();
      if (leFile != null && !(leFile.getName().equals(""))) {
        FileOutputStream unFichier = new FileOutputStream(leFile);

        int unEntier = 1629696561; // (97*2^24)+(35*2^16)+(50<<2^8)+49 = "a#21" en String;
        // Convertir unEntier en un tableau de 4 octets
        byte[] tampon = new byte[4];
        for (int i = 3; i >= 0; i--) {
          tampon[i] = (byte) (unEntier & 0XFF); // Extrait l'octet le moins significatif
          unEntier >>>= 8; // Décalage de 8 bits (remplissage à 0)
        }
        unFichier.write(tampon);
        unFichier.close();

      } else {
        System.out.println("Nom de fichier invalide");
      }
    } else {
```

```
    System.out.println("Fichier non choisi");
   }
   System.exit(0);
  }

  public static void main(String args[]) throws Exception {
   new CreerFichierFileChooser();
  }
}
```

La ligne

```
        JFileChooser unFileChooser = new JFileChooser();
```

crée un **JFileChooser**.

La ligne

```
        unFileChooser.setFileSelectionMode(JFileChooser.FIL
ES_ONLY);
```

restreint le choix des éléments qu'il est possible de sélectionner. Il sera par la suite possible de sélectionner uniquement un fichier et pas un répertoire (dossier).

La ligne

```
        int résultat = unFileChooser.showSaveDialog(this);
```

affiche le dialogue et permet à l'utilisateur de sélectionner un fichier. La sélection est confirmée en cliquant le bouton *Save*. Le paramètre *this* est la composante parente. Ici, la classe parente est *CreerFichierFileChooser* qui est un *Jframe*. Le parent doit être un *JComponent*.

L'entier retourné permet de vérifier si le choix a bien été effectué. La ligne suivante

```
        if (résultat != JFileChooser.CANCEL_OPTION){
```

vérifie que le bouton *Cancel* a été cliqué.

La ligne suivante extrait l'objet *leFile* de la classe *File* qui représente le fichier choisi :

```
        File leFile = unFileChooser.getSelectedFile();
```

Ensuite, la ligne suivante vérifie que la sélection n'est pas nulle ou la chaîne vide :

```
      if          (leFile          !=          null          &&
!(leFile.getName().equals(""))){
```

Enfin la ligne suivante crée un *FileOutputStream* à partir du *File*.

```
      FileOutputStream         unFichier         =         new
FileOutputStream(leFile);
```

Exemple. L'exemple suivant reprend l'exemple *LireEntierEnOctets* de lecture d'octets d'un fichier mais en permettant à l'utilisateur de sélectionner le fichier à lire à l'aide d'un dialogue **JFileChooser**.

JavaPasAPas/chapitre_8/LireFichierFileChooser.java

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

public class LireFichierFileChooser extends JFrame {
  public LireFichierFileChooser() throws Exception {
    JFileChooser unFileChooser = new JFileChooser();
    unFileChooser.setFileSelectionMode(JFileChooser.FILES_ONLY);
    int resultat = unFileChooser.showOpenDialog(this);
    if (resultat != JFileChooser.CANCEL_OPTION) {
      File leFile = unFileChooser.getSelectedFile();
      if (leFile != null && !(leFile.getName().equals(""))) {
        FileInputStream unFichier = new FileInputStream(leFile);

        byte[] tampon = new byte[4];
        unFichier.read(tampon); // Lecture des 4 octets

        // Convertir le tableau d'octets tampon en int unEntier
        int unEntier = 0;
        for (int i = 0; i <= 3; i++) {
          unEntier <<= 8;
          unEntier += ((int) tampon[i]) & 0XFF;
        }
        unFichier.close();
        System.out.println("Valeur décimale de l'entier : " + unEntier);

      } else {
        System.out.println("Nom de fichier invalide");
      }
    } else {
      System.out.println("Fichier non choisi");
    }
    System.exit(0);
  }

  public static void main(String args[]) throws Exception {
    new LireFichierFileChooser();
  }
}
```

La différence à noter en comparaison avec l'exemple d'écriture est l'emploi de la méthode *showOpenDialog*() plutôt que *showSaveDialog*() :

```
      int résultat = unFileChooser.showOpenDialog(this);
```

8.5 Fichier d'objets en Java

Dans l'exemple de lecture des données du catalogue de plants dans un vecteur, les données ne sont pas conservées après la fin du programme. Souvent, on voudra stocker ce genre de données en mémoire secondaire pour qu'elles soient conservées à long terme au-delà de l'exécution des programmes qui les manipulent. Cette section montre comment stocker des objets dans un fichier de manière à pouvoir les récupérer sans avoir à les convertir explicitement sous forme d'une série d'octets en employant des classes prévues à cet effet.

8.5.1 Fichier sériel d'objets en Java

Voyons d'abord comment créer un fichier qui contient une suite d'objets en Java avec la classe **ObjectOutputStream**. Ce genre de fichier est souvent appelé fichier sériel ou séquentiel. La méthode **writeObject**(**Object** `obj`) de la classe **ObjectOutputStream** permet d'écrire directement un objet et symétriquement la méthode **readObject**() de la classe **ObjectInputStream** permet de lire cet objet par la suite. Java n'impose pas de structure comme tel au fichier au sens où c'est le programme qui détermine sa structure par les écritures qu'il effectue. Il est ainsi possible d'écrire n'importe quelle suite d'objets provenant possiblement de classes différentes dans le même fichier. Pour retrouver ces objets correctement, ils doivent être lus dans le même ordre. Le programme d'application doit donc s'assurer que les données sont écrites et lues selon une discipline cohérente.

Pour illustrer le concept de fichier sériel, nous allons dans un premier temps créer un fichier qui contient les objets du catalogue de plants sous forme d'une suite d'objets. Comme point de départ, nous supposons que les données se trouvent dans un vecteur d'objets de la classe *Plant* tel que produit par l'exemple de programme vu précédemment *ExempleStreamTokenizer*. La classe **ObjectOutputStream** est utilisée pour écrire directement un objet dans le fichier sans avoir à se préoccuper de la conversion de l'objet en une suite d'octets. La classe **ObjectOutputStream** se sert du mécanisme de *sérialisation* de Java qui permet de convertir un objet sous forme d'une suite d'octets.

Exemple. Le programme *ExempleEcritureObjectOutputStream* lit les données de *Plants.txt* les convertit en un vecteur d'objets et écrits les objets du vecteur les uns à la suite des autres dans le fichier *FluxDePlants.dat*.

Dans le code suivant la méthode *lirePlantsFichierTexte*() reprend essentiellement le code de *ExempleStreamTokenizer*. Cette méthode lit le fichier texte *Plants.txt* et retourne un vecteur d'objets de la classe *Plant*, *vecteurDePlants*. La méthode *écrireFichierFluxPlants* écrit les objets de *vecteurDePlants* les uns à la suite des autres dans le fichier *FluxPlants.dat*.

JavaPasAPas/chapitre_8/EcrireFluxPlants.java

```
/* Illustration de la création d'un fichier d'objets sériel
 * Lit le fichier plants.txt, stocke le contenu dans un vecteur d'objets Plant et
 * crée ensuite le fichier d'objets fluxPlants.dat par accès sériel*/

import java.io.*;
import java.util.*;

public class EcrireFluxPlants {

 // La méthode lit les données de Plants.txt et les retournent dans un vecteur d'objets
 // de la classe Plant
 // Reprend essentiellement le code de ExempleStreamTokenizer
 public static Vector lirePlantsFichierTexte() throws Exception {

  FileReader unFichier = new FileReader("Plants.txt");
  StreamTokenizer unStreamTokenizer = new StreamTokenizer(unFichier);

  // Les 5 lignes suivantes ne sont pas nécessaires car les paramètres
  // donnés sont les valeurs de défaut
  unStreamTokenizer.quoteChar((int) '"');
  unStreamTokenizer.whitespaceChars((int) '\r', (int) '\r');
  unStreamTokenizer.whitespaceChars((int) '\n', (int) '\n');
  unStreamTokenizer.whitespaceChars((int) '\t', (int) '\t');
  unStreamTokenizer.whitespaceChars((int) ' ', (int) ' ');

  Vector vecteurDePlants = new Vector();
  int noPlant = 0;
  String description = "";
  double prixUnitaire = 0.0;

  while (unStreamTokenizer.nextToken() != StreamTokenizer.TT_EOF) { // fin du fichier ?
   // Lecture du noPlant
   if (unStreamTokenizer.ttype == StreamTokenizer.TT_NUMBER) { // Est-ce bien un nombre ?
    noPlant = (int) unStreamTokenizer.nval; // nval est un double !
   } else {
    System.out.println("Le format du fichier est incorrect : noPlant attendu");
    System.exit(1);
   }
   // Lecture de la description
   unStreamTokenizer.nextToken();
   if (unStreamTokenizer.ttype == (int) '"') { // Est-ce bien une chaîne encadr?e par " ?
    description = unStreamTokenizer.sval;
   } else {
    System.out.println("Le format du fichier est incorrect : description attendue");
    System.exit(1);
   }
   // Lecture du prixUnitaire
   unStreamTokenizer.nextToken();
   if (unStreamTokenizer.ttype == StreamTokenizer.TT_NUMBER) { // Est-ce bien un nombre ?
    prixUnitaire = unStreamTokenizer.nval;
   } else {
    System.out.println("Le format du fichier est incorrect : prix attendu");
    System.exit(1);
   }

   // création de l'objet Plant
   Plant unPlant = new Plant(noPlant, description, prixUnitaire);
   System.out.println(noPlant + " " + description + " " + prixUnitaire);
   vecteurDePlants.addElement(unPlant);
  }
  unFichier.close();
  return vecteurDePlants;
 }
```

```
 // La méthode suivante écrit les objets de vecteurDePlants les uns à la suite
 // des autres dans le fichier FluxPlants.dat par accès sériel
 public static void ecrireFichierFluxPlants(Vector vecteurDePlants) throws Exception {
  ObjectOutputStream fichierFluxPlants =
     new ObjectOutputStream(new FileOutputStream("FluxPlants.dat"));
  for (Iterator iterateurPlants = vecteurDePlants.iterator(); iterateurPlants.hasNext(); ) {
   fichierFluxPlants.writeObject(iterateurPlants.next());
  }
  /* pour JDK 1 :
  Enumeration enumerationPlants = vecteurDePlants.elements();
  while (enumerationPlants.hasMoreElements()){
     // le writeObject ajoute le nouvel objet à la fin du fichier
     fichierFluxPlants.writeObject(enumerationPlants.nextElement());
  }*/
  fichierFluxPlants.close();
 }

 public static void main(String args[]) throws Exception {
  Vector vecteurDePlants = lirePlantsFichierTexte();
  ecrireFichierFluxPlants(vecteurDePlants);
 }
}
```

Pour ajouter un objet de la classe *Plant* dans le fichier, il suffit d'appeler **writeObject(Object obj)** qui fait tout le travail de conversion de l'objet sous forme d'une suite d'octets et qui ajoute les octets à la fin du fichier. Pour pouvoir écrire un objet de la classe *Plant* avec la méthode **writeObject(Object obj)**, il faut que *Plant* implémente l'interface *Serializable*. Dans notre exemple, il n'y a qu'à s'assurer de la présence de la clause *implements Serializable* tel qu'illustré par le code suivant :

```
package LivreJava;
import java.io.Serializable;
public class Plant implements Serializable{
    private int noPlant; //numéro de catalogue du plant
    private String description; //description du plant
    private double prixUnitaire; //prix unitaire du plant

    public Plant(int noPlant, String description, double prixUnitaire) {
        this.noPlant = noPlant;
        this.description = description;
        this.prixUnitaire = prixUnitaire;
    }
    public void setNoPlant(int noPlant){this.noPlant = noPlant;}
    public int getNoPlant(){ return noPlant; }
    public  void  setDescription(String  description){this.description  =
description;}
    public String getDescription(){ return description; }
    public void setPrixUnitaire(double prixUnitaire){this.prixUnitaire =
prixUnitaire;}
    public double getPrixUnitaire(){ return prixUnitaire; }
}
```

Sérialisation (*serialization*) Java

Le mécanisme de *sérialisation* Java effectue la conversion d'un objet en une suite d'octets. Le mécanisme inverse est appelé *désérialisation*. Le mécanisme de sérialisation Java est très puissant et il permet même de sérialiser un objet complexe qui fait référence à d'autres objets en incluant les objets référencés dans la sérialisation. On pourrait d'ailleurs écrire tout le vecteur d'objets *Plant* d'un coup en une seule écriture.

Le mécanisme de sérialisation n'est pas le plus économique d'un point de vue de l'occupation d'espace. La suite d'octets produite contient non seulement les données, c'est-à-dire les valeurs des attributs mais aussi des *métadonnées* qui décrivent l'objet, c'est-à-dire le nom de la classe incluant son package, le nom et le type des attributs, etc. Par exemple, un objet sérialisé de la classe *Plant* occupe environ 120 octets. En écrivant uniquement les données, comme nous le verrons à la section suivante, une cinquantaine d'octets par objet serait suffisant. Les métadonnées ne sont pas nécessaires lorsque le programme connaît le contenu du fichier à priori. Cependant, l'ajout de ces informations assure un plus grand niveau de fiabilité car une exception est levée dans le cas où l'objet lu n'est pas de la classe attendue.

Allocation automatique d'espace au fichier

Lors de l'écriture d'un objet, il est ajouté à la fin du fichier et l'espace nécessaire est automatiquement alloué au besoin. Le programme n'a donc pas à se préoccuper de l'allocation d'espace. Évidemment, ce manque du contrôle au niveau de la stratégie d'allocation d'espace peut conduire à une fragmentation importante du fichier.

Exemple. Le programme *LireFluxPlants* lit les objets un par un en partant du premier jusqu'à ce que la fin du fichier soit atteinte.

JavaPasAPas/chapitre_8/LireFluxPlants.java

```
/* Illustration de la lecture d'un fichier d'objets par itération sérielle
 * Lit le fichier fluxPlants.dat et en affiche le contenu */

import java.io.*;

public class LireFluxPlants {

  public static void main(String args[]) throws Exception {

    ObjectInputStream fichierFluxPlants =
```

```
    new ObjectInputStream(new FileInputStream("FluxPlants.dat"));

  while (true) {
   Plant unPlant = new Plant(0, "", 0.0);
   try { // Lecture de l'objet suivant
    unPlant = (Plant) fichierFluxPlants.readObject();
   } catch (EOFException e) {
    break;
   }
   System.out.println(
     unPlant.getNoPlant() + " " + unPlant.getDescription() + " " + unPlant.getPrixUnitaire());
  }
  fichierFluxPlants.close();
 }
}
```

L'appel à *readObject*() retourne le prochain objet lu relativement à la position courante dans le fichier par le mécanisme de désérialisation. Lors de l'ouverture du fichier, la position courante est le début du fichier. Une exception est levée lorsque la fin du fichier est atteinte.

Cette manière d'organiser un fichier est très restrictive. En particulier, supposons que l'on veuille simplement modifier une donnée, par exemple, le prix d'un plant. La seule façon de procéder consiste à lire tous les objets et à les réécrire dans un nouveau fichier en modifiant le prix au passage. Ceci serait très inefficace en particulier dans le cas d'un gros volume de données.

S'il est important de pouvoir accéder sélectivement à un ou quelques enregistrements d'un fichier, une organisation à accès direct peut être utilisé. La classe *RandomAccessFile* offre cette possibilité.

- Écriture d'un objet complexe

Tel que mentionné précédemment, il est possible de sérialiser un objet complexe qui fait référence à d'autres objets. Dans l'exemple précédent l'objet *vecteurDePlants* est un exemple d'objet complexe. C'est un vecteur qui contient des références aux objets de la classe *Plant*. Plutôt que d'écrire les objets du vecteur un à un dans le fichier, il est possible d'écrire tout le vecteur d'un coup.

Exemple. Le programme suivant écrit l'objet complexe *vecteurDePlants* dans le fichier *VecteurPlants.dat*.

JavaPasAPas/chapitre_8/EcrireVecteurPlants.java

```
/* Illustration de l'écriture d'un objet complexe dans un fichier par sérialisation
 * Lit le fichier plants.txt, stocke le contenu dans un vecteur d'objets Plant et
 * écrit ensuite le vecteur dans le fichier VecteurPlants.dat*/

import java.io.*;
import java.util.*;

public class EcrireVecteurPlants {

  // La méthode lit les données de Plants.txt et les retournent dans un vecteur d'objets
  // de la classe Plant
  // Reprend essentiellement le code de ExempleStreamTokenizer
  public static Vector lirePlantsFichierTexte() throws Exception {

    FileReader unFichier = new FileReader("Plants.txt");
    StreamTokenizer unStreamTokenizer = new StreamTokenizer(unFichier);

    // Les 5 lignes suivantes ne sont pas nécessaires car les paramètres
    // donnés sont les valeurs de défaut
    unStreamTokenizer.quoteChar((int) '"');
    unStreamTokenizer.whitespaceChars((int) '\r', (int) '\r');
    unStreamTokenizer.whitespaceChars((int) '\n', (int) '\n');
    unStreamTokenizer.whitespaceChars((int) '\t', (int) '\t');
    unStreamTokenizer.whitespaceChars((int) ' ', (int) ' ');

    Vector vecteurDePlants = new Vector();
    int noPlant = 0;
    String description = "";
    double prixUnitaire = 0.0;

    while (unStreamTokenizer.nextToken() != StreamTokenizer.TT_EOF) { // fin du fichier ?
      // Lecture du noPlant
      if (unStreamTokenizer.ttype == StreamTokenizer.TT_NUMBER) { // Est-ce bien un nombre ?
        noPlant = (int) unStreamTokenizer.nval; // nval est un double !
      } else {
        System.out.println("Le format du fichier est incorrect : noPlant attendu");
        System.exit(1);
      }
      // Lecture de la description
      unStreamTokenizer.nextToken();
      if (unStreamTokenizer.ttype == (int) '"') { // Est-ce bien une chaîne encadrée par " ?
        description = unStreamTokenizer.sval;
      } else {
        System.out.println("Le format du fichier est incorrect : description attendue");
        System.exit(1);
      }
      // Lecture du prixUnitaire
      unStreamTokenizer.nextToken();
      if (unStreamTokenizer.ttype == StreamTokenizer.TT_NUMBER) { // Est-ce bien un nombre ?
        prixUnitaire = unStreamTokenizer.nval;
      } else {
        System.out.println("Le format du fichier est incorrect : prix attendu");
        System.exit(1);
      }

      // création de l'objet Plant
      Plant unPlant = new Plant(noPlant, description, prixUnitaire);
      System.out.println(noPlant + " " + description + " " + prixUnitaire);
      vecteurDePlants.addElement(unPlant);
    }
    unFichier.close();
    return vecteurDePlants;
  }
```

```
 // La méthode suivante écrit les objets de vecteurDePlants les uns à la suite
 // des autres dans le fichier FluxPlants.dat par accès sériel
 public static void ecrireFichierFluxPlants(Vector vecteurDePlants) throws Exception {
  ObjectOutputStream fichierFluxPlants =
     new ObjectOutputStream(new FileOutputStream("VecteurPlants.dat"));
  fichierFluxPlants.writeObject(vecteurDePlants);
  fichierFluxPlants.close();
 }

 public static void main(String args[]) throws Exception {
  Vector vecteurDePlants = lirePlantsFichierTexte();
  ecrireFichierFluxPlants(vecteurDePlants);
 }
}
```

L'énoncé

```
fichierFluxPlants.writeObject(vecteurDePlants);
```

écrit tout le vecteur d'un coup. Comme le vecteur contient des objets de la classe *Plant*, ces objets sont sérialisés automatiquement.

Inversement, on peut lire tout le vecteur d'un coup.

Exemple. Le programme suivant lit l'objet complexe *Vector* contenant des objets de la classe *Plant* par désérialisation.

JavaPasAPas/chapitre_8/LireVecteurPlants.java

```
/* Illustration de la lecture d'un objet complexe par dé-sérialisation
 * Lit un vecteur de plants du VecteurPlants.dat et en affiche le contenu */

import java.io.*;
import java.util.*;

public class LireVecteurPlants {

 public static void main(String args[]) throws Exception {

  ObjectInputStream fichierFluxPlants =
     new ObjectInputStream(new FileInputStream("VecteurPlants.dat"));
  Vector vecteurDePlants = (Vector) fichierFluxPlants.readObject();
  fichierFluxPlants.close();

  Enumeration enumerationPlants = vecteurDePlants.elements();
  while (enumerationPlants.hasMoreElements()) {
   Plant unPlant = (Plant) enumerationPlants.nextElement();
   System.out.println(
      unPlant.getNoPlant() + " " + unPlant.getDescription() + " " + unPlant.getPrixUnitaire());
  }
 }
}
```

L'instruction suivante lit le vecteur dans une seule invocation *readObjet*() :

```
Vector vecteurDePlants = (Vector)
fichierFluxPlants.readObject();
```

La sérialisation du vecteur de plants va occuper plus d'espace que la séquence des objets *Plant* sérialisés. En effet, la structure de données utilisées pour le vecteur lui-même doit aussi être sérialisée.

8.5.2 Fichier à adressage relatif en Java avec RandomAccessFile

Cette section présente un exemple d'implémentation d'un fichier à adressage relatif en Java en passant par la classe **RandomAccessFile**. Ceci permet de manipuler les données d'une manière plus performante en accédant directement aux données sans devoir passer par le traitement de tous les objets du fichier comme c'est le cas du fichier sériel. Comme pour le cas précédent, les données sont disposées dans le fichier les unes à la suite des autres. Les données d'un objet sont représentées par un ensemble d'octets appelé un enregistrement (*record*). Les données sont écrites et lues sans passer par la sérialisation. Il est possible d'accéder directement aux données d'un objet particulier soit pour le lire ou soit pour l'écrire. A cet effet, un numéro d'enregistrement relatif (NER) est assigné à chacun des enregistrements du fichier en fonction de sa position.

Un nouvel enregistrement est toujours ajouté à la fin du fichier. Pour simplifier, la suppression d'un enregistrement n'est pas permise. Tel qu'illustré à la figure suivante, le nombre courant d'enregistrements est stocké sous forme d'un *int* dans les quatre premiers octets du fichier. Lors de la création d'un nouvel enregistrement, cette information sert à déterminer le NER du nouvel enregistrement.

Dans notre figure, nous supposons que la taille d'un enregistrement est fixée à 50. L'emploi d'une taille fixe permet de retrouver facilement la position d'un enregistrement dans le fichier. Comme les octets sont numérotés à partir de 0, la position du premier octet d'un enregistrement d'adresse NER est :

$$\textit{Position du premier octet} = \text{NER} * 50 + 4$$

Dans notre implémentation, une méthode statique *tailleMaxEnregistrement()* de la classe *Plant* retourne la taille maximale d'un enregistrement. De fait, tout l'espace est réservé pour l'enregistrement même si en réalité, sa taille peut être inférieure. Dans ce cas, les derniers octets sont inutilisés mais tout de même alloués.

La méthode *ecrireEnregistrementTailleMax* est ajoutée à *Plant* afin d'écrire la suite d'octets de l'objet *Plant* dans le fichier. Contrairement à l'implémentation de fichier à accès sériel de la section précédente, la sérialisation n'est pas utilisée. Ceci permet d'avoir un contrôle précis sur les octets écrits et de réduire l'espace occupé par l'enregistrement correspondant à un objet parce qu'il n'y a pas de métadonnées stockées avec les données.

Exemple. Classe *Plant* modifiée.

JavaPasAPas/chapitre_8/Plant.java

```
import java.io.*;

public class Plant implements Serializable {
  private int noPlant; // numéro de catalogue du plant
  private String description; // description du plant
  private double prixUnitaire; // prix unitaire du plant

  public Plant(int noPlant, String description, double prixUnitaire) {
    this.noPlant = noPlant;
    this.description = description;
    this.prixUnitaire = prixUnitaire;
  }

  public void setNoPlant(int noPlant) {
    this.noPlant = noPlant;
  }

  public int getNoPlant() {
    return noPlant;
  }

  public void setDescription(String description) {
    this.description = description;
  }

  public String getDescription() {
    return description;
  }

  public void setPrixUnitaire(double prixUnitaire) {
    this.prixUnitaire = prixUnitaire;
  }

  public double getPrixUnitaire() {
    return prixUnitaire;
  }

  public void ecrireEnregistrementTailleMax(RandomAccessFile unFichier) throws Exception {
    unFichier.writeInt(noPlant); // 4 octets
    if (description.length() > 38) {
      System.exit(1);
    }
    unFichier.writeInt(description.length()); // 4octets
    unFichier.writeBytes(description); // max 38 octets
    unFichier.writeDouble(prixUnitaire); // 8 octets
  }

  public void lireEnregistrementTailleMax(RandomAccessFile unFichier) throws Exception {
    noPlant = unFichier.readInt();
```

```
    int tailleDescription = unFichier.readInt();
    byte[] tampon = new byte[tailleDescription];
    unFichier.readFully(tampon);
    description = new String(tampon);
    prixUnitaire = unFichier.readDouble();
  }

  public static int tailleMaxEnregistrement() {
    return 50;
  }
}
```

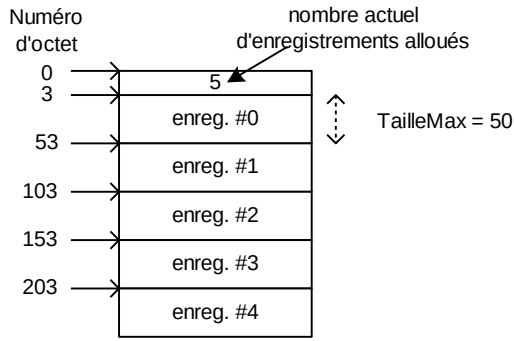


Figure 39. Organisation du fichier à adressage relatif.

Pour simplifier l'exemple, les seules opérations permises sont :

- Créer un nouvel enregistrement.
- Sélectionner un enregistrement à partir du NER.
- Modifier le prix d'un enregistrement sélectionné à partir du NER.

Exemple. Le programme suivant illustre l'accès direct et l'adressage relatif.

JavaPasAPas/chapitre_8/AccesDirect.java

```
/* Illustration de l'accès direct avec un fichier à adressage relatif
 * Opérations permises :
 *     sélectionner un enregistrement par son NER
 *     modifier le prix d'un enregistrement sélectionné par son NER
 *     créer un nouvel enregistrement (toujours à la fin)
 *     (ne permet pas la suppression)
 */

import java.io.*;
import javax.swing.JOptionPane;
```

```
public class AccesDirect {

 public static void main(String args[]) throws Exception {
  // Ouverture du fichier ou creation si n'existe pas
  int nombreAlloue; // nombre d'enregistrements actuellement alloués
  RandomAccessFile fichierDirectPlants;
  File leFichier =
     new File(
  "paramètres/Users/Robert/Documents/NetBeansProjects/JavaLivre/build/classes/DirectPlants.dat");
  if (leFichier.exists()) { // Fichier existe ?
   fichierDirectPlants = new RandomAccessFile(leFichier, "rw");
   // Cherche le nombre d'enregistrements actuellement alloués
   nombreAlloue = fichierDirectPlants.readInt();
  } else { // Le fichier n'existe pas, il faut initialiser nombreAlloue
   fichierDirectPlants = new RandomAccessFile(leFichier, "rw");
   // Initialiser nombreAlloue
   nombreAlloue = 0;
   fichierDirectPlants.writeInt(nombreAlloue);
  }

  String chaineNER;
  int numeroER;
  Plant unPlant = new Plant(0, "", 0.0);

  while (true) {
   String chaineChoix =
      JOptionPane.showInputDialog("Menu: 1(lire); 2(modifier prix); 3(ajouter) ; 0 (terminer)");
   int choix = Integer.parseInt(chaineChoix);

   switch (choix) {
    case 1: // Lire et afficher l'enregistrement
     chaineNER = JOptionPane.showInputDialog("Entrez le numéro d'enregistrement relatif :");
     numeroER = Integer.parseInt(chaineNER);
     if (numeroER >= 0 && numeroER < nombreAlloue) {
      // sélectionner un enregistrement par son NER
      fichierDirectPlants.seek(numeroER * Plant.tailleMaxEnregistrement() + 4);
      unPlant.lireEnregistrementTailleMax(fichierDirectPlants);

      JOptionPane.showMessageDialog(
         null,
         "NER :"
           + numeroER
           + "\nnoPlant :"
           + unPlant.getNoPlant()
           + "\ndescription :"
           + unPlant.getDescription()
           + "\nprixUnitaire :"
           + unPlant.getPrixUnitaire());

     } else {
      JOptionPane.showMessageDialog(null, "Numéro incorrect :" + numeroER);
     }
     break;

    case 2: // Modifier un enregistrement
     chaineNER = JOptionPane.showInputDialog("Entrez le numéro d'enregistrement relatif :");
     numeroER = Integer.parseInt(chaineNER);
     if (numeroER >= 0 && numeroER < nombreAlloue) {
      // D'abord sélectionner l'enregistrement par son NER
      fichierDirectPlants.seek(numeroER * Plant.tailleMaxEnregistrement() + 4);
      unPlant.lireEnregistrementTailleMax(fichierDirectPlants);

      // Modifier son prix en mémoire centrale
      String chainePrix = JOptionPane.showInputDialog("Entrez le nouveau prix :");
      unPlant.setPrixUnitaire(Double.parseDouble(chainePrix));
```

```
        // Ecrire l'enregistrement modifié
        fichierDirectPlants.seek(numeroER * Plant.tailleMaxEnregistrement() + 4);
        unPlant.ecrireEnregistrementTailleMax(fichierDirectPlants);

        JOptionPane.showMessageDialog(
            null,
            "NER :"
              + numeroER
              + "\nnoPlant :"
              + unPlant.getNoPlant()
              + "\ndescription :"
              + unPlant.getDescription()
              + "\nprixUnitaire :"
              + unPlant.getPrixUnitaire());
      } else {
        JOptionPane.showMessageDialog(null, "Numéro incorrect :" + numeroER);
      }
      break;

    case 3: // créer un enregistrement
      String chaineNoPlant = JOptionPane.showInputDialog("Entrez le noPlant :");
      unPlant.setNoPlant(Integer.parseInt(chaineNoPlant));
      unPlant.setDescription(JOptionPane.showInputDialog("Entrez la description :"));
      String chainePrix = JOptionPane.showInputDialog("Entrez le prixUnitaire :");
      unPlant.setPrixUnitaire(Double.parseDouble(chainePrix));

      // Allocation sérielle : le NER du nouvel enregistrement = nombreAlloue
      fichierDirectPlants.seek(nombreAlloue * Plant.tailleMaxEnregistrement() + 4);
      unPlant.ecrireEnregistrementTailleMax(fichierDirectPlants);

      JOptionPane.showMessageDialog(
          null,
          "NER :"
            + nombreAlloue
            + "\nnoPlant :"
            + unPlant.getNoPlant()
            + "\ndescription :"
            + unPlant.getDescription()
            + "\nprixUnitaire :"
            + unPlant.getPrixUnitaire());

      // Incrémenter le nombre d'enregistrements alloués
      nombreAlloue++;
      fichierDirectPlants.seek(0);
      fichierDirectPlants.writeInt(nombreAlloue);
      break;

    case 0: // Terminer
      fichierDirectPlants.close();
      System.exit(0);
      break;

    default:
      JOptionPane.showMessageDialog(null, "Choix incorrect :" + choix);
    }
  }
 }
}
```

La méthode *seek*() de **RandomAccessFile** fixe la position courante dans le fichier à un numéro d'octet particulier. Ainsi, l'instruction

```
fichierDirectPlants.seek(numeroER*Plant.tailleMaxEnregistrement()+4);
```
établit la position courante au début de l'enregistrement *numeroER*. Ensuite l'instruction

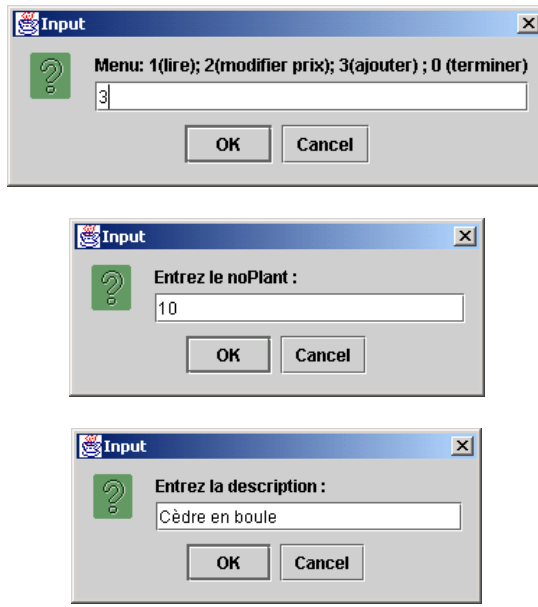
```
unPlant.lireEnregistrementTailleMax(fichierDirectPlants);
```
lit l'enregistrement à cette position.

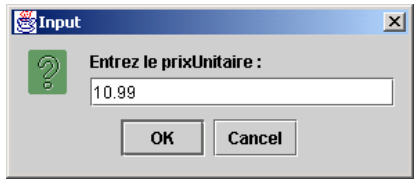
L'instruction

```
unPlant.ecrireEnregistrementTailleMax(fichierDirectPlants);
```
écrit un enregistrement à cette position. Ceci remplace effectivement le contenu précédent par le contenu de l'objet *Plant*.

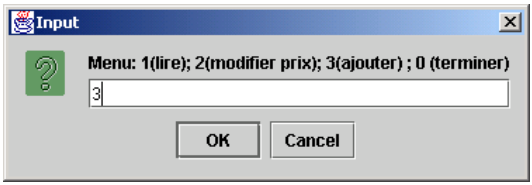
L'espace est alloué automatiquement lors de l'écriture si nécessaire. Par contre, une tentative de lecture au-delà du dernier enregistrement écrit provoque une exception.

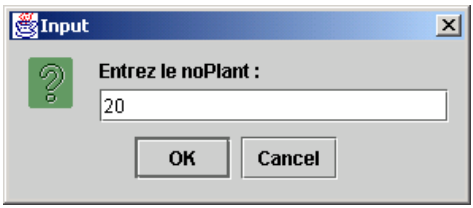
Exemple. Le scénario suivant illustre le principe de l'exécution du programme :

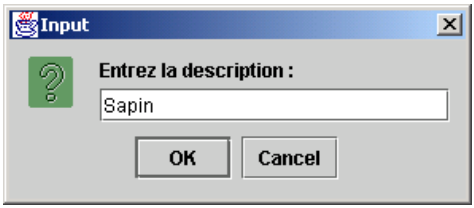


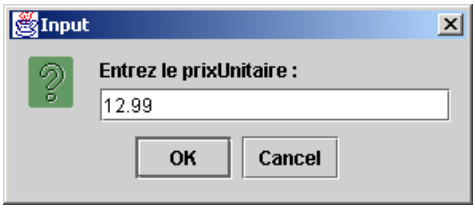

Input
Entrez le prixUnitaire :
10.99
OK
Cancel


Message
NER :0
noPlant :10
description :Cèdre en boule
prixUnitaire :10.99
OK

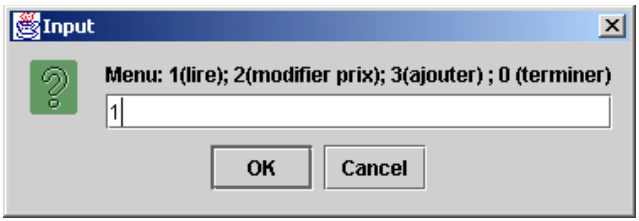

Input
Menu: 1(lire); 2(modifier prix); 3(ajouter) ; 0 (terminer)
3
OK
Cancel

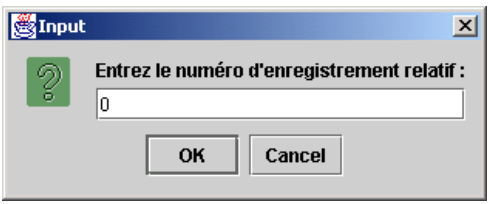

Input
Entrez le noPlant :
20
OK
Cancel


Input
Entrez la description :
Sapin
OK
Cancel

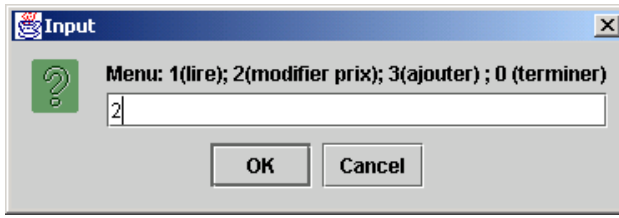

Input
Entrez le prixUnitaire :
12.99
OK
Cancel

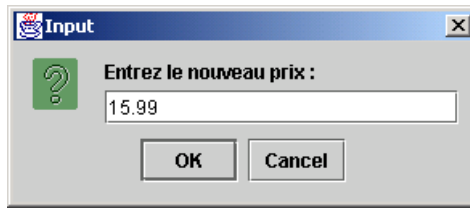

Message
NER :1
noPlant :20
description :Sapin
prixUnitaire :12.99
OK


Input
Menu: 1(lire); 2(modifier prix); 3(ajouter) ; 0 (terminer)
1
OK
Cancel


Input
Entrez le numéro d'enregistrement relatif :
0
OK
Cancel


Message
NER :0
noPlant :10
description :Cèdre en boule
prixUnitaire :10.99
OK





Exercice. Charger le fichier à adressage relatif *DirectPlants.dat* avec les données du catalogue de plants saisies du fichier *Plants.txt*.

JavaPasAPas/chapitre_8/CreerFichierDirect.java

```
/* Illustration de la création d'un fichier d'objets sériel
 * Lit le fichier plants.txt, stocke le contenu dans un vecteur d'objets Plant et
 * crée ensuite le fichier d'objets fluxPlants.dat par accès sériel*/

import java.io.*;
import java.util.*;

public class CreerFichierDirect {

 // La méthode lit les données de Plants.txt et les retournent dans un vecteur d'objets
 // de la classe Plant
 // Reprend essentiellement le code de ExempleStreamTokenizer
```

```
public static Vector lirePlantsFichierTexte() throws Exception {

 FileReader unFichier = new FileReader("Plants.txt");
 StreamTokenizer unStreamTokenizer = new StreamTokenizer(unFichier);

 // Les 5 lignes suivantes ne sont pas nécessaires car les paramètres
 // donnés sont les valeurs de défaut
 unStreamTokenizer.quoteChar((int) '"');
 unStreamTokenizer.whitespaceChars((int) '\r', (int) '\r');
 unStreamTokenizer.whitespaceChars((int) '\n', (int) '\n');
 unStreamTokenizer.whitespaceChars((int) '\t', (int) '\t');
 unStreamTokenizer.whitespaceChars((int) ' ', (int) ' ');

 Vector vecteurDePlants = new Vector();
 int noPlant = 0;
 String description = "";
 double prixUnitaire = 0.0;

 while (unStreamTokenizer.nextToken() != StreamTokenizer.TT_EOF) { // fin du fichier ?
  // Lecture du noPlant
  if (unStreamTokenizer.ttype == StreamTokenizer.TT_NUMBER) { // Est-ce bien un nombre ?
   noPlant = (int) unStreamTokenizer.nval; // nval est un double !
  } else {
   System.out.println("Le format du fichier est incorrect : noPlant attendu");
   System.exit(1);
  }

  // Lecture de la description
  unStreamTokenizer.nextToken();
  if (unStreamTokenizer.ttype == (int) '"') { // Est-ce bien une chaîne encadr?e par " ?
   description = unStreamTokenizer.sval;
  } else {
   System.out.println("Le format du fichier est incorrect : description attendue");
   System.exit(1);
  }
  // Lecture du prixUnitaire
  unStreamTokenizer.nextToken();
  if (unStreamTokenizer.ttype == StreamTokenizer.TT_NUMBER) { // Est-ce bien un nombre ?
   prixUnitaire = unStreamTokenizer.nval;
  } else {
   System.out.println("Le format du fichier est incorrect : prix attendu");
   System.exit(1);
  }

  // création de l'objet Plant
  Plant unPlant = new Plant(noPlant, description, prixUnitaire);
  System.out.println(noPlant + " " + description + " " + prixUnitaire);
  vecteurDePlants.addElement(unPlant);
 }
 unFichier.close();
 return vecteurDePlants;
}

// La méthode suivante écrit les objets de vecteurDePlants les uns à la suite
// des autres dans le fichier FluxPlants.dat par accès sériel
public static void ecrireFichierFluxPlants(Vector vecteurDePlants) throws Exception {
 RandomAccessFile fichierDirectPlants = new RandomAccessFile("DirectPlants.dat", "rw");

 Enumeration enumerationPlants = vecteurDePlants.elements();
 int numeroEnregistrementRelatif = 0;
 while (enumerationPlants.hasMoreElements()) {
  // créer nouvel enregitrement
  // Le seek établit la position courante
  // NB Les quatre premiers octets du fichier contiennent le nombre d'enregistrements créés
  fichierDirectPlants.seek(numeroEnregistrementRelatif * Plant.tailleMaxEnregistrement() + 4);
  ((Plant) enumerationPlants.nextElement()).ecrireEnregistrementTailleMax(fichierDirectPlants);
```

```
    numeroEnregistrementRelatif++;
   }

   // Stocke le nombre d'enregistrements dans les octets 0 à 3
   fichierDirectPlants.seek(0);
   fichierDirectPlants.writeInt(numeroEnregistrementRelatif);
  }

  public static void main(String args[]) throws Exception {
   Vector vecteurDePlants = lirePlantsFichierTexte();
   ecrireFichierFluxPlants(vecteurDePlants);
  }
}
```

Exercice. Etendre le programme précédent en permettant de modifier le numéro de plant et sa description en plus du prix. Ajouter la possibilité de sélectionner un *Plant* par son numéro.

Exercice. Dans l'exercice précédent, cherchez une solution pour sélectionner un enregistrement par son numéro qui permet d'éviter de lire tout le fichier dans le pire cas.

Exercice. Ajoutez la possibilité de supprimer un *Plant*.