INTRODUCTION A JAVA 4

1. INTRODUCTION A LA PROGRAMMATION OBJET 4

1.1 LIMITES DES APPROCHES CLASSIQUES 4
1.2 MODELE OBJET 5
1.3 PRINCIPES DE LA PROGRAMMATION ORIENTEE OBJET 6

2. JAVA 7

2.1 HISTOIRE 7
2.2 CARACTERISTIQUES 8

CONCEPTS NON OBJET 12

1. INTRODUCTION 12

1.1 PREMIER PROGRAMME INTRODUCTIF 12
1.2 COMMENTAIRES 13

2. DONNEES ET VARIABLES 13

2.1 TYPES PRIMITIFS 14
2.2 CONSTANTES LITTERALES 14
2.3 DECLARATIONS ET INITIALISATIONS 15
2.4 PORTEE LOCALE 16
2.5 POSITION DE LA DECLARATION D'UNE VARIABLE LOCALE 17
2.6 INITIALISATION DIFFEREE 17

3. EXPRESSIONS ET OPERATEURS 18

3.1 INTRODUCTION 18
3.2 OPERATEURS LOGIQUES ET BOOLEENS 19
3.3 CONVERSION ET PROMOTION 20
3.4 SYNTHESE DES OPERATEURS 22

4. TABLEAUX 23

4.1 INTRODUCTION 23
4.2 REFERENCE 23
4.3 OPERATEUR NEW 25
4.4 GARBAGE COLLECTOR 27
4.5 DONNEE LENGTH 27
4.6 TABLEAU A PLUSIEURS DIMENSIONS 28

5. INSTRUCTIONS 30

5.1 INTRODUCTION 30
5.2 BOUCLES 30
5.3 INSTRUCTIONS DE SELECTION 32
5.4 INSTRUCTIONS DE RUPTURE 34

6. METHODES 37

6.1 INTRODUCTION 37
6.2 PASSATION DE PARAMETRES 39
6.3 RECURSIVITE 42
6.4 SURCHARGE DE METHODE 44
6.5 METHODE A NOMBRE D'ARGUMENTS VARIABLE 48
6.6 METHODE MAIN 49

7. EXERCICES DE SYNTHESE 50

7.1 CALCUL PARTIEL D'UNE SERIE 50
7.2 SUITE DE SYRACUSE 51
7.3 SUITE DE NEWTON 51
7.4 AFFICHAGE EN BASE DEUX 52
7.5 CALCULATRICE SUR LA LIGNE DE COMMANDE 53
7.6 COPIE, COMPARAISON ET CONCATENATION DE TABLEAUX 54
7.7 CALCUL MATRICIEL 56

CLASSES ET OBJETS 58

1. INTRODUCTION 58

1.1 EXEMPLE INTRODUCTIF 58
1.2 PREMIERE CLASSE ET PREMIERS OBJETS 59

2. BASES SUR LES CLASSES ET OBJETS 61

2.1 DEFINITION D'UNE CLASSE ET DROITS D'ACCES 61
2.2 CREATION ET CYCLE DE VIE D'UN OBJET 62
2.3 OBJETS ARGUMENTS DE METHODES 65
2.4 REGLES DE PORTEES DES CHAMPS ET VARIABLES LOCALES 68
2.5 SURCHARGE ET DROIT D'ACCES 69

3. CONSTRUCTEURS ET INITIALISATION DES OBJETS 70

3.1 ROLE ET DEFINITION 70
3.2 SURCHARGE ET CONSTRUCTEUR PAR DEFAUT 73
3.3 MOT CLE THIS ET CONSTRUCTEURS 75
3.4 BLOCS D'INITIALISATION 80

4. TABLEAU D'OBJETS 81

5. MEMBRES STATIQUES 83

5.1 CHAMPS STATIQUES 83
5.2 METHODES STATIQUES 85
5.3 BLOC D'INITIALISATION STATIQUE 87
5.4 CHAMPS CONSTANTS STATIQUES OU NON STATIQUES 90

6. ECRITURE STANDARD D'UNE CLASSE 92

6.1 TYPOLOGIE DES METHODES 92
6.2 EXEMPLE DE LA CLASSE POINT 94

7. OBJET, CHAMPS MUTABLES ET DISSIMULATION 97

8. CLASSES INTERNES 101

8.1 CLASSE INTERNE NON STATIQUE 102
8.2 CLASSE INTERNE STATIQUE 105
8.3 CLASSE INTERNE LOCALE 108

9. ENUMERATION 110

10. PAQUETAGES 115

11. CLASSES DE BASE DE L'API JAVA 118

11.1 CHAINES DE CARACTERES 118
11.1.1 String (chaînes immuables) 119
11.1.2 StringBuffer (chaînes mutables) 122
11.1.3 Méthode toString 125
11.2 WRAPPER 126
11.2.1 Généralités 126
11.2.2 Integer et traitement de bits 128
11.2.3 Double et infinis 130
11.2.4 Character et types de caractères 131
11.2.5 Autoboxing 134
11.2.6 Synthèse sur les possibilités de conversion 134
11.3 MATH 136
11.3.1 Fonctions mathématiques 136
11.3.2 Comparaisons 137
11.3.3 Arrondis 138
11.3.4 Génération de nombres aléatoires 139
11.4 ARRAYS 139

12. EXERCICES DE SYNTHESE 142

12.1 NOMBRES COMPLEXES 142
12.2 COMPTE EN BANQUE 144
12.3 CERCLE ET CLASSE INTERNE 148

12.4 LISTE CHAINEE....150
12.5 CARTES....155
HERITAGE ET POLYMORPHISME....158
1. INTRODUCTION....158
1.1 DEFINITIONS....158
1.2 EXEMPLE INTRODUCTIF....158
2. BASES SUR L'HERITAGE....159
2.1 DEFINITION D'UNE CLASSE DERIVEE ET DROITS D'ACCES....159
2.2 SUITE DE L'EXEMPLE INTRODUCTIF....160
3. CONSTRUCTEURS....163
3.1 LE MOT CLE SUPER....163
3.2 CYCLE DE VIE DES OBJETS ET ORDRE D'APPEL DES CONSTRUCTEURS....164
3.3 CONSTRUCTEUR PAR DEFAUT....166
4. REDEFINITION ET SURDEFINITION....167
4.1 DEFINITION ET PRINCIPES....167
4.2 CONTRAINTES....170
4.3 MOT CLE SUPER....173
4.4 MASQUAGE DES CHAMPS....174
5. MEMBRES STATIQUES....175
5.1 PARTICULARITES....175
5.2 ORDRE DE CREATION....177
6. CLASSE OBJET....180
6.1 HERITAGE IMPLICITE ET SUPER CLASSE....180
6.2 COPIE PROFONDE....182
7. POLYMORPHISME....184
7.1 COMPATIBILITE....184
7.1.1 Cas général....184
7.1.2 Cas des tableaux....188
7.2 POLYMORPHISME UNIVERSEL D'INCLUSION....189
7.3 CONTROLE STATIQUE....191
7.4 SURCHARGE ET HIERARCHIE DE CLASSES....192
7.5 LIMITES DU POLYMORPHISME EN JAVA....194
7.5.1 Types de paramètres....194
7.5.2 Méthode statique....196
7.6 OPERATEUR INSTANCEOF....197
7.7 CLASSES ET METHODES FINAL....200
8. CLASSES ABSTRAITES ET INTERFACES....201
8.1 CLASSES ABSTRAITES....201
8.2 INTERFACE....203
8.2.1 Principes et définitions....203
8.2.2 Exemple....205
8.2.3 Interface Comparable et Arrays....207
9. CLASSES INTERNES ANONYMES....208
10. EXERCICES DE SYNTHESES....210
10.1 HIERARCHIE DE CLASSES DES CETACES....210
10.2 CLASSES POUR REPRESENTER DES SALARIES....213
10.3 STOCK....216
ANNEXES....220
1. BIBLIOGRAPHIE....220
1.1 QUELQUES OUVRAGES GENERAUX....220
1.2 QUELQUES SITES INTERNET DE REFERENCE....220

Les objets se transfigurent selon le magnétisme des personnes qui les approchent, toutes choses n'ayant d'autre signification, pour chacun, que celle que chacun peut leur prêter.

Auguste Villiers de l'Isle-Adam

Introduction à Java

1. Introduction à la programmation objet

1.1 Limites des approches classiques

Pour un certain nombre de difficultés liées aux coûts, à l'évolutivité, et à la complexité du logiciel, il est nécessaire de pouvoir disposer de méthodes et moyens permettant de maîtriser la conception (respecter les délais et les spécifications, maîtriser les coût et la qualité, assurer la maintenabilité et l'évolutivité). Le génie logiciel a pour objet d'apporter une réponse à tous ces problèmes. Le génie logiciel (software engineering) désigne l'ensemble des méthodes, des techniques et outils concourant à la production d'un logiciel, au-delà de la seule activité de programmation. Il couvre donc toutes les phases du cycle de vie d'un logiciel allant de l'analyse des besoins jusqu'à l'intégration et la maintenance, mais aussi des activités transversales, comme la gestion de projet, la qualité, la réutilisation, ….

Les langages informatiques concernent plutôt les phases dites de développement : conception détaillée, réalisation et test unitaire du logiciel.

Le concept d'abstraction est largement utilisé pour identifier et regrouper des caractéristiques et traitements communs applicables à des entités ou concepts variés. Une représentation abstraite commune de telles entités permet d'en simplifier la manipulation. En informatique, l'abstraction n'a cessé de progresser, un langage assembleur est une abstraction de la machine sous-jacente (manipulation symbolique du code d'ordre). Les langages informatiques impératifs procéduraux (tels que le Pascal et le langage C) peuvent être aussi considérés comme des abstractions du langage assembleur, il sont centrés sur les traitements qui consomment et produisent des données. Ces langages constituent de nettes améliorations qui permettent de s'affranchir des particularités de la machine, ils offrent des primitives (structures de contrôle, variables, procédures, blocs) permettant de décrire déjà certains modes de résolution de problème.

Cependant, même avec ces langages déjà évolués, le programmeur doit établir l'association entre le modèle informatique du programme constitué de ses primitives et le modèle du problème à résoudre. Ces deux modèles étant généralement basés sur des primitives très différentes, les efforts requis pour réaliser une relation entre ces deux familles de primitives produit des programmes difficiles à réaliser et à maintenir.

1.2 Modèle objet

Le modèle **objet** au sens large, et plus particulièrement les langages objets, ont pour objectif de fournir des abstractions permettant de réduire la distance entre le modèle du problème et le modèle informatique. La force de la programmation objet, c'est qu'elle s'appuie sur un modèle calqué sur la réalité physique du monde.

Les objets se comportent comme des entités indépendantes. Un objet est **identifiable** (l'objet possède une identité, qui permet de le distinguer des autres objets) :

- La voiture immatriculée « 187 CBT 93 » est différente de la voiture « 456 AFB 59 », même si ces deux voitures correspondent au même modèle, avec les mêmes options, la même couleur, …

Chaque objet possède un **état interne** :

- La couleur « rouge », la vitesse « 50 km/h », le niveau d'essence « 20l », …

Chaque objet reçoit des **message**s provenant de l'extérieur (d'autres objets) :

- Accélère fort, freine doucement, mets toi en route, bloque tes portes, (en provenance du conducteur), mets en route des essuie-glaces (en provenance de la pluie), …

Un objet réagit à ces messages en effectuant des **opération**s (appelées aussi méthodes dans les langages objets) qui dépendent aussi de son état :

- Augmente de la valeur de la vitesse suite à une accélération, bloque les portes arrières car la sécurité enfant est enclenchée, ….

Un objet peut, en réponse à ces messages, lui même envoyer des messages vers d'autres objets :

- Affiche la valeur de la vitesse 50 km/h sur le compteur, émet d'un bip (car la vitesse dépassent 10 km/h et la ceinture du conducteur n'est pas attachée), …

L'ensemble des messages auquel l'objet peut réagir se nomme son **interface** qui correspond aux opérations qu'un objet sait faire, la manière dont il les fait s'appelle le comportement de l'objet et l'ensemble des valeurs que peuvent prendre ses caractéristiques se nomme les états possibles (ou espace d'états) de l'objet.

Une **classe** regroupe les objets ayant même structure (mêmes états possibles) et même comportement. La classe correspond à la structure d'un objet, c'est-à-dire la déclaration de l'ensemble des membres, c'est-à-dire les attributs et les opérations qui décrivent un objet. Ainsi, les objets de même nature possédant la même structure et le même comportement sont membres d'une même classe. Les objets d'une même classe auront donc des opérations identiques, des attributs de même type, mais avec des valeurs différentes. L'objet est relié à sa classe par la relation d'**instanciation**, un objet est l'instance d'une classe, et la classe est reliée à l'objet par la relation de classification. L'instanciation de la classe correspond à la fabrication d'un objet conforme à la classe qui est en quelque sorte un « moule à objets ».

Une classe peut être considérée comme un type abstrait de données caractérisé par des propriétés (attributs et opérations) communes à ses objets :

- Les attributs d'une voiture sont immatriculation, vitesse, couleur, en marche, …
- Les opérations d'une voiture sont accélérer, freiner, mouvoir les essuie glaces, …

1.3 Principes de la programmation orientée objet

Les langages orientés objets se fondent en particulier sur trois principes :

L'**encapsulation** consiste en une fusion entre les attributs et les fonctions appelées **méthodes** dans les langages objets, destinées à les manipuler. Le principe de **dissimulation** est associé à l'encapsulation : certains attributs et certaines méthodes sont cachés ; ils forment l'**implémentation**. Le premier avantage réside dans la capacité à pouvoir modifier la structure interne des objets ou les méthodes associées aux messages sans impact sur les utilisateurs des objets qui ne voient que l'interface. Cela évite aussi que l'utilisateur d'un objet le modifie de manière inconsidérée et le rende incohérent (préservation de l'intégrité de l'objet). Il s'agit de bien séparer l'implémentation de l'interface d'une classe, l'interface doit fournir aux utilisateurs des instances de la classe uniquement les informations nécessaires pour les manipuler correctement et rien d'autre. Les niveaux de **visibilité** (interface et implémentation) sont spécifiés par des **droits d'accès** aux membres. Il s'agit d'établir une « interface publique » et une « implémentation privée ». L'interface correspond à la spécification d'une classe et l'implémentation à sa réalisation.

L'utilisateur de l'objet voiture ne peut pas fixer brutalement la vitesse de son véhicule, il doit accélérer ou freiner.

L'**héritage** permet de construire de nouvelles classes, sous classes ou classes dérivées, à partir d'une classe préexistante, super classe ou classe de base, et héritant des attributs et des opérations de celle-ci, la sous classe possède tous les membres de sa super classe. Il est néanmoins (et heureusement) possible de définir dans une sous classe de nouvelles opérations et de nouveaux attributs, voire de remplacer ceux de la super classes par de nouveaux membres mieux adaptés aux problèmes à traiter, on dit que l'on spécialise la super classe. L'héritage est un outil puissant de réutilisation de classes par **spécialisation** de classes existantes basé sur la transmission des membres. Ainsi l'héritage permet de fabriquer des « moules » à partir d'autres « moules ». La relation d'héritage est une relation inter classes transitive, si la classe B hérite de la classe A et que la classe C hérite de la classe B, alors la classe C hérite de la classe A. Ainsi, on peut représenter sous forme de **hiérarchie de classes**, la relation de parenté qui existe entre les différentes classes possédant une classe de base commune.

La classe voiture hérite de la classe véhicule à moteur, comme la classe moto, des caractéristiques comme la cylindrée, la puissance ou la consommation lui seront transmises, par contre, le nombre de portes est une particularité des voitures qui n'aura pas de sens pour une moto.

Le **polymorphisme** vient du grec et signifie « qui peut prendre plusieurs formes ». Cette caractéristique est un des concepts essentiels de la programmation orientée objet. Il consiste à donner un même nom à une méthode (opération) qui est ensuite partagée à plusieurs niveaux d'une même hiérarchie de classe ou/et à des méthodes qui ont des types de paramètre différents. Le polymorphisme va permettre de choisir automatiquement de la bonne méthode à adopter en fonction du type de données passées en argument (opérandes de l'opération) et de la classe de l'objet à laquelle la méthode est appliquée.

Le comportement à l'accélération d'une voiture ou d'une moto étant différents, les méthodes associées le seront aussi, mais se nomment accélération.

Toutes ces caractéristiques permettent de développer des programmes plus fiables et plus faciles à faire évoluer et à maintenir. Un programme orienté objet est constitué d'un ensemble d'objets qui communiquent par envoi de messages (appels de méthodes

faisant partie de l'interface). L'encapsulation permet d'améliorer la fiabilité d'une application (dissimulation de l'implémentation) et de rendre plus facile l'utilisation des objets, les autres objets ne se préoccupent que de l'interface en ignorant les détails de codage. elle facilite l'évolutivité, si l'implémentation est modifiée, il suffit que l'interface ne le soit pas. L'héritage quant à lui va permettre de faciliter la réutilisation en créant de nouvelle classes à partir de classes existantes déjà éprouvées et testées.

De manière simplifiée, le développement d'un programme orienté objet consiste à partir du comportement souhaité d'une application à effectuer une conception par raffinements :

- A identifier un certain nombre de composants du système qui seront des objets
- A attribuer à chaque objet un ensemble de responsabilités, et organiser la délégation de celles-ci, le cas échéant vers d'autres objets, de telle sorte que :
  - Chaque composant doit se voir attribuer un ensemble de responsabilités simple et bien défini (des composants trop complexes sont difficiles à concevoir, à tester et à réutiliser)
  - Chaque composant doit interagir le moins possible avec les autres composants (facilite la conception et limite les risques d'effets de bord)
  - La réutilisation de classes existantes (par instanciation) ou de conception de classes dérivée à partir de classes de base existantes (par spécialisation) est privilégiée

2. Java

2.1 Histoire

Java est à la fois un langage de programmation informatique orienté objet et un environnement d'exécution informatique portable. Il fut lancé par Sun Microsystems en 1991 dans le cadre du projet Green, groupe de recherche spécialisé dans le développement de logiciels domotiques (contrôler des appareils électroniques). A l'origine le langage de programmation choisi fut le C++, mais l'idée fut abandonnée (faible niveau de la sécurité, pas de la programmation distribuée, pas de multi-threading, …) au profit du développement d'un nouveau langage de programmation qu'ils appelèrent Oak (chêne), plus souple.

En 1994, le Web qui semblait promis à un grand avenir fut visé. Le navigateur WebRunner fut écrit en Oak pour démontrer les capacités du langage. WebRunner fut ensuite renommé HotJava, puis le langage Oak fut renommé lui-même en Java. La version 1.0 alpha de Java fut disponible fin 1994. La version 1.0 définitive fut lancée en janvier 1996 et Java fut intégré au navigateur Netscape. Le succès des applets Java lance la plate-forme. Depuis plusieurs années, Java n'est plus réservé à la programmation d'applets pour le WEB. Il semble constituer une véritable alternative pour le développement d'applications importantes.

Il existe deux types de programmes avec la version standard de Java : les applets et les applications. Les applications s'exécutent dans le système d'exploitation à condition d'avoir installé une machine virtuelle. Les applets qui sont de petites applications destinées à fonctionner sur un navigateur.

A l'heure actuelle Java est le langage le « plus utilisé », d'après l'indice TIOBE (Mai 2008) qui classe les différents langages :

Java (20.5%)

C (14.7%)
Visual Basic (11.7%)
PHP (10.3%)
C++ (9.9%)

2.2 Caractéristiques

Java possède un certain nombre de caractéristiques qui le rende particulièrement attrayant.

Langage orienté objet

Java est un langage orienté objet proche syntaxiquement du C++, mais qui se veut syntaxiquement plus simple (c'est de moins en moins le cas). Un programme Java est constitué uniquement de classes (pas de variable globale, pas de fonction), ce qui n'est pas le cas de C++. Chaque fichier source contient la définition d'une ou plusieurs classes qui sont utilisées les unes avec les autres pour former une application.

Portabilité

Classiquement, la chaîne de compilation consiste à transformer un fichier source, écrit par exemple en langage C, en un fichier exécutable (programme). Le fichier exécutable est dépendant du système d'exploitation (et du microprocesseur) de la machine cible. Ainsi un fichier exécutable sur une machine disposant de Linux, ne le sera pas sur une machine disposant de Windows. Dès que l'on désire changer de système d'exploitation, il faut recompiler le programme source.

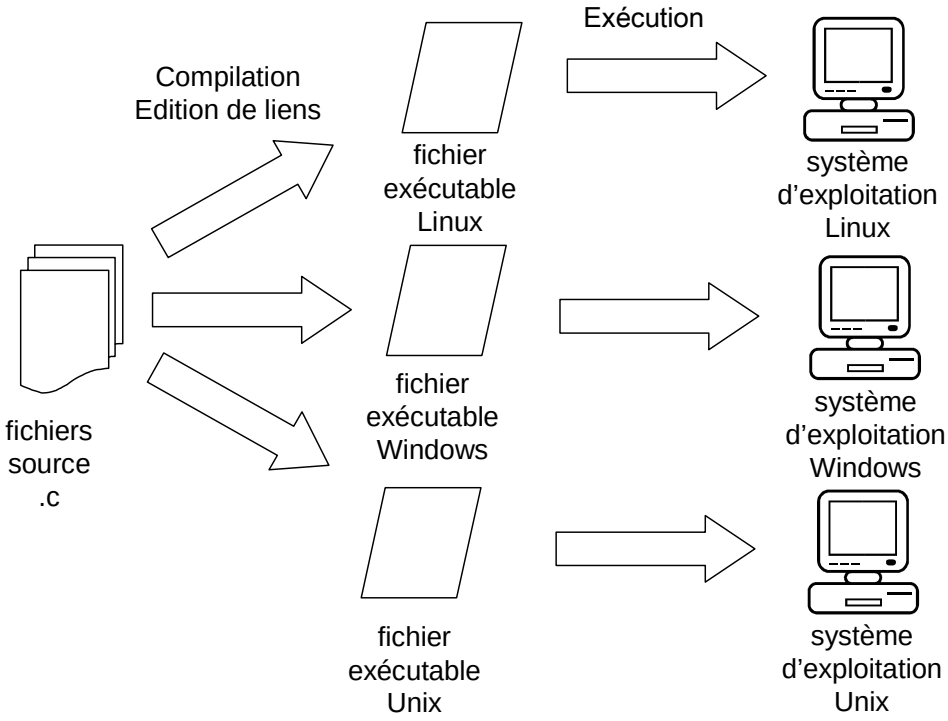


Figure 1 : Chaîne de compilation classique

Le compilateur Java ne fournit pas directement du code machine exécutable, il produit un code portable, du byte-code, interprétable par la machine virtuelle : JVM (JAVA Virtual Machine). Ce byte-code est indépendant du système d'exploitation de la machine cible. Chaque machine possédant une machine virtuelle peut « exécuter » une application Java, c'est la machine virtuelle qui interprète le byte-code et le convertit en code natif.

Cette caractéristique est majeure, car c'est elle qui fait qu'un programme écrit en Java est portable, c'est-à-dire qu'il ne dépend pas d'un système d'exploitation donné. Les navigateurs permettant d'utiliser des applets possèdent une machine virtuelle.

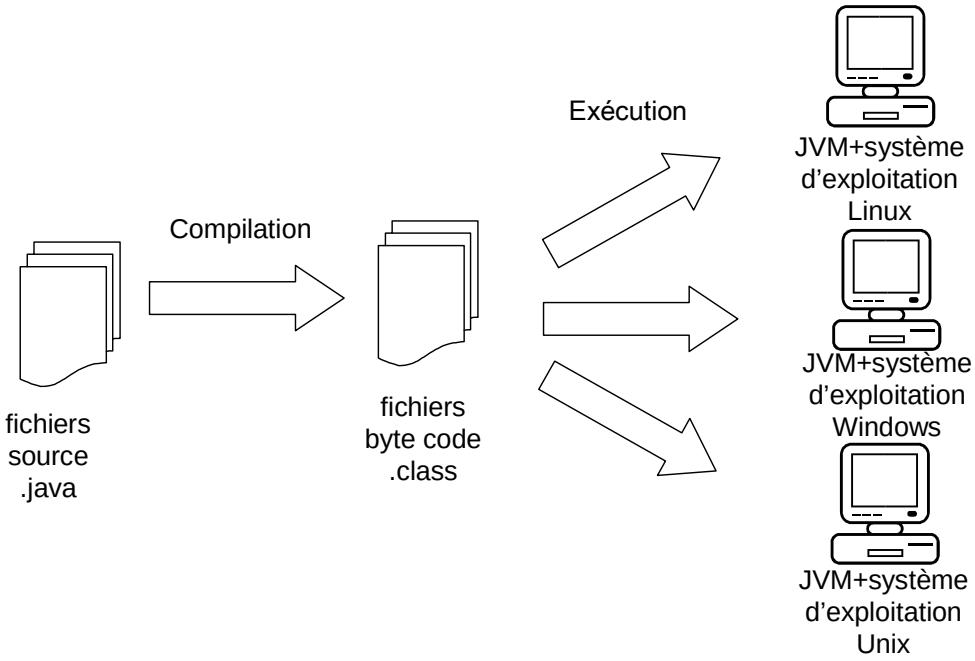


Figure 2 : Chaîne de compilation Java

Robustesse

La gestion de la mémoire est effectuée automatiquement par un ramasse-miettes (garbage collector). Dans le code on ne manipule pas des pointeurs mais des références. Le programmeur n'a pas à gérer la destruction de la mémoire allouée.

De nombreuses vérifications sont effectuées à la compilation et à l'exécution, en particulier, le langage Java est fortement typé.

Java dispose d'un mécanisme d'exception intégré à la machine virtuelle pour traiter les erreurs. Une exception peut être un accès à une référence nulle, indice non valide d'un tableau, l'ouverture d'un fichier inexistant, division d'un entier par zéro, ... Le compilateur est contraignant : il vérifie que certaines exceptions sont traitées.

Les applets fonctionnant sur le Web sont soumises à certaines restrictions, en particulier, elles ne peuvent pas accéder aux ressources de la machine sur laquelle elle s'exécutent (fichier ou autre programme) .

Multi-tâche

Java est multi-threads. Un thread est une partie de code s'exécutant en concurrence avec d'autres flots d'instructions dans un même processus. Les threads, parfois appelés « processus légers »' permettent de développer des programmes multitâches. Les avantages principaux du multi-threads sont des performances plus élevées en matière d'interactivité et un meilleur comportement en temps réel, bien que ce dernier soit en général dépendant du système.

Les threads faisant partie intégrante du langage Java, ils sont plus faciles à utiliser que leurs équivalents C/C++. Java intègre les threads de manière transparente (garbage collector, horloge, chargement des images ou des sons), mais permet également au programmeur de développer ses propres threads.

API Java

Sun fournit gratuitement un ensemble d'outils et d'API (Application Programmable Interface) pour permettre le développement de programmes avec Java. Ces API permettent de prendre en compte :

Réseau et Internet

Java possède une importante bibliothèque de routines permettant de gérer les protocoles TCP/IP tels que HTTP et FTP. Les applications Java peuvent charger et accéder à des pages Web sur Internet via des URL avec la même facilité qu'elles accèdent à un fichier local sur le système. Elles permettent de programmer un serveur de socket pouvant accepter des connexions en parallèle. Java dispose d'un mécanisme d'invocation de méthode à distance (RMI) et autorise la communication entre objets distribués.

Interfaces Graphiques

Java propose plusieurs API permettant de développer des interfaces graphiques. Elles fournissent en particulier des classes pour représenter des éléments d'interface comme les fenêtres, des boutons, des boîtes combo, des boîtes de dialogues, des menus, … Ces API sont regroupées sous le nom Java Foundation Classes, les deux plus connues étant Swing et AWT.

Bases de données

La technologie de connectivité de bases de données de Java, appelée JDBC (Java Database Connectivity) est assurée par une bibliothèque de classes qui permet de travailler avec différents serveurs de bases de données relationnelles de manière transparente.

XML[1]

JAXP (Java API for XML Processing) est l'API "standard" (appartenant au JDK) pour la manipulation du format XML. Cette API contient en fait plusieurs API, dont SAX, DOM, TrAX. permettant la création, la manipulation et le traitement de fichiers XML à bas niveau. Ces API permettent par exemple d'analyser les fichiers XML, et de les transformer à l'aide de feuille de style XSL.

Autres

Il existe de très nombreuses autres API permettant de gérer des collections, de traiter des expressions régulières, de réaliser des animations en 3 dimensions, d'exploiter des fichiers multimédias, … certaines de ces API font partie du JDK, d'autres sont des API supplémentaires qui ne sont pas initialement fournies en standard dans le JDK et enfin d'autres sont directement développées par la communauté Java (GNU, Apache Software Foundation, …). Les meilleures API sont progressivement intégrées au JDK de SUN.

Le graphique suivant montre l'évolution du nombre de classes intégrées aux différentes versions du JDK.

[1] eXtensible Markup Language : Langage dont l'objectif initial est de faciliter l'échange automatisé de contenus entre systèmes d'informations hétérogènes.

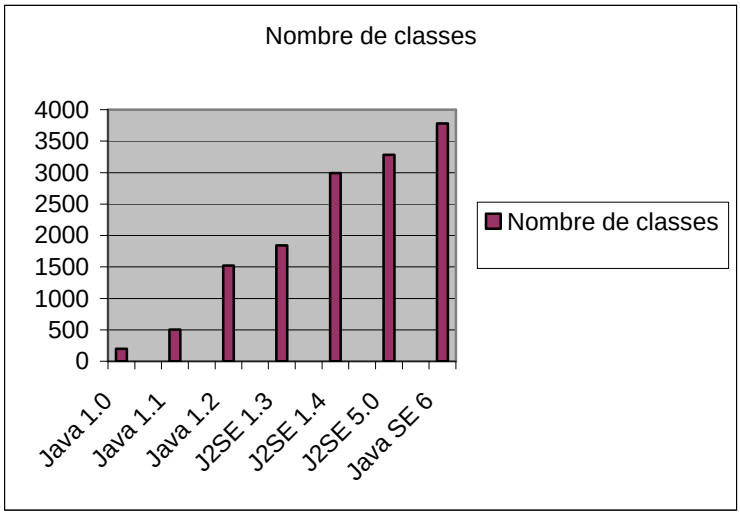


Figure 3 : Evolution du nombre de classes intégrées au JDK

Outils, environnements et plates- formes

Sun fournit gratuitement un ensemble d'outils et d'API (Application Programmable Interface) pour permettre le développement de programmes avec Java, ce kit est nommé le JDK. Le JRE (Java Runtime Environment) contient uniquement l'environnement d'exécution de programmes Java. Le JDK contient lui même le JRE. Le JRE seul doit être installé sur les machines où des applications Java doivent être exécutées. Le JRE et le JDK, sont téléchargeables sur le site web de Sun http://java.sun.com/.

Le JDK est constitué des outils de développement de base :

- java : Machine virtuelle Java, permet l'exécution des programmes
- javac : Compilateur Java.
- appletviewer : Machine virtuelle Java permettant l'exécution des applets.
- jar : Permet la création et la manipulation d'Archives Java (JAR).
- javadoc : Générateur de documentation Java, au format HTML.
- javap : Désassembleur de classes Java compilées.
- jdb : Débogueur Java en ligne de commande.

Il existe de nombreux environnements de développement intégrés très complets permettant d'intégrer des outils puissants (édition de code, développement pour les services web, analyse de code, gestion de projet, intégration de tests, …) : Eclipse, NetBeans, IntelliJ, JDeveloper, JCreator, … dont la majorité sont gratuits

Par ailleurs, Sun fournit plusieurs éditions afin de permettre l'utilisation de Java pour des usages très diversifiés. On distingue essentiellement 4 grandes éditions :

- JSE : Cette édition est destinée aux applications pour poste de travail.
- JEE : Cette édition est spécialisée dans les applications serveurs. Il contient pour ce faire un grand nombre d'API et d'extensions.
- JME : Cette édition est spécialisée dans les applications mobiles.
- JavaCard : Cette édition est spécialisée dans les applications liées aux cartes à puces et autres SmartCards.

La nourriture intellectuelle donnée par l'instruction est comparable à la nourriture matérielle. Ce n'est pas ce qu'on mange qui nourrit, mais seulement ce qu'on digère..

Gustave Le Bon

Concepts non objet

1. Introduction

Dans cette première partie, nous introduisons, en particulier, les nouveautés du Java par rapport au langage C qui ne concernent pas l'aspect objet du Java. Certaines sont de véritables améliorations : typage plus fort, enrichissement des types, possibilité de surcharge de méthodes, … , d'autres, présentent moins d'intérêt comme les instructions de ruptures étiquetées, plutôt dangereuses ou la boucle Java 5 simplifiant l'exploitation des collections, mais peu intéressante dans un autre contexte. Les concepts de base communs à Java et à C sont rapidement décrits au travers d'illustration.

1.1 Premier programme introductif

Ce cours de Java s'adressant à des utilisateurs ayant connaissance du langage C ou du langage C++, nous proposons d'introduire la structure minimum de base d'un programme Java. Cette structure nous permettra d'illustrer les concepts de ce langage (type, variable, expression, instruction, fonction, …) au travers de rappels avant d'aborder les notions orientées objets. Les aspects non objets propres au langage Java et absents du langage C seront quant à eux développés (type booléen, boucle « for each », signature de méthode, référence, particularités des tableaux, …).

Le Java est plus contraignant sur l'aspect objet que le C++, un programme Java doit être uniquement constitué de classes, contrairement au C++ qui permet de manipuler des fonctions indépendantes de toute classe. Par ailleurs, un programme Java doit posséder un point d'entrée équivalent d'une fonction main. Nous proposons un exemple qui nous servira de canevas pour la présentation des notions de base.

Le programme suivant contient l'unique classe `Exemple1` qui possède une unique fonction membre (que l'on appelle aussi méthode) `main`. Le mot clef `public` signifie que la classe et la fonction sont accessibles à l'extérieur de la classe et le mot clef `static` signifie que la fonction est accessible même si aucune instance de la classe `Exemple1` n'est créée (ces notions seront détaillées dans la suite). Deux variables locales à `main` sont créées `i` de type `int` (entier) et `x` de type `float` (réel) initialisée à `0`. Une boucle `for` est définie, initialement, la variable i est initialisée à `0` (dans la première expression du `for`), et tant que la valeur de cette variable est inférieure à `5` (seconde expression du `for`) le bloc qui suit est exécuté, après chaque exécution du bloc la variable `i` est incrémentée (dernière expression du `for`). Dans ce bloc, la valeur de la variable `i` est affichée puis un espace, la valeur de l'expression `2*x` est affichée, puis la variable `x` est incrémentée

(l'incrémentation effective est exécutée après), enfin un espace est affiché. On retrouve comme en langage C, des variables, des expressions, des instructions.

Programme :

```
public class Exemple1 {
  public static void main(String args[]){
  int i;
  float x=0f;
    for (i=0;i<5; i++) {
      System.out.print(i + " ");
      System.out.print(2*x++ + " ");
    }
  }
}
```

Résultat de l'exécution :

```
0 0.0 1 1.0 2 2.0 3 3.0 4 4.0
```

Après avoir édité le programme source Exemple1.java avec un éditeur de texte, par exemple, celui-ci doit être compilé puis exécuté avec les commandes :

```
javac Exemple1.java
java Exemple1
```

qui produira le résultat de l'exécution donné.

Un fichier source Java doit porter le même nom que la classe publique qui contient la méthode `main` (la casse doit être identique).

1.2 Commentaires

Par définition un commentaire est une partie du code source qui n'est pas prise en compte par le compilateur. En programmation les commentaires permettent de comprendre le source lors de sa relecture. Les commentaires sont importants, comme les conventions de codage, ils aide à la compréhension du code, en particulier, lors des opérations de maintenance, il ne faut donc pas les négliger (sans en abuser). Un commentaire peut être introduit par `//`. Tout le reste de la ligne ne sera pas pris en compte par le compilateur.

```
int i = 0 // la variable i est initialisee à 0 (commentaire ridicule)
```

Comme en langage C, les commentaires peuvent également être compris entre `/*` et `*/`. Dans ce cas, ils peuvent s'étendre sur plusieurs lignes.

```
/*
Méthode d'inversion de matrice basée sur
l'algorithme de Gauss
*/
```

Il existe une troisième catégorie de commentaires permettant de générer une documentation d'API en format HTML depuis un code source Java. Javadoc est le standard pour la documentation des classes Java, ces commentaires commencent par /**. Il ne sont pas étudiés ici.

2. Données et variables

Java étant un langage fortement typé, toute variable doit être déclarée avant d'être utilisée (Il en va de même pour le type de retour des méthodes). Le langage Java possède deux catégories de types qui vont permettre de créer des données :

- Les types primitifs ou prédéfinis ou encore de base, comme les réels, les booléens, les entiers, les caractères, ..., ils ne sont pas construits à partir d'autres types (§2.1 p14).
- Les types composés qui sont construits à partir d'autres types comme les tableaux, les classes, les énumérations ou les interfaces. Les API Java (Application Programming Interface : bibliothèques utilisées pour écrire des applications) possèdent de nombreux types composés `String`, `File`, `Color`, … facilement réutilisables par les programmeurs.

Dans cette partie, on s'intéresse uniquement aux données de type primitif. Il est à noter que les chaînes de caractères en Java ne constituent pas un type primitif, mais une classe (`String`), et sont très intégrées au langage. Par ailleurs, à chaque type primitif, correspond une classe Java, qui permet d'associer un objet à une valeur, ce point sera abordé dans la suite (§11.2 p126).

2.1 Types primitifs

Java dispose de huit types de données prédéfinis, dits types primitifs (ou types de base), dont les domaines de valeurs sont fixés, à l'opposé du langage C dont les domaines dépendent de l'implémentation. En Java, les variables membres d'un objet (définies comme attribut de l'objet) ou les composantes d'un tableau sont initialisées avec une valeur par défaut, en accord avec son type, au moment de la création, ces valeurs par défaut correspondent à la « valeur nulle ». Cette initialisation automatique ne s'applique pas aux variables locales.

Quatre types d'entiers sont définis, deux types de réels[2], le type booléen et le type caractère (`char`).

Type	Taille (octets)	Domaine de valeurs	Valeur par défaut
`boolean`	-	`false, true`	`false`
`byte`	1	`-128 à 127`	`0`
`short`	2	`- 32768 à 32767`	`0`
`int`	4	`-2147483648 à 2147483647`	`0`
`long`	8	-2^{63} à $2^{63}-1$	`0l`
`float`	4	$\pm 1.40239846\ 10^{-45}$ à $\pm 3.40282347\ 10^{38}$ précision : 7 chiffres	`0.0f`
`double`	8	$\pm 4.0406564584124654\ 10^{-324}$ à $\pm 1,797693134862316\ 10^{308}$ précision : 15 chiffres	`0.0`
`char`	*2*	`unicode(u0000) à unicode(uffff)`	`u0000`

Tableau 1 : Types de données (taille et domaine de valeurs)

2.2 Constantes littérales

Comme en langage C, il est possible de définir des constantes littérales. Ces constantes sont, en particulier, très utilisées pour initialiser les variables. On dispose, comme en langage C, des trois formats pour les entiers (correspondants à trois bases) :

- décimal : `1`, `16`, `527` (nombre constitué des caractères 0-9)

[2] Conformes à la norme IEEE 754: Standard for Binary Floating-Point Arithmetic

- octal : `01`, `020`, `01017` (nombre constitué des caractères 0-7 et préfixé par un 0)
- hexadécimal : `0x1`, `0x10`, `0x20F` (nombre constitué des caractères 0-9 et/ou a-f en majuscules ou en minuscules et préfixé par un 0x)

Pour spécifier qu'il s'agit d'une constante de type long, il suffit de la suffixer par `l` ou `L`.

Pour préciser qu'une constante littérale décimale correspond à un nombre réel, il faut que celle-ci possède un point ou un exposant ou qu'elle soit suffixée par `f` ou `F`. Par défaut, une constante décimale possédant un point ou un exposant est de type `double`. Afin de préciser qu'il s'agit d'une constante de type `float`, il faut obligatoirement la suffixer par `f` ou `F`.

Les constantes littérales booléennes n'ont que deux valeurs possible `true` et `false`.

Les constantes de types caractères sont constituées d'un unique caractère entre apostrophe (') ou d'un Unicode de la forme '\uXXXX', XXXX correspond à un nombre en hexadécimale. Les méta caractères du langage C sont aussi reconnus, ils correspondent à des séquences d'échappement pour les caractères non imprimables (`'\n'`, `'\t'`, …)

2.3 Déclarations et initialisations

La déclaration d'une variable permet de réserver la mémoire pour en stocker la valeur. Une variable possède : un nom, un type et une valeur (et un emplacement mémoire).

Une déclaration de variable a toujours la forme suivante :

```
[final] type identificateur [= valeur initiale];
```

Le type d'une variable correspond à un type primitif ou à une classe.

Un identificateur Java peut commencer par un lettre, par le caractère de soulignement (_) ou par le signe dollar ($). Le reste de son nom peut comporter des lettres ou des nombres, des caractères de soulignement ou des signes dollar. Java est sensible à la casse, ainsi `Toto` et `toto` sont deux identificateurs distincts.

Le mot clé optionnel `final` spécifie que la valeur de la variable ne peut pas être modifiée, il s'agit d'une constante qui conserve la même valeur pendant toute sa durée de vie (à l'opposé d'une constante littérale, celle-ci possède un emplacement mémoire). Par convention l'identificateur d'une constante est tout en majuscules (`PI`, `MAX_VALUE`, …).

Par convention, l'identificateur d'une variable locale ou une donnée membre non constante est basé sur la notation hongroise, si c'est un identificateur simple, il n'est composé que de minuscules (`i`, `index`, `tab`). Si c'est un identificateur complexe, on concatène les mots que le composent, le premier mot est toujours écrit en minuscule puis la première lettre de tous les mots suivants en majuscule (`indexMax`, `plusGrandElement`, …).

La valeur initiale correspond à la valeur que l'on donne à la variable au début de sa vie, c'est une expression (on utilise fréquemment une constante littérale).

L'exemple de code Java suivant illustre les possibilités d'initialisation de variables. Il est à noter qu'il est possible d'effectuer des déclarations multiples (variables séparées par des virgules partageant un même type).

Exemple :

```
int x = 0, y = 0;
final float PI = 3.141F;
```

```
byte bb = 3;
double w = 3.1415, z;
long t = 99;
boolean test = true;
char c = 'a';
char d = '\u010F';
final short E;
```

Comme dans le cas de tous les langages fortement typés[3], toute variable doit être déclarée avant d'être utilisée. Le langage Java est plus contraignant que le langage C car, en Java, les variables locales doivent aussi être initialisées avant d'être utilisées. Les variables locales ne sont pas initialisées par défaut (En langage C, elles le sont avec une valeur aléatoire). Ainsi, la séquence suivante :

Exemple :

```
int i;
int a = 2*i;
```

provoquera l'erreur de compilation : « `variable i might not have been initialized` »

Dans le cas de l'initialisation d'un réel de type `float` avec une constante littérale réelle, il est nécessaire de suffixer celle-ci avec `f` ou `F`. Ainsi, dans l'exemple suivant, la première déclaration est licite et la seconde provoquera l'erreur de compilation : « `possible loss of precision` »

Exemple :

```
final float PI = 3.141F;
final float E = 2.71828;
```

2.4 Portée locale

Lorsqu'une variable est déclarée dans un bloc ou en tant que paramètre d'une méthode, il s'agit d'une variable locale. Une variable locale devient utilisable dans le bloc où elle est définie dès qu'elle a été initialisée.

Par ailleurs, il existe des règles de portée, la portée d'une variable locale correspond au bloc dans lequel celle-ci a été déclarée ; ainsi deux variables possédant le même nom ne peuvent pas coexister dans le même bloc, mais le peuvent dans des blocs distincts dans ce cas, il s'agit de variables distinctes.

Dans le programme suivant, trois variables locales `i` différentes sont déclarées dans la méthode `main` sans que cela pose de problème. Cependant la dernière déclaration n'aurait pas pu être remontée au début du bloc de la méthode `main` (erreur de compilation : « `i is already defined in main(Java.lang.String[])` »), car il n'y a pas de masquage d'une variable locale appartenant à un bloc par une variable locale appartenant dans un bloc imbriqué.

Programme :

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

[3] Un langage est fortement typé si le type de données est associé au nom de la variable (il existe de nombreuses définitions).

```
  {
   int i = 2;
   System.out.println(i);
  }
  for (int i = 2; i < 5; i++){
   System.out.println(i);
  }
  int i = 1;
  System.out.println(i);
  }

}
```

2.5 Position de la déclaration d'une variable locale

En langage C, les variables locales doivent être déclarées soit au début d'une fonction, soit au début d'un bloc, avant la première instruction. En Java (comme en C++), il suffit que la variable locale soit déclarée avant son utilisation ; ceci est même possible dans la partie initialisant un `for`, ou toute autre structure de bloc (`while`, `switch` …). Dans ce cas, la portée de la variable est limitée au bloc.

Dans le programme suivant, trois variables locales sont déclarées. La variable max peut être utilisée n'importe où dans le `main`. La variable `s` ne peut être utilisée que dans la boucle. Ainsi, le programme suivant donnera lieu à deux reprises à la même erreur de compilation : « `cannot find symbol variable s` » (idem pour `i`) .

Programme :

```
public class Test {
 static public void main(String [] args) {
  int max = 10;
  for(int i = 0; i < max; i++) {
   int s = 0;
   s+=i;
  }
  System.out.println(s); // erreur de compilation
  System.out.println(max);
  System.out.println(i); //  erreur de compilation
 }
}
```

2.6 Initialisation différée

L'initialisation[4] peut être faite lors de la déclaration ou plus loin dans le code. Cette règle s'applique même pour une constante (évidemment, une constante, une fois initialisée ne pourra plus être modifiée). La séquence suivante est donc parfaitement licite.

Exemple :

```
{
```

[4] Attention, ces règles diffèrent pour les données membres des classes

```
int i;
i = 0x5;
System.out.println(i);
int a = 2*i;
final int p;
a++;
p = a;
…
```

3. Expressions et opérateurs

3.1 Introduction

Le langage Java dispose d'un certain nombre d'opérateurs qui permettent d'effectuer des opérations sur des opérandes ; ce sont des calculs élémentaires. Les opérateurs associés aux opérandes forment des expressions (une expression possède toujours une valeur qui a implicitement un type). Ces opérateurs sont identiques à ceux du Langage C (à quelques exceptions près : pas d'opérateur `sizeof`, d'indirection (`*`) ou d'adressage (`&`) et un opérateur de décalage supplémentaire).

Le programme suivant illustre l'utilisation des principaux opérateurs arithmétiques applicables aux entiers. Les premiers opérateurs peuvent aussi être appliqués aux types réels (`%` modulo compris). Les opérateurs `|`, `&` et `^` sont aussi applicables aux booléens.

On peut noter que Java différencie les décalages arithmétique (>>) et logique (>>>) à gauche. Lorsqu'il s'agit d'un nombre négatif (bit de poids fort à `1`), dans le cas d'un décalage arithmétique des `1` sont ajoutés (conservation du signe), dans le cas d'un décalage logique, des `0` sont ajoutés (perte du signe).

Programme :

```
public class Test {
 static public void main(String [] args) {
  int a = 123, b = 5;
  System.out.println("a + b = " + (a + b));
  System.out.println("a - b = " + (a - b));
  System.out.println("a * b = " + (a * b));
  System.out.println("a / b = " + (a / b));
  System.out.println("a % b = " + (a % b));
  System.out.println("a << 2 = " + (b << 2));
  System.out.println("a >> 3 = " + (a >> 3));
  System.out.println("a >>> 3 = " + (a >>> 3));
  System.out.println("-a >>> 3 = " + (-a >>> 3));
  System.out.println("-a >> 3 = " + (-a >> 3));
  System.out.println("a | b = " + (a | b));
  System.out.println("a & b = " + (a & b));
  System.out.println("a ^ b = " + (a ^ b));
  System.out.println("~a = " + (~a));
 }
}
```

Résultat de l'exécution :

```
a + b = 128
a - b = 118
```

```
a * b = 615
a / b = 24
a % b = 3
a << 2 = 20
a >> 3 = 15
a >>> 3 = 15
-a >>> 3 = 536870896
-a >> 3 = -16
a | b = 127
a & b = 1
a ^ b = 126
~a = -124
```

3.2 Opérateurs logiques et booléens

Le langage Java dispose du type booléen qui doit être obligatoirement utilisé pour exprimer une condition logique, par exemple, la condition logique d'un `if`. Il est impossible, comme en langage C, d'utiliser par exemple une valeur numérique entière pour exprimer une condition. Du reste, le résultat d'une comparaison en Java est de type booléen (`a==b`, `c < d`, `a != f`, …).

Ainsi la séquence suivante, licite en langage C, permettant de faire la somme des `99` premiers entiers positifs, ne se compile pas en Java. Le type des expressions dans les conditions du `if` et du `while` étant de type entier et non booléen, on aurait eu deux fois l'erreur de compilation : « `incompatible types` ». Pour corriger le problème, il faut utiliser un opérateur de comparaison (`>`, `<`, `==`, `<=`, `!=`, `>=`) qui lui donne forcément un résultat de type booléen.

Exemple :

```
int s = 0, i = 100;
  while(i--){
   if(i & 1) s += i; // ne se compile pas
  }
```

Exemple corrigé :

```
int s = 0, i = 100;
  while(i-- != 0){
   if((i & 1) == 1) s += i;
  }
```

Les booléens peuvent être combinés avec les opérateurs logiques (`&&`, `||`, `!`) et quelques opérateurs de manipulation de bits (`&`, `|`, `^`). Le programme suivant illustre l'utilisation de ces opérateurs.

Programme :

```
public class Test {
 static public void main(String [] args) {
  boolean a = true, b = false;
  System.out.println("a | b = " + (a | b));
  System.out.println("a & b = " + (a & b));
  System.out.println("a ^ b = " + (a ^ b));
  System.out.println("a || b = " + (a | b));
  System.out.println("a && b = " + (a & b));
```

```
   System.out.println("!b = " + (!b));
  }
 }
```

Résultat de l'exécution :

```
a | b = true
a & b = false
a ^ b = true
a || b = true
a && b = false
!b = true
```

3.3 Conversion et promotion

Dans le cas d'une valeur d'un type numérique qui peut être convertie vers un autre type numérique sans risque de perte d'information (entier ou caractère vers entier de taille supérieure, réel vers réel de taille supérieure, entier ou caractère vers réel), on peut parler de compatibilité ascendante. La valeur impliquée est convertie automatiquement à la compilation puis à l'exécution, la conversion peut être implicite. On donne ici les conversions implicites possibles (les autres étant obtenues par transitivité) :

`byte` → `short` → `int` → `long` → `float` → `double`

`char` → `int`

Ainsi, on peut affecter à une variable de type `long` une valeur de type `byte` ou à une variable de type `double` une valeur de type `char`.

Les conversions avec risque de perte d'information sont refusées par le compilateur. Le programmeur doit explicitement utiliser l'opérateur de conversion de type (transtypage ou casting). Dans ce cas, le programmeur prend la responsabilité de la conversion qui pourrait provoquer des résultats inattendus suite à une perte d'information, les règles de conversion sont les suivantes :

- La conversion d'un type entier vers un autre type entier codé sur moins de bits s'effectue par troncature : on perd les bits de poids forts (les plus à gauche).
- La conversion d'un réel vers un entier commence par une conversion de la valeur du flottant en `int` ou `long`, puis il y a une conversion en appliquant la règle précédente.

Le programme suivant illustre ces mécanismes de conversion avec les pertes d'information qui en résultent.

Programme :

```
public class Test {
 public static void main(String [] args){
 double x = 1.5;
 int i = (int) x;
 System.out.println(x + " " +i);
 x = 1.5e40;
 i = (int) x;
 System.out.println(x + " " +i);
 int j = 123456;
 byte k = (byte) j;
 System.out.println(j + " " +k);
```

```
  }
 }
```

Résultat de l'exécution :

```
1.5 1
1.5E40 2147483647
123456 64
```

Le type `boolean` est incompatible avec les autres types, aucune conversion, même explicite n'est possible.

Dans l'exemple qui suit aucune des affectations de `boolean` vers `int` et de `int` vers `boolean` n'est possible. Chacune des quatre dernières lignes donne lieu à la même erreur de compilation : « incompatible types ».

Exemple :

```
boolean b = false;
int i = 0;
b =  i;
b = (boolean) i;
i = b;
i = (int) b;
```

Le programmeur peut réaliser la conversion à l'aide d'une expression (ou d'une méthode), par exemple :

```
boolean b = false;
int i = 0;
b =  i==0 ? false : true;
i =  b==false ? 0 : 1;
```

Les règles de conversion et de promotion dans les expressions sont identiques à celles du C. Cependant, le langage Java étant plus fortement typé ces règles ont des conséquences plus immédiates qui peuvent désorienter un programmeur C. L'exemple suivant met en évidence les conversions automatiques et leurs implications. Les trois affectations donnent lieu à une erreur de compilation de type « `possible loss of precision` » :

- `b3 = b1+1;` on additionne un `byte` à un `int` le résultat est de type `int` (la valeur de `b1` étant convertie en `int` avant d'effectuer l'addition), `b3` étant de type `byte`, il y a risque de perte d'information.
- `b3 = b1 + b2;` on additionne deux bytes, il y a promotion automatique des petits entiers en `int`, le résultat est de type `int`, `b3` étant de type `byte`, il y a risque de perte d'information.
- `b3 = x + b1;` on additionne un `byte` à un `float` le résultat est de type `float` (la valeur de `b1` étant convertie en `float` avant d'effectuer l'addition), `b3` étant de type `byte`, il y a risque de perte d'information.

Il suffit d'effectuer une conversion explicite du résultat de l'addition dans le type de la variable affectée.

Exemple :

```
byte b1 = 1, b2 = 2, b3;
float x = 1.0f;
 b3 = b1+1;
```

```
   b3 = b1 + b2;
   b3 = x + b1;
```

Exemple corrigé :

```
byte b1 = 1, b2 = 2, b3;
 float x = 1.0f;
  b3 = (byte)(b1+1);
  b3 = (byte)(b1 + b2);
  b3 = (byte)(x + b1);
```

3.4 Synthèse des opérateurs

Le tableau suivant récapitule les opérateurs du langage Java : priorité, définition et associativité. Les opérateurs sont classés par priorité décroissante.

Priorité	opérateur	définition	Associativité
1	[]	indiçage	G
1	()	appel de fonction	G
1	.	sélection directe	G
1	-- ++	incrémentation postfixé	D
2	-- ++	incrémentation suffixé	D
2	~	complémentation	D
2	!	négation logique	D
2	-	moins unaire	D
2	(type)	modification de type	D
3	* / %	multiplication, division, modulo	G
4	+ -	addition, soustraction	G
5	<< >> >>>	décalage à gauche, à droite	G
6	< > <= >=	comparaisons strictes et larges	G
7	== !=	égalité et inégalité logiques	G
8	&	ET arithmétique	G
9	^	OU exclusif arithmétique	G
10	|	OU arithmétique	G
11	&&	ET logique	G
12	||	OU logique	G
13	? :	condition ternaire	D
14	= *= /= %=-= <<= >>= &= ^= |=	opérateurs d'affectation	D
15	,	évaluation d'expressions	G

Tableau 2 : Opérateurs (règles d'association et priorité)

4. Tableaux

4.1 Introduction

Le langage Java, comme la plupart des langage de programmation, offre la possibilité de définir et manipuler des tableaux; toutefois Java, qui est un langage avec des objets, manipule les tableaux d'une façon particulière que l'on va décrire ici. On rappelle qu'un tableau est un ensemble de données partageant le même type, chaque donnée, appelée composante ou élément, est accessible au travers de son indice. En Java, comme en langage C, les indices d'un tableau vont de zéro à la taille du tableau moins un. Ainsi, les indices d'un tableau de taille 5 pourront prendre les valeurs 0, 1, 2, 3, 4, mais pas 5.

Dans l'exemple suivant, on déclare un tableau d'entiers. Ce tableau est initialisé à la déclaration. On peut noter que lors de la déclaration, les crochets se trouvent à gauche de la variable (on aurait pu les mettre, comme en langage C, à droite : `int t[] = {1,2,3}`[5]`;`). Les éléments de ce tableau sont manipulés via leur indice, `t[1]` désigne la valeur de la composante d'indice `1` du tableau `t`. L'écriture `t[3]` aurait provoqué l'erreur d'exécution « `Exception in thread "main" Java.lang.ArrayIndexOutOfBoundsException: 3 at Test.main …` » qui signifie que l'indice est trop grand.

Exemple :

```
public class Test {
 static public void main(String []args) {
  int [] t =  {1,2,3};
  System.out.println(t[0]);
  System.out.println(t[1]);
  System.out.println(t[2]);
  t[2] = -1;
  System.out.println(t[2]);
 }
}
```

Résultat de l'exécution :

```
1
2
3
-1
```

4.2 Référence

Une variable possède un emplacement mémoire. La référence d'une variable est l'adresse mémoire où est stockée la variable. Les variables de type primitif sont manipulées par valeur, les variables des autres types (objet et tableau) sont manipulées par référence. Ainsi, la valeur d'une variable (correspond à ce qui est dans l'emplacement mémoire associé) de type `int` est un entier, la valeur d'une variable de type `char` est un caractère et plus généralement la valeur d'une variable de type primitif est une valeur de

[5] Cet ensemble de valeurs entre «`{}`»est appelé une liste d'initialisation

ce type. En revanche, la valeur d'une variable de type non primitif est l'adresse d'une donnée de ce type. Un tableau se manipule via sa référence.

La séquence suivante déclare et initialise une variable de type primitif `a` et un tableau d'entiers `ta` en utilisant une liste d'initialisation.

```
int a = 4;
int [] ta =  {1,2,3};
```

La figure suivante décrit la façon dont les données sont stockées. La variable entière `a` dont la valeur est `4` désigne directement l'emplacement dans lequel cette valeur est stockée. La variable `ta` désigne un emplacement mémoire dans lequel se trouve l'adresse (`@3e25a`) des données du tableau (son contenu), l'accès s'effectue de manière indirecte. Nous avons utilisé ici deux représentations une physique (gauche) et une symbolique (droite) que nous utilisons dans la suite.

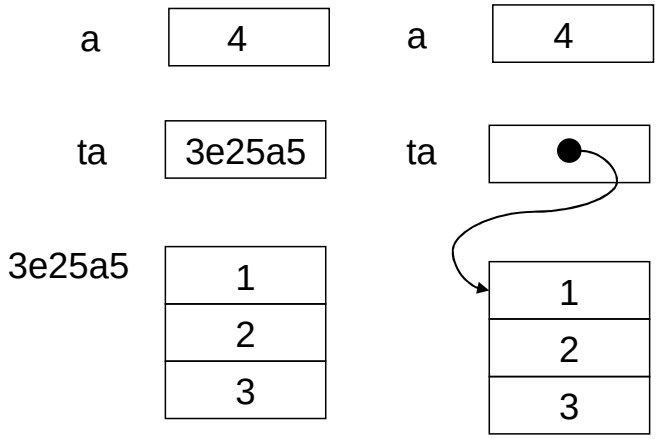


Figure 4 : Variable de type simple et référence

La manipulation de références est plus limitée que la manipulation des données de types de base : on peut seulement les comparer et les affecter, mais on ne peut pas les diviser, les sommer ou effectuer un décalage ... Lorsqu'on modifie une référence, cela n'entraîne pas le déplacement physique des données.

La valeur par défaut d'une référence est `null`, cela signifie que cette référence ne correspond à aucune adresse mémoire. Si on tente d'accéder au contenu de cette adresse, cela provoque une erreur d'exécution de type « `Java.lang.NullPointerException` ».

Remarque

Lorsque l'on appelle la variable `ta` tableau, on effectue un abus de langage, on devrait l'appeler « référence de tableau ». On admet cet abus de langage

Le programme suivant montre la manière dont les tableaux sont traités. Lors de l'affichage de la référence d'un tableau, la méthode `println` affiche la concaténation « `[I` » qui doit être interprétée comme tableau `d'int` et l'adresse du contenu du tableau « `@3e25a5` ». On peut observer que la comparaison donne bien un résultat faux, en effet, ce ne sont pas les contenus qui sont comparés mais bien les valeurs des références (adresses), `[I@19821f≠[I@19821f`. La figure (Figure 5) suivante donne la représentation symbolique de la configuration mémoire des variables (`ta`, `tb` et `tc`) et leur évolution, lorsqu'une variable, ici `tb`, vaut `null`, cette valeur est indiquée par une simple croix. Après l'affectation `tc = ta`, les deux variables désignent le même emplacement mémoire, l'emplacement qui n'est plus référencé est perdu (ici, celui qui l'était par `tc`). Désormais, les deux variables `ta` et `tc` désigne la même zone mémoire, toute modification de l'un des

éléments de `ta` est identique à la même modification opérée sur `tc` (on dit que `ta` et `tc` sont des alias ou des synonymes).

Exemple :

```
public class Test {
 static public void main(String []args) {
  int [] ta =  {1,2,3};
  int [] tb = null;
  int [] tc =  {1,2,3};
  System.out.println(ta);
  System.out.println(tb);
  System.out.println(tc);
  System.out.println(tc == ta);
  tc = ta;
  ta[0] = 5;
  System.out.println(tc[0]);
 }
}
```

Résultat de l'exécution :

```
[I@3e25a5
null
[I@19821f
false
5
```

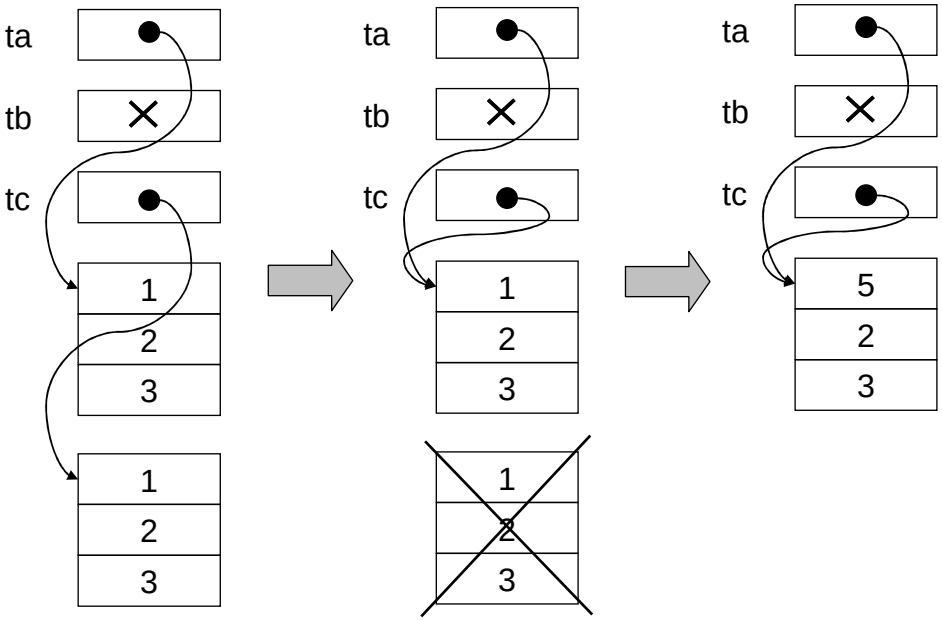


Figure 5 : Références de plusieurs tableaux

4.3 Opérateur new

Java permet de créer dynamiquement des tableaux (et des objets), cette caractéristique va permettre en cours d'exécution de spécifier la taille des tableaux en fonction des besoins du programme, on utilise l'opérateur `new` pour créer dynamiquement des tableaux ou des objets, cet opérateur retourne la référence du tableau créé.

Dans la partie précédente, les tableaux étaient dimensionnés à la déclaration lors de leur initialisation. On peut donc séparer la déclaration, la création (ou allocation) et l'initialisation d'un tableau.

Le programme suivant décompose ces trois étapes pour un tableau d'entiers de type `int`.

Déclaration d'un tableau[6] (d'entiers de type `int`) :

`int [] t; ⇔ int t[];`

Allocation dynamique ou **création** en utilisant l'opérateur `new` :

`t = new int[3];`

On effectue l'allocation d'un tableau de trois entiers de type `int`. Le nombre d'éléments ou taille du tableau est une constante entière ou plus généralement une expression de type entier dont l'évaluation doit pouvoir être effectuée à l'exécution, cette valeur est « passée » à l'opérateur `new` qui effectue la réservation mémoire et retourne la référence du tableau créé. Une fois qu'un tableau est créé avec une taille donnée, celle-ci ne peut plus être modifiée. L'opérateur `new` permet aussi d'instancier une classe. Cet opérateur est similaire à la fonction malloc, cependant, il effectue tout seul le calcul de la taille (en faisant l'analogie avec le langage C, `new int[3] ⇔ malloc(3*sizeof(int))`).

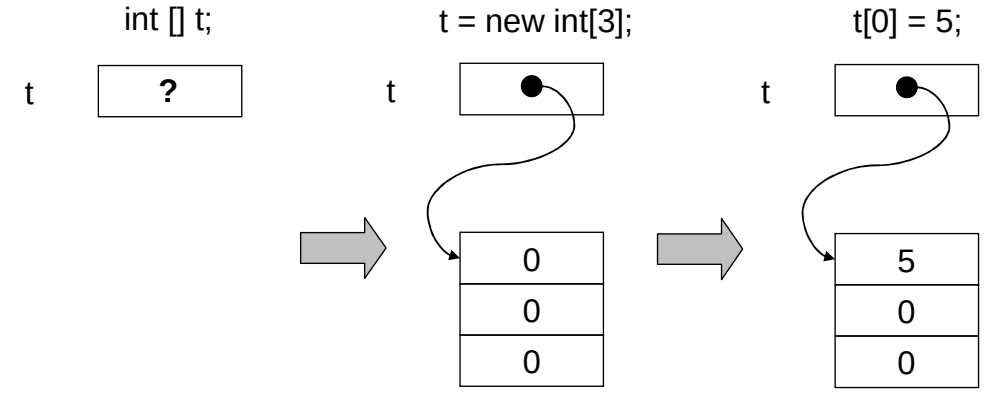
A la création, les éléments d'un tableau sont initialisées avec une valeur par défaut qui dépend du type. Cette valeur est `0` pour les types primitifs numériques, `false` pour le type booléen, et `null` pour tout champ objet ou tableau :

Initialisation des éléments d'un tableau

Si on ne désire pas utiliser les valeurs d'initialisation par défaut, on peut utiliser l'opérateur d'accès ou d'indice « `[]` », par exemple, `t[0] = 5` ;

Exemple :

```
public class Test {
 static public void main(String []args) {
  int [] t;
  t = new int[3];
  t[0] = 5;
 }
}
```



[6] On devrait plutôt dire que t est la référence d'un tableau

Figure 6 : Evolution du tableau t

On peut aussi utiliser cet opérateur pour créer un tableau et l'initialiser avec des valeurs distinctes des valeurs par défaut, après la déclaration afin d'effectuer une initialisation différée dans ce cas là, il ne faut pas donner la dimension du tableau. C'est ce qui est fait dans la séquence suivante, dans laquelle `t` référence un tableau de `3` entiers de type `int` dont les composantes sont initialisées respectivement à `3,4,5` :

```
int [] t;
t = new int[] {3,4,5};
```

Il est à noter que comme en langage C, une liste d'initialisation ne peut être utilisée qu'à la déclaration pour initialiser un tableau, la séquence suivante est illicite (erreur de compilation « `illegal start of expression` ») :

```
int [] t;
t = {3,4,5}; // erreur de compilation
```

Mode	Exemple
Déclaration, création et initialisation	`int [] ta = {1,2,3};`
Déclaration, puis création et initialisation	`int [] t;` `t = new int[] {3,4,5};`

Déclaration, puis création puis initialisation	`int [] t;`
	`t = new int[3];`
	`t[0] = 1 ;`
	`t[1] = 2 ;`
	`t[2] = 3 ;`

Tableau 3 : Synthèse des modes de déclaration – initialisations de tableau

On peut aussi créer grâce à l'opérateur `new` des tableaux anonymes, ces tableaux pouvant être passés à des méthodes, nous verrons cette possibilité à la fin du paragraphe§6.2 p39.

4.4 Garbage collector

4.5 Donnée length

La valeur du champ `length` est un champ entier automatiquement associée à un tableau dès sa création (pas à la déclaration) qui contient la dimension du tableau[7] référencé. Cette valeur est accessible comme un champ en utilisant le sélecteur de champ « `.` ». Il faut privilégier ce moyen d'accéder à la dimension d'un tableau car `length` est toujours à jour.

L'exemple suivant montre l'utilisation du champ `length` qui permet, après la création des deux références de tableaux affectées à `t`, d'en connaître la dimension, l'accès s'effectue simplement en écrivant `t.length` qui contient bien la dimension du tableau (référencé par) `t`.

Exemple :

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

[7] En fait, Java traite les tableaux comme des objets et length est une variable d'instance d'un objet de type tableau

```
   int [] t = {1,2,3,4};
   System.out.println(t.length);
   t = new int[10];
   System.out.println(t.length);
  }
 }
```

Résultat de l'exécution :

```
4
10
```

4.6 Tableau à plusieurs dimensions

Les tableaux à deux dimensions sont définis en Java comme des tableaux de tableaux. Chaque composante du tableau est elle-même un tableau à une dimension. Par exemple, les matrices sont des tableaux réguliers à deux dimensions, c'est-à-dire que toutes les lignes font la même taille.

Comme pour les tableaux à une dimension, il est possible de définir les tableaux multidimensionnels de plusieurs manières. L'exemple suivant exploite ces possibilités. On crée successivement trois tableaux d'entiers de type `int` qui possèdent le même contenu, mais des références différentes. Chacun de ces tableaux correspond à la matrice d'entiers 3×2 suivante :

$$\begin{pmatrix} 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{pmatrix}$$

Le tableau `ta` est directement déclaré et complètement (lignes + colonnes) créé puis initialisé, le tableau `tb` est déclaré, créé et initialisé (liste d'initialisation), le tableau `tc` est déclaré et partiellement créé, puis chacune des lignes est créée et enfin chaque composante initialisée.

Exemple :

```
public class Test {
 static public void main(String []args) {
  int [][]ta = new int [3][2];
  ta[0][0] = ta[0][1] =1;
  ta[1][0] = ta[1][1] =2;
  ta[2][0] = ta[2][1] =3;
  int [][]tb = {{1, 1}, {2, 2}, {3, 3}};
  int [][]tc = new int [3][];
  tc[0] = new int [2];
  tc[1] = new int [2];
  tc[2] = new int [2];
  tc[0][0] = tc[0][1] =1;
  tc[1][0] = tc[1][1] =2;
  tc[2][0] = tc[2][1] =3;
 }
}
```

La figure suivante (Figure 7) donne la représentation symbolique de la configuration mémoire de la variable `ta` après l'initialisation complète du tableau qu'elle référence (les

configurations associées à `tb` et `tc` sont identiques). Ici, `ta` a pour valeur la référence du tableau de dimension trois dont chaque composante a pour valeur la référence d'un tableau de deux `int`.

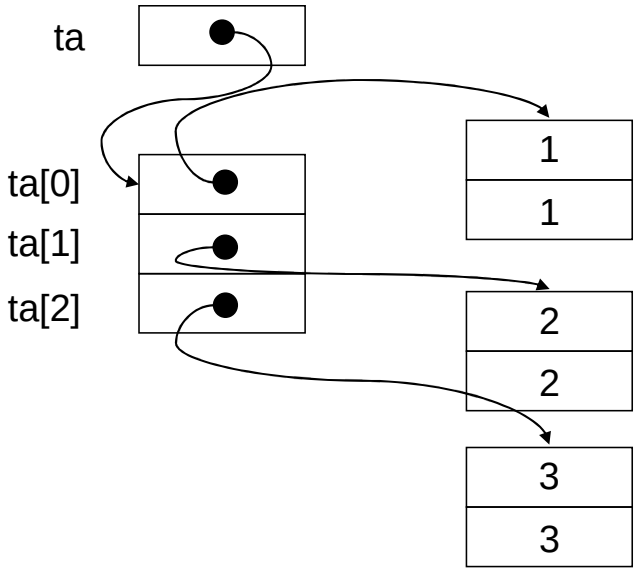


Figure 7 : Représentation des tableaux à deux dimensions

On aurait pu séparer la déclaration de la création :

```
int [][]ta;
ta = new int [3][2];
ta[0][0] = ta[0][1] =1;
...
int [][]tb ;
tb = new int [3][];
tb[0] =   new int []{1, 1};
...
int [][]tc;
tc = new int [3][];
```

Un des avantages de définir les tableaux à deux dimensions comme des tableaux de tableaux, c'est qu'il devient possible de créer des tableaux irréguliers. Un tableau est dit irrégulier si toutes les lignes n'ont pas le même nombre d'éléments.

Le programme suivant illustre l'utilisation de tableaux irréguliers. Le tableau `t` est un tableau de `3` lignes, la première ligne est constituée de `1` élément, la seconde de `2` élément et la troisième de `3` éléments. On note l'utilisation du champ `length` dans le cas d'un tableau à deux dimensions ; pour connaître le nombre d'éléments du tableau, il suffit d'associer ce champ au nom du tableau (la variable qui référence le tableau), ici `t.length`, et pour connaître la dimension d'une ligne, il suffit de l'associer à la composante correspondante, par exemple `t[1].length`.

Exemple :

```
public class Test {
 static public void main(String []args) {
  int [][]t;
  t = new int[3][];
  t[0] = new int[] {1};
  t[1] = new int[] {2,2};
```

```
   t[2] = new int[] {3,3,3};
   System.out.println(t.length);
   System.out.println(t[0].length);
   System.out.println(t[1].length);
   System.out.println(t[2].length);
  }
 }
```

Résultat de l'exécution :

```
3
1
2
3
```

5. Instructions

5.1 Introduction

Les instructions ou structures de contrôle permettent de contrôler le déroulement d'un programme. Il y a quatre types d'instructions : les instructions d'itération pour répéter des instructions, les instructions de sélection qui indiquent au programme les choix des instructions à exécuter, et les instructions de branchement ou de rupture qui passent le contrôle à une autre partie du programme.

5.2 Boucles

La boucle **`while`** (tant que faire) permet d'exécuter un bloc d'instructions tant qu'une condition particulière, appelée condition de continuation, est satisfaite (c'est-à-dire vraie). Si cette condition est fausse juste avant de rentrer dans le `while`, le bloc associé n'est pas exécuté et on passe à la première instruction qui suit ce bloc (la condition doit être obligatoirement de type booléen).

Dans l'exemple suivant, tant que la valeur de la variable `i` (variable de boucle) est inférieure ou égale à `10`, le bloc associé à la boucle `while` est exécuté. Cette extrait de code calcule la somme des entiers compris entre `0` et `10`.

Exemple :

```
int s = 0;
int i = 0;
while(i <= 10) {
 s += i;
 i++;
}
```

La boucle **`do while`** (faire jusqu'à) permet d'exécuter un bloc d'instructions tant qu'une condition particulière est satisfaite (condition de continuation), à l'opposé du `while` le bloc associé est exécuté au moins une fois (le test étant effectué après). Dans l'exemple suivant, le bloc associé à la boucle `do while` est exécuté une fois, puis le bloc est exécuté jusqu'à ce que la valeur de `i` soit supérieure à `10`. Cet extrait de code calcule la somme des entiers compris entre `0` et `10`. Si la variable `i` avait été initialisée avec la valeur `11`, le bloc n'aurait été exécuté qu'une fois (à l'inverse d'une boucle `while`).

Exemple :

```
int s = 0;
```

```
int i = 0;
do {
 s += i;
 i++;
} while (i <= 10);
```

La boucle **for** (pour) est une boucle `while` enrichie. Elle est constituée de trois parties : une expression d'initialisation, une expression conditionnelle booléenne de continuation et une expression d'itération. L'expression d'initialisation (premier terme) est exécutée une unique fois, le bloc d'instruction associé est exécuté tant que la condition de continuation (second terme) est vraie, l'expression d'itération (dernier terme) est exécutée après l'exécution du bloc d'instruction associé. Toute boucle `for` peut être transformée en boucle `while` équivalente.

```
for(exp_initialisation; exp_continuation;
exp_itération) {
   bloc d'instruction ;
}
```

```
exp_initialisation ;
while(exp_continuation)
{
   bloc d'instruction;
   exp_itération;
}
```

Tableau 4 : Equivalence boucle for et while

Dans l'exemple suivant, la variable `i` est initialisée à `0`, tant que la valeur de la variable `i` (variable de boucle) est inférieure ou égale à `10`, le bloc associé à la boucle est exécuté, puis la variable `i` est incrémentée. Cet extrait de code calcule la somme des entiers compris entre `0` et `10`.

Exemple :

```
int s = 0;
int i;
for (i = 0; i <=10; i++) {
 s+=i;
}
```

On peut directement déclarer la ou les variables de boucle dans l'expression d'initialisation, dans ce cas, la portée de la variable se limite à la boucle, la variable n'est plus reconnue à l'extérieur de la boucle. On peut noter que, dans l'exemple suivant, le bloc associé n'étant constitué que d'une unique instruction les accolades sont optionnelles.

Exemple :

```
int s = 0;
for (int i = 0; i <=10; i++) {
 s+=i;
}
```

On peut noter que l'exemple précédent peut être réécrit de manière plus condensée en exploitant l'opérateur de séparation d'expressions.

Exemple version condensée :

```
for (int s = 0,i = 0; i <=10; i++, s+=i);
```

La boucle « for each » qui existe depuis le JDK1.5[8] porte sur une structure de données « itérable » qui peut être un tableau ou une énumération (ou de manière plus générale une instance de classe implémentant l'interface `Iterable`) permet de parcourir et d'extraire simplement les valeurs des éléments contenus dans cette structure.

Dans l'exemple suivant, la variable `a` va prendre successivement toutes les valeurs du tableau `t`. Cette structure est une simplification d'écriture qui ne nécessite pas l'utilisation d'une variable de boucle (voir Tableau 5). La déclaration de `a` doit être obligatoirement avant les « : »

Exemple :

```
public class Test {
 static public void main(String []args) {
  int [] t = {1,3,5,7};
  for(int a : t)
   System.out.println(a);
 }
}
```

Résultat de l'exécution :

```
1
3
5
7
```

`for(int a : t)`
`  System.out.println(a);`
	`for(int a, i= 0; i < t.length; i++){`
	`  a = t[i];`
	`  System.out.println(a);`
	`}`

Tableau 5 : Equivalence boucle for et for each

Dans le cas de la boucle « for each », la variable `a` ne peut pas être déclarée en amont de la boucle. L'écriture suivante provoque l'erreur de compilation « `not a statement` ».

```
…
int a;
for(a : t)
…
```

5.3 Instructions de sélection

Les instructions de sélection (ou conditionnelle ou de choix) permettent de n'exécuter un ensemble d'instructions (bloc) que si une condition est satisfaite ou de choisir une alternative entre plusieurs options en fonction d'une condition (la condition doit être obligatoirement de type booléen).

Le programme suivant illustre l'utilisation de l'instruction de sélection **`if else`** (le `else` est optionnel). Si la valeur de `a` est supérieure à celle de `b` (`a > b` vaut alors `true`), on affecte à `c` la valeur de `a` et on affecte à `test false`, sinon on affecte à `c` la valeur de

[8] Cette boucle a été définie pour faciliter l'exploitation des collections.

b. On peut noter que lorsque l'on veut associer plusieurs instructions, les accolades sont obligatoires (bloc après le `if`). Dans ce premier cas, on a bien une alternative. Le second `if` n'offre pas d'alternative (pas de `else`). Si `test` vaut `true` on affiche `a > b`, si ce n'est pas le cas, le programme reprend son exécution après le bloc du `if`.

Exemple :

```
public class Test {
 static public void main(String []args) {
  int a, b, c;
  boolean test = false;
  a = 3;
  b = 2;
  if(a > b){
   c = a;
   test = true;
  }
  else
   c = b;
  if(test) {
   System.out.println("a > b");
  }
  System.out.println(c);
 }
}
```

Résultat de l'exécution :

```
a > b
3
```

Il est aussi possible d'effectuer une sélection entre plusieurs alternatives en utilisant une structure **`if - else if – else`**, le nombre de `else if` pouvant être quelconque. Le programme suivant illustre l'utilisation de cette structure de contrôle. Si les valeurs de `a` et `b` sont négatives, on affecte à `c` la valeur `0`, sinon, si la valeur de `a` est supérieure à celle de `b`, on affecte à `c` la valeur de `a`, sinon on affecte à `c` la valeur de `b`.

Exemple :

```
public class Test {
 static public void main(String []args) {
  int a, b, c;
  a = 3;
  b = 2;
  if(a < 0 && b < 0)
   c = 0;
  else if (a > b)
   c = a;
  else
   c = b;
  System.out.println(c);
 }
}
```

Résultat de l'exécution :

```
3
```

Il existe une dernière instruction de sélection, dite à choix multiple ou de branchement ou d'aiguillage qui permet de sélectionner un ensemble d'instructions à exécuter en fonction de la valeur d'une expression, il s'agit du **`switch case`**. Dans ce cas, l'expression (entière : `byte`, `short`, `int` et `char`) est comparée successivement à plusieurs constantes, dès qu'une des constantes est égale à l'expression, l'exécution reprend juste après le `case` correspondant (on dit qu'il y a branchement), puis toutes les instructions jusqu'à la fin du `switch` sont exécutées ou jusqu'à l'exécution d'une instruction de rupture. Si aucune constante n'est égale, le branchement s'effectue au niveau du `default` (optionnel) et s'il n'y a pas de `default`, l'exécution reprend après le `switch`. Tout ce qui se trouve entre deux étiquettes est considéré comme un bloc.

Le programme suivant teste la valeur de la variable `j`, si celle-ci est comprise entre `1` et `5`, il affiche `On travaille`, ou si celle-ci est comprise entre `6` et `7` il affiche `On se repose`, sinon il affiche `Jour inconnu`. Ce dernier cas correspond au traitement d'une valeur inattendue, l'instruction `default` est donc utile pour traiter les cas imprévus. On constate l'utilisation du `break` uniquement au niveau des cas `5` et `7`, afin d'éviter de dupliquer les instructions d'affichage. Si, il n'y avait pas eu de `break` le programme suivant aurait affiché :

```
On travaille
On se repose
Jour inconnu
```

Exemple :

```
public class Test {
 static public void main(String []args) {
  char j = 2;
  switch (j) {
   case 1 :;
   case 2 :;
   case 3 :;
   case 4 :;
   case 5 : System.out.println("On travaille");
     break;
   case 6 :;
   case 7 : System.out.println("On se repose");
     break;
   default : System.out.println("Jour inconnu");
  }
 }
}
```

Résultat de l'exécution :

```
On travaille
```

5.4 Instructions de rupture

Il y a trois instructions de rupture qui sont `return`, `break` et `continue`. Ces instructions arrêtent l'exécution dans un bloc et reprennent celle-ci ailleurs. L'instruction

`return` est utilisée dans les méthodes (fonctions), elle sort de la méthode et permet de retourner une valeur (voir §6 p37).

L'instruction **`break`** permet d'interrompre une suite d'instructions dans une boucle ou un `switch` pour passer à l'instruction qui suit la boucle ou le `switch` dans le programme.

Le programme suivant a pour objet d'afficher l'indice du premier élément d'un tableau dont la valeur est supérieure à une valeur donnée. Le tableau `t` est parcouru dans la boucle `for`, à chaque itération, la valeur du ième élément du tableau est comparée avec la valeur de la variable `v`, si cet élément est plus grand, on sort de la boucle et on affiche l'indice de cet élément.

Programme :

```
public class Test {
 static public void main(String []args) {
  int [] t = {1,3,5,7,9};
  int i, v = 6;
  for(i = 0; i < t.length; i++){
   if(t[i] > v) break;
  }
  System.out.println(i);
 }
}
```

Résultat de l'exécution :

```
3
```

L'instruction **`continue`** a un comportement similaire à celui de `break`, mais redonne le contrôle à l'itération suivante de la boucle au lieu d'en sortir. Cette instruction ne peut être associée qu'à une boucle.

Le programme suivant a pour objet de faire la somme des premiers entiers pairs. A chaque itération, si `i` est impair (test de parité), les instructions qui suivent ne sont pas exécutées et la boucle est relancée après que le troisième terme de l'entête de boucle ait été exécuté (`i++`).

Exemple :

```
public class Test {
 static public void main(String []args) {
  int i, s = 0;
  for(i = 0; i < 20; i++){
   if((i & 1) == 1) continue;
   s += i;
  }
  System.out.println(s);
 }
}
```

Résultat de l'exécution :

```
90
```

Il est aussi possible d'étiqueter les **`break`** et les **`continue`**, si on désire que la rupture ne se fasse pas sur la boucle courante, mais sur une boucle englobante. L'étiquette, un

identificateur Java suivi de « : », doit obligatoirement se trouver devant l'entête de la boucle sur laquelle la rupture doit s'effectuer.

Le programme suivant stocke dans un tableau (`t`) l'indice de la première composante de chaque ligne d'une matrice (`m`) d'entier égale à une valeur donnée (`v`). Dans la seconde boucle `for` dès que l'on vérifie l'égalité (`m[i][j] == v`) alors, on stocke le numéro de colonne correspondant dans `t[i]`, puis on exécute le `continue` (les autres composantes ne seront pas inutilement testées), et la boucle englobante (sur `i`) est relancée après que le troisième terme de l'entête de boucle ait été exécuté (`i++`).

Exemple :

```
public class Test {
 static public void main(String []args) {
  int [][] m = {{1,2,5, 8, 2},{1,5,3,4}, {5,2,5,1},{1,2,3}};
  int [] t = {-1,-1,-1,-1};
  int v = 5;
  debut :
  for(int i = 0; i < m.length; i ++){
   for(int j = 0; j < m[i].length; j++){
    if(m[i][j] == v){
     t[i] = j;
     continue debut;
    }
   }
  }
  for(int i = 0; i < m.length; i ++)
   System.out.println(t[i]);
 }
}
```

Résultat de l'exécution :

```
2
1
0
-1
```

Le programme suivant affiche les indices de la première composante d'une matrice (`m`) d'entiers égale à une valeur donnée (`v`). Dans la seconde boucle `for` dès que l'on vérifie l'égalité (`m[i][j] == v`) alors, on exécute le `break` qui provoque la sortie de la boucle englobante (sur `i`), ainsi les valeurs de `i` et de `j` qui sont sauvegardées correspondent aux indices de l'élément recherché.

Exemple :

```
public class Test {
 static public void main(String []args) {
  int [][] m = {{1,2,3}, {1,2,5,8,2}, {1,5,4,7,3}, {5,2,5,1}};
  int v = 7;
  int i = 0, j = 0;
  debut :
  for(i = 0; i < m.length; i ++){
   for(j = 0; j < m[i].length; j++){
```

```
     if(m[i][j] == v){
      break debut;
     }
    }
   }
   System.out.println(i);
   System.out.println(j);
  }
 }
```

Résultat de l'exécution :

```
2
3
```

Remarque importante

Pour l'écriture de fragments de codes assez courts, la programmation structurée recommande une organisation hiérarchique simple du code. On peut le faire dans la plupart des langages de programmation modernes par l'utilisation de structures de contrôles `while`, `do while`, `for`, `if else`. Il est également recommandé de n'avoir qu'un point d'entrée pour chaque boucle (et un point de sortie unique dans la programmation structurée originelle), et quelques langages l'imposent. Le langage Java, bien qu'ayant abandonné le `goto` présent en langage C et langage C++, permet l'utilisation des instructions de rupture dans les boucles, en particulier. Nous recommandons donc de limiter l'utilisation de ces instructions qui réduisent la structuration des programmes.

6. Méthodes

6.1 Introduction

En Java, les méthodes sont semblables à des fonctions ou des procédures dans d'autres langages de programmation. Les méthodes vont permettre de définir des traitements en vue de leur réutilisation. Ecrire des méthodes remplit plusieurs rôles : au-delà de la possibilité de réutilisation des fonctions à différents endroits du programme, le plus important est peut-être de clarifier la structure du programme, pour le rendre lisible et compréhensible par d'autres personnes que le programmeur original, et par conséquent permet de rendre la maintenance du programme plus facile. Une bonne structuration d'un programme en méthodes aident donc à créer des petits morceaux de code, faciles à comprendre isolément sans avoir à étudier l'ensemble du programme, c'est un des paradigmes de la programmation structurée.

Les méthodes peuvent recevoir des données à traiter et peuvent retourner des données ou encore modifier les données qu'on leur a transmises.

Une méthode, en Java, appartient forcément à une classe. Une méthode possède un droit d'accès et d'autres modificateurs sur lesquels nous reviendrons dans la suite. Pour le moment, nous nous limitons à des méthodes regroupées dans la classe possédant la méthode principale `main`.

La forme générale d'une méthode est :

```
[modificateurs] typeRes nomMéthode(type1 nom1, ..., typek nomk) {
 Bloc d'instructions
 [return r ; ]
```

```
}
```

Dans cette écriture, `typeRes` est le type du résultat retourné par la méthode, ce type pouvant correspondre à un type primitif, à une classe, un tableau, une énumération ou `void` dans le cas d'une méthode qui ne retourne rien.

Chaque couple de la liste `type1 nom1`, ..., `typek nomk` correspond à la déclaration d'une variable locale à la fonction appelée paramètres. Chacune de ces variables à un type pouvant correspondre à un type primitif, à une classe, un tableau ou une énumération. Ce sont ces paramètres qui vont recevoir les données à traiter.

Le type d'un paramètre peut être précédé par le mot clé `final`, dans ce cas la valeur du paramètre ne peut être modifiée (comme une variable locale constante) dans la méthode.

En Java, toute méthode dont le type de retour est différent de `void` doit au moins posséder un `return` avant l'accolade fermante suivie d'une expression d'un type compatible avec le type du résultat retourné. Lorsque le `return` est exécuté, on sort de la méthode, la valeur de l'expression est retournée à la partie appelante. Une méthode dont le type de retour est `void`, ne possède pas obligatoirement de `return` avant l'accolade de fin de méthode, dans ce cas, le compilateur en synthétise un (`return` implicite).

L'appel d'une méthode s'écrit de la façon suivante :

```
nomMéthode(val1, val2, ... , valk)
```

la liste de valeurs `val1`, `val2`, ... , `valk` constituent les arguments passés à la méthode (généralement des expressions), les valeurs de ces arguments doivent être de types compatibles avec les paramètres correspondants de la méthode. Au début de l'exécution de la méthode, à chaque paramètre va être affecté la valeur de l'argument correspondant.

Les conventions de nommage des méthodes sont les mêmes que celles des variables locales (§ 2.3 p15).

On peut noter qu'en Java, il n'y a pas de déclaration de méthodes (prototypes), comme en langage C ou C++.

L'exemple suivant illustre les différents cas de figure. La première méthode `max` possède deux paramètres de type `int` et retourne un `int` (la valeur du plus grand des arguments passés). Au moment de l'appel, la valeur de la variable `n` est copiée dans le paramètre `a` et celle de `m` dans `b`, la valeur retournée est recopiée dans `p`. La seconde méthode `sommeVect` retourne la somme des composantes d'un tableau d'`int`, lors de l'appel, c'est la référence de la première ligne de `ma` qui est recopiée dans `tb` (qui est aussi une référence). La méthode `sommeMat` retourne un vecteur dont chacune des composantes est la somme des composantes de la ligne d'une matrice dont la référence est passée en argument. Lors de l'appel, c'est la référence `ma` qui est recopiée dans `mat`, le vecteur résultat (référencé par) `v` est crée et mis à jour dans `sommeMat`, c'est (la référence) `v` qui est retournée et sa valeur est recopiée dans `va`. On peut observer que `sommeMat` appelle `sommeVect`. La dernière méthode `printVect` de type `void` reçoit comme argument un tableau d'`int` qu'elle affiche élément par élément. Dans ce cas, nous avons omis le `return` avant l'accolade fermante car il est optionnel pour une méthode de type `void`.

Exemple :

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

```
   int n = 5, m = 4, p;
   int [][] ma = {{1,2,5,8,2},{1,5,3,4},{5,2,5,1},{1,2,3}};
   p = max(n, m);
   System.out.println(p);
   int [] va;
   p = sommeVect(ma[0]);
   System.out.println(p);
   va = sommeMat(ma);
   printVect(va);
  }
  static int max(int a, int b){
   if(a > b) return a;
   return b;
  }
  static int sommeVect(int [] tb){
   int s = 0;
   for(int i = 0; i < tb.length; i++) {
    s += tb[i];
   }
   return s;
  }
  static int [] sommeMat(int [][] mat){
   int [] v = new int[mat.length];
   for(int i = 0; i < v.length; i++)
    v[i] = sommeVect(mat[i]);
   return v;
  }
  static void printVect(int [] tb){
   for(int i = 0; i < tb.length; i++)
    System.out.println(tb[i]);
  }
 }
```

Résultat de l'exécution :

```
5
18
18
13
13
6
```

6.2 Passation de paramètres

La passation de paramètres en Java s'effectue par valeur, c'est-à-dire que la valeur de chaque argument passé à une méthode est recopiée dans le paramètre correspondant. Ainsi, lorsque l'on passe une variable en argument à une méthode, la valeur de cette variable reste inchangée après l'appel. Les paramètres étant des variables locales de la méthode, ils disparaissent après l'appel.

Le programme suivant illustre la passation de paramètre par valeur, des affichages sont ajoutés afin de visualiser l'évolution des données. Au moment de l'appel, les valeurs des variables `n` et `m` sont respectivement recopiées dans les variables `a` et `b`, dans la

méthode échange, les valeurs des variables `a` et de `b` sont échangées (on utilise une variable locale intermédiaire `t`). Lorsque l'on retourne dans `main` les valeurs des variables `m` et `n` sont inchangées dans la mesure où la méthode `echange` manipulait des copies des valeurs de ces variables. La figure suivante (Figure 8) représente l'évolution des variables durant l'exécution de ce programme.

Exemple :

```
public class Test {
 static public void main(String []args) {
  int n = 1, m = 2;
  System.out.println("n = "  + n + " m = "  +  m);
  echange(n, m);
  System.out.println("n = "  + n + " m = "  +  m);
 }
 static void echange(int a, int b){
  System.out.println("--------- début echange ---------");
  int t = a;
  System.out.println("a = "  + a + " b = "  +  b + " t = "  +  t);
  a = b;
  b = t;
  System.out.println("a = "  + a + " b = "  +  b + " t = "  +  t);
  System.out.println("--------- fin echange ---------");
  Return ; //optionnel
 }
}
```

Résultat de l'exécution :

```
n = 1 m = 2
--------- début echange ---------
a = 1 b = 2 t = 1
a = 2 b = 1 t = 1
--------- fin echange ---------
n = 1 m = 2
```

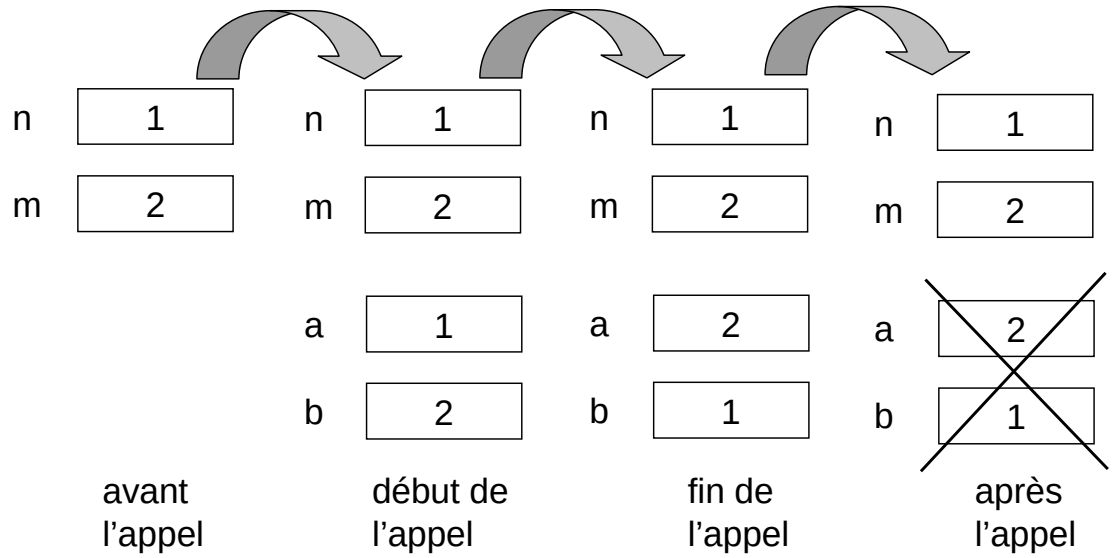


Figure 8 : Passation de paramètres

Si on désire modifier une donnée, le seul moyen est d'utiliser une référence, dans ce cas, c'est le contenu de cette référence qui sera modifié (c'est-à-dire l'emplacement mémoire désigné par cette référence).

Le programme suivant illustre la passation de paramètres par référence (qui est un cas particulier de passation de paramètres par valeur), des affichages sont ajoutés afin de visualiser l'évolution des données. Le tableau `tb` créé et initialisé dans le `main` est passé à la méthode `echange`, or, la valeur de `tb` est une référence, cette référence, lors de l'appel est recopiée dans ta, dès lors, `ta` et `tb` sont des synonymes qui désignent la même zone mémoire. Ainsi, toutes les modifications du tableau effectuées dans `echange` modifient aussi le tableau `tb`. On peut noter que `ta` et `tb` ont bien les mêmes valeurs, correspondant à la référence du tableau (`@3e25a5` adresse de la zone mémoire qui le contient).

Exemple :

```
public class Test {
 static public void main(String []args) {
  int [] tb = {1, 2};
  System.out.println("tb[0] = "  + tb[0] + " tb[1] = "  +  tb[1]);
  System.out.println(tb);
  echange(tb);
  System.out.println("tb[0] = "  + tb[0] + " tb[1] = "  +  tb[1]);
 }
 static void echange(int [] ta){
  System.out.println("--------- début echange ---------");
  System.out.println(ta);
  int t = ta[0];
  System.out.println("ta[0] = "+ta[0]+" ta[1] = "+ta[1]+" t = "+t);
  ta[0] = ta[1];
  ta[1] = t;
  System.out.println("ta[0] = "+ta[0]+" ta[1] = "+ta[1]+" t = "+t);
  System.out.println("--------- fin echange ---------");
 }
}
```

Résultat de l'exécution :

```
tb[0] = 1 tb[1] = 2
[I@3e25a5
--------- début echange ---------
[I@3e25a5
ta[0] = 1 ta[1] = 2 t = 1
ta[0] = 2 ta[1] = 1 t = 1
--------- fin echange ---------
tb[0] = 2 tb[1] = 1
```

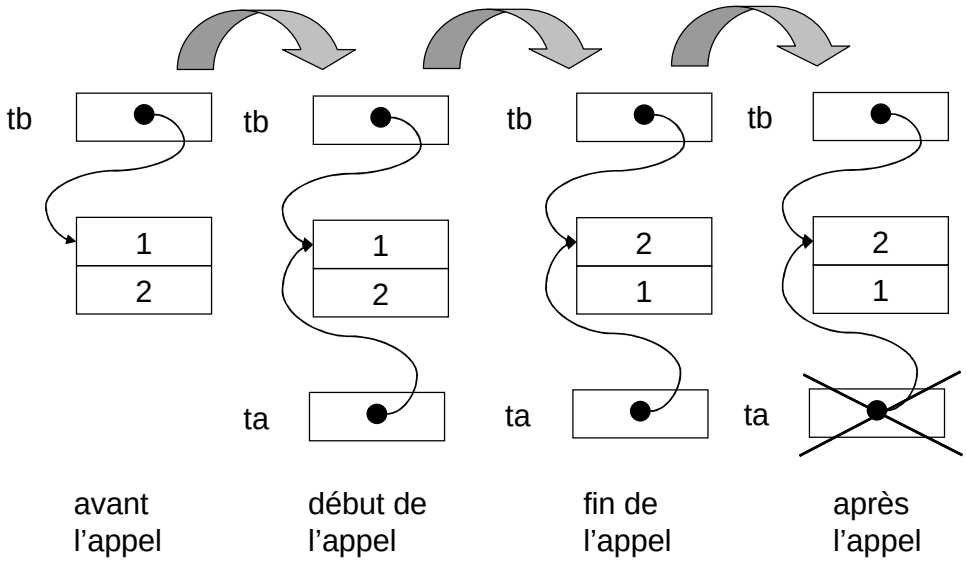


Figure 9 : Passation de paramètre avec référence

Comme nous l'avions évoqué, il est possible de créer grâce à l'opérateur `new` des tableaux anonymes, ces tableaux pouvant être passés à des méthodes. Ce mécanisme est aussi (et surtout) utilisé pour créer des objets anonymes.

Dans l'exemple suivant, on passe directement à la méthode `init` la référence du tableau créé dynamiquement par l'expression `new int[5]`, ce tableau est anonyme, car dans `main`, il n'est référencé par aucune variable lors de l'appel.

Exemple :

```
public class Test {
 static public void main(String []args) {
  printVect(init(new int[5]));
 }
 static int [] init(int [] ta){
  for(int i = 0; i < ta.length; i++)
   ta[i] = 2*i;
  return ta;
 }
 static void printVect(int [] tb){
  for(int i = 0; i < tb.length; i++)
   System.out.println(tb[i]);
 }
}
```

Résultat de l'exécution :

```
0
2
4
6
8
```

6.3 Récursivité

Une méthode récursive est une méthode qui s'appelle elle-même. Ce mode de programmation (récursivité) peut être utilisé pour définir des méthodes représentant un

problème défini par récurrence (suite, triangle de Pascal, …) ou parce qu'il s'adapte bien à des structures de données autoréférencées (listes, arbres, …). Par analogie avec les suites définies par récurrence, une méthode récursive possède généralement un test d'arrêt qui correspond à l'initialisation de la suite et le rappel de la fonction elle-même qui correspond au terme général de la suite. Le test d'arrêt doit toujours précéder le rappel de la méthode, sinon, la méthode se rappelle indéfiniment. Les méthodes récursives ont cependant un défaut, à chaque rappel de la méthode le contexte est sauvegardé (en particulier la valeur des variables locales), ces méthodes ont donc tendance à consommer beaucoup de mémoire.

Dans l'exemple suivant, la méthode récursive `facto` calcule le factoriel d'un nombre entier, par définition de la suite factorielle :

Terme initial : $u_0 = 1 \Rightarrow$ condition d'arrêt `if(n == 0) return 1;`

Terme général : $u_n = n \times u_{n-1} \Rightarrow$ calcul général de factoriel : `return n * facto(n-1);`

L'affichage des valeurs de `n` montre bien le comportement de la méthode, au moment de l'appel à partir de `main`, la valeur `4` est passée en argument, cette valeur étant différente de `0`, l'expression `n * facto(n-1)` est évaluée ce qui provoque le rappel de `facto`, la valeur `3` (`4-1`) est passée, cette valeur étant différente de `0`, l'expression `n * facto(n-1)` est évaluée, …, la valeur `0` (`1-1`) est passée, cette valeur étant égale de `0`, donc la valeur `1` est retournée au niveau de l'expression `n * facto(n-1);` avec `n = 1`, donc `1*1 = 1` est retournée, … , on revient finalement à la première évaluation de l'expression `n * facto(n-1)` pour `n = 4`, qui vaut `4*6 = 24` qui est retournée dans `main`.

Exemple :

```
public class Test {
 static public void main(String []args) {
  int r;
  r = facto(4);
  System.out.println("r = " + r);
 }
 static int facto(int n){
  System.out.println("n->" +n);
  if(n == 0) return 1;
  return n * facto(n-1);
 }
}
```

Résultat de l'exécution :

```
n->4
n->3
n->2
n->1
n->0
r = 24
```

Ce second exemple montre, l'utilisation d'une méthode récursive qui exploite un tableau. A chaque appel, avant que la condition d'arrêt soit vraie, on retourne la valeur de l'élément courant `t[n]` avec la somme de tous ses successeurs `sommeVect(t, n+1)`. Lorsque l'on arrive au dernier élément, la condition d'arrêt est vraie, la valeur du dernier élément est retournée. La seconde méthode, affiche dans l'ordre inverse les éléments du

tableau passé en argument, cette fois, au lieu de parcourir le tableau par indice croissant, il l'est par indice décroissant.

Exemple :

```
public class Test {
 static public void main(String []args) {
  int []ta = {1,2,5,8,2};
  int r;
  r = sommeVect(ta, 0);
  System.out.println("r = " + r);
  printVect(ta, ta.length - 1);
 }
 static int sommeVect(int []t, int n){
  if(n == (t.length - 1)) return t[n];
  return t[n] + sommeVect(t, n+1);
 }
 static void printVect(int []t, int n){
  if(n == -1) return;
  System.out.println(t[n]);
  printVect(t, n-1);
 }
}
```

Résultat de l'exécution :

```
r = 18
2
8
5
2
1
```

6.4 Surcharge de méthode

En Java, il est possible de donner le même nom à plusieurs méthodes, la décision de faire exécuter l'une plutôt que l'autre étant automatiquement prise par le compilateur en fonction du type des arguments, c'est un cas de *polymorphisme* (polymorphisme est la possibilité d'autoriser le même nom à être utilisé avec différents types). Cette caractéristique s'appelle la **surcharge** ou **surdéfinition** de méthodes (*overloading* en anglais). Ce concept, très utilisé dans les langages objets, n'est pas neuf lorsque l'on écrit `c = a + b`, le signe « + » est interprété différemment en fonction du type de `a` et `b` (si `a` et `b` sont des entiers, c'est une addition entière qui est exécutée, si `a` et `b` sont des réels, c'est une addition réelle).

Si deux méthodes possèdent le même identificateur, Java les différencie grâce à leurs signatures spécifiques.

- le nom,
- le type des paramètres,
- l'ordre des paramètres.

Le type de retour ne fait pas partie de la signature d'une méthode car lors de l'appel d'une méthode, rien ne suggère au compilateur le type de retour attendu (une valeur de retour de type `int` peut être affectée à un `int`, un `long`, un `double`, …), la valeur de

retour n'étant pas forcément exploitée. Pour chaque appel le compilateur sélectionne la méthode selon le type des arguments passés.

Par convention, on effectue une surcharge entre deux méthodes que si celles-ci ont la même sémantique pour des raisons évidentes de lisibilité.

Dans l'exemple suivant, deux méthodes effectuant la somme de deux variables et portant le même nom `somme` sont définies. Lors du premier appel, les arguments étant de type double, c'est la méthode `somme(double a, double b)` qui est appelée (paramètres de type `double`), Lors du second appel, les arguments étant de type `int`, c'est la méthode `somme(int a, int b)` qui est appelée (paramètres de type `int`).

Exemple :

```
static public void main(String []args) {
  int i = 2, j = 3;
  double x = 5.3, y = 3.3;
  System.out.println(somme(x, y));
  System.out.println(somme(i, j));
 }
 static int somme(int a, int b) {
  return a + b;
 }
 static double somme(double a, double b) {
  return a + b;
 }
```

Résultat de l'exécution :

```
8.6
5
```

Cependant il se peut qu'il n'existe pas lors d'un appel une méthode dont le type des arguments corresponde parfaitement à celui des paramètres. Le choix de la « bonne méthode » à appeler s'effectue de la manière suivante :

1. Le compilateur recherche d'abord une correspondance exacte (attention aux promotions automatiques).
2. Si aucune méthode n'a encore été sélectionnée, le compilateur effectue ensuite des promotions simples comme `char` → `int` et `float` → `double` et recherche une correspondance.
3. Si aucune méthode n'a encore été sélectionnée, il effectue des conversions `int` → `double` (il remonte la hiérarchie de types ou de classes) et recherche une correspondance, dès que l'on en trouve une, elle est sélectionnée, …
4. Sinon, c'est le message d'erreur (« `cannot find symbol method …` »).

On remarque que la hiérarchie de type de Java est exploitée (relation d'ordre stricte, non totale) :

- `byte` < `short` < `int` < `long` < `float` < `double`
- `char` < `int` (par transitivité < `long` < `float` < `double`)

Pour qu'une méthode soit sélectionnable pour un appel, il faut que chaque type d'argument soit « inférieur ou égal » au type de paramètre de la méthode. L'ensemble des règles permettant de modifier automatiquement et implicitement le type des arguments

afin de les adapter se nomme la coercition. Le polymorphisme en Java, basé sur la coercition et la surcharge, est appelée du **polymorphisme had hoc**.

Dans la suite, lorsque l'héritage sera défini, les hiérarchies de classes seront exploitées de façon similaire.

Dans l'exemple suivant, trois méthodes `somme` sont définies, une de type (`int`, `int`), une autre de type (`float`, `float`) et une de type (`short`, `short`). Pour faciliter la compréhension de l'exemple, un affichage correspondant à l'entête de la méthode est ajouté au début de chaque méthode. Etudions le comportement de ce programme, appel par appel :

- `somme(x, y)` : `x` et `y` sont de type `float`, il y a une méthode `somme` dont le type de chaque paramètre correspond exactement : `somme(float a, float b)` est appelée (on est dans le cas 1).
- `somme(i, j)` : `i` et `j` sont de type `int`, il y a une méthode `somme` dont le type de chaque paramètre correspond exactement : `somme(int a, int b)` est appelée (on est encore dans le cas 1).
- `somme(si + bi, si - bi)` : `si` et `bi` sont des petits entiers, lors des opérations de somme et de différence, ils sont promus en `int`, les arguments passés sont de type `int`, il y a une méthode `somme` dont le type de chaque paramètre corresponde exactement : `somme(int a, int b)` est appelée (on est encore dans le cas 1).
- `somme(li, lj)` : `li` et `lj` sont de type `long`, il n'y a aucune méthode `somme` dont le type de chaque paramètre corresponde parfaitement. Il faut donc en rechercher une en effectuant une conversion en `float` du premier argument (aucune méthode ne coïncide) ou du second argument (aucune méthode ne coïncide), puis des deux, les types correspondent : `somme(float a, float b)` est appelée (on est dans le cas 2).
- `somme(x, i)` : `i` est de type `int` et `x` est de type `float`, il n'y a aucune méthode `somme` dont le type de chaque paramètre corresponde parfaitement. Il faut donc en rechercher une en effectuant une conversion de `x` en `long` (aucune méthode ne coïncide), puis en effectuant une conversion de `x` en `float`, les types correspondent : `somme(float a, float b)` est appelée (on est dans le cas 3).

Exemple :

```
public class Test {
 static public void main(String []args) {
  int i = 2, j = 3;
  float x = 5.3f, y = 3.3f;
  long li = 5, lj = 3;
  byte bi = 4;
  short si = 4;
  System.out.println(somme(x, y));
  System.out.println(somme(i, j));
  System.out.println(somme(si + bi, si - bi));
  System.out.println(somme(li, lj));
  System.out.println(somme(x, i));
 }
 static int somme(int a, int b) {
```

```
   System.out.println("somme(int a, int b)");
   return a + b;
  }
  static float somme(float a, float b) {
   System.out.println("somme(float a, float b)");
   return a + b;
  }
  static short somme(short a, short b) {
   System.out.println("somme(short a, short b)");
   return (short)(a + b);
  }
 }
```

Résultat de l'exécution :

```
somme(float a, float b)
8.6
somme(int a, int b)
5
somme(int a, int b)
8
somme(float a, float b)
8.0
somme(float a, float b)
7.3
```

Si nous avions cherché à compiler la séquence :

```
double z = 7.0;
System.out.println(somme(x, z));
```

z étant de type double, donc plus haut que `float` dans la hiérarchie des types (`double` > `float`), aucune méthode ne peut convenir, ainsi, cette séquence aurait donné lieu à l'erreur de compilation : « `cannot find symbol method somme(float,double)` ».

On peut avoir des cas où plusieurs méthodes sélectionnables sont en concurrence. Si une de ces méthodes possèdent des paramètres qui sont tous plus bas dans la hiérarchie de types que les paramètres des autres méthodes candidates, elle est choisie. Dans l'exemple précédent, l'appel `somme(bi, bi)` aurait donné lieu à l'exécution de `somme(short a, short b)` et non de `somme(int a, int b)` car pour les deux arguments `short` < `int` donc (`short`, `short`) < (`int`, `int`).

Cependant, si le compilateur n'a pas la possibilité de sélectionner une unique méthode, il y a une ambiguïté qui engendre une erreur de compilation. Dans l'exemple suivant, les deux méthodes `somme(int a, short b)` et `somme(short a, int b)` conviennent pour l'appel `somme(si, sj)` ((`short`, `short`) < (`short`, `int`) et ((`short`, `short`) < (`int`, `short`)). Mais, les deux couples (`short`, `int`) et (`int`, `short`) sont incomparables. Il y a donc une erreur de compilation : « `reference to somme is ambiguous, both method somme(int,short) in Test and method somme(short,int) in Test match` ».

Exemple :

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

```
   short si = 4, sj = 3;
   System.out.println(somme(si, sj));
  }
  static int somme(int a, short b) {
   return a + b;
  }
  static float somme(short a, int b) {
   return a + b;
  }
 }
```

6.5 Méthode à nombre d'arguments variable

Depuis le JDK 5.0, en Java, comme en C, il est possible de définir des méthodes à nombre variable d'arguments en utilisant la notation « … », réservée uniquement au dernier argument. Dans ce cas, les valeurs de ces arguments sont recopiés dans un tableau et traités comme tels (autoboxing). L'exploitation est donc plus simple qu'en langage C. Une liste d'arguments variable est déclarée en séparant le dernier paramètre de son type par trois points « ... ».

```
typeRes nomMéthode(type1 nom1, ..., typev ... nomv) {
```

Dans l'exemple suivant, deux méthodes à nombre variable d'arguments sont définies. La première retourne la plus petite valeur de la liste d'arguments passée. Lors du premier appel, la liste (`5,2,4,5,1`) est copiée dans un tableau `{5,2,4,5,1}` dont la référence est passée à la méthode `min`. Puis cette référence est affectée à `argint` qui est implicitement de type référence de tableau d'`int` qui est traité par la méthode `min`. Concernant le troisième appel de `min`, comme on passe directement un tableau, on affecte à `argint` la référence de ce tableau. La méthode `cherche` teste que le premier argument se trouve dans la liste des arguments suivants (partie à nombre variable d'arguments), et retourne `true` si c'est le cas, `false` sinon, une boucle « for each » est utilisée afin de parcourir le tableau.

Exemple :

```
public class Test {
 static public void main(String [] args) {
  System.out.println(min(5,2,4,5,1));
  System.out.println(min(2));
  System.out.println(min(new int[] {9,5,7,8,3}));
  System.out.println(cherche(4,5,2,4,5,1));
  System.out.println(cherche(1,2));
  System.out.println(cherche(3,new int[] {9,5,7,8,3}));
 }
 public static int min(int...argint) {
  int i, imin = 0;
  for(i = imin + 1; i < argint.length; i++){
   if(argint[i] < argint[imin]) imin = i;
  }
  return argint[imin];
 }
 public static boolean cherche(int v, int...argint) {
  for(int n : argint){
   if(n == v) return true;
```

```
   }
   return false;
  }
```

Résultat de l'exécution :

```
1
2
3
true
false
true
```

Remarques :

Comme les nombres d'arguments variables sont traités comme des tableaux, il impossible de surcharger une méthode en substituant le paramètre de type nombre d'argument variables avec un tableau de même type de base.

```
public class Test {
…
 public static int min(int...argint) {
…
 public static int min(int []t) {
…
```

Provoque l'erreur de compilation « `min(int...) is already defined in Test` ».

6.6 Méthode main

La méthode `main` constitue la méthode principale de l'application, c'est une méthode particulière du programme appelée en premier lors du lancement de l'application. Cette méthode constitue donc le point d'entrée du programme[9], elle est unique pour une application Java. La méthode `main()` est déclarée en `public` et en `static` et se trouve obligatoirement dans la classe dont le nom correspond à celui de l'application.

Les parenthèses suivant le mot `main` permettent de définir les paramètres de la méthode, c'est-à-dire les données qu'elle reçoit en entrée. Dans toute application l'unique paramètre de la méthode `main` est de type `String []`, ce qui correspond à un tableau de chaînes de caractères. Les composantes de ce tableau vont recevoir comme valeur les références des arguments que le programme va lui-même recevoir sur la ligne de commande. Ces arguments sont très utiles pour définir des options ou des fichiers de données par exemple.

Le programme suivant permet d'afficher les arguments passés sur la ligne de commande. La valeur de `args[0]` correspond à la référence du premier argument, ..., la valeur de `args[args.length-1]` correspond à la référence du dernier argument.

Lors de la première exécution de l'application `Test`, `args[0]` référence la chaîne de caractères `Bonjour`, `args[1]` référence la chaîne de caractères `les`, `args[2]` référence la chaîne de caractères `amis`. Sur la ligne de commande l'espace joue le rôle de séparateur. Pour traiter plusieurs mots séparés par des espaces, comme une unique

[9] Les applets sont différentes des applications et n'ont pas de méthode main.

chaîne de caractères, il suffit de les mettre entre des ". Cette possibilité est illustrée par la seconde exécution de l'application `Test`.

Exemple :

```
public class Test {
 static public void main(String [] args) {
  for(int i = 0; i < args.length; i++)
   System.out.println(args[i]);
 }
}
```

Résultat des deux exécutions :

```
>Java Test Bonjour les amis
Bonjour
les
amis
>Java Test "Bonjour les amis"
Bonjour les amis
```

7. Exercices de synthèse

7.1 Calcul partiel d'une série

L'objectif de cet exercice est de calculer la somme $\sum_{1}^{N} 1/n^2$. cette somme est calculée dans la méthode `serie`, le calcul aurait pu être fait directement dans la méthode `main`. Sur le principe, il n'y a pas de difficulté, la méthode est la retranscription de la formule, à chaque itération `s` a pour valeur la somme au rang précédent et on lui ajoute le terme courant.

Il faut se méfier de deux difficultés. D'une part, la conversion explicite `(double)(i*i)` est ici nécessaire, sinon la formule du calcul du terme courant `1/(i*i)` aurait eu pour valeur `0` pour i > 1, en effet, `i` étant de type `int` comme la constante littérale `1`, on aurait eu une division entière, par exemple, au rang 2, 1 / 4 = 0. La conversion explicite donne lieu à une division réelle. D'autre part, les parenthèses sont nécessaires, sinon la règle d'associativité à gauche s'appliquent et `1/(double)i*i = (1/(double)i)*i`, le terme courant vaut donc toujours 1.

Solution possible :

```
public class Test {
 static public void main(String [] args) {
  System.out.println(serie(10));
 }
 static double serie(int n){
  double s = 0;
  for(int i = 1; i <= n; i++)
   s += 1/(double)(i*i);
  return s;
 }
}
```

Résultat de l'exécution :

```
1.5497677311665408
```

7.2 Suite de Syracuse

Le problème de Syracuse est le suivant. On définit la suite un par $u_0 > 1$ et

- $u_{n+1} = u_n / 2$ si un est pair;
- $u_{n+1} = 3u_n + 1$ sinon:

Normalement cette suite converge vers 1. Une boucle `while` est utilisée, la condition de fin correspondant à l'obtention de la valeur limite recherchée.

Solution possible :

```
public class Test {
 static public void main(String [] args) {
  System.out.println(syracuse(10));
 }
 static int syracuse(int u0){
  int n = 0, u = u0;
  while(u != 1){
   if((u%2) == 0) u = u/2;
   else u = 3*u +1;
   n ++;
  }
  return n;
 }
}
```

Résultat de l'exécution :

```
6
```

7.3 Suite de Newton

On rappelle que si f est une fonction à valeurs réelles définie et dérivable, et que f' ne s'annule pas sur l'intervalle d'étude, alors la suite :

- $u_{n+1} = u_n - f(u_n)/ f'(u_n)$

converge vers une racine de f à partir d'un « bon point de départ ». Par exemple, si on recherche une solution de $f(x) = x^2-a$ (a étant supposé positif), on a $f'(x) = 2x$ et on obtient directement $u_n = (u_{n-1} + a/u_{n-1})/2$ qui est donc sensée converger vers $\sqrt{a}$. Nous allons donc proposer deux méthodes, l'une itérative, l'autre récursive, afin de calculer la racine d'un réel positif par cette méthode.

Sur le principe, on itère le calcul des u_n, en partant de u_0, et on s'arrête quand la différence entre deux valeurs consécutives est plus petite qu'une valeur donnée correspondant à la précision de calcul désirée.

Les méthodes (récursive et itérative) ont donc trois arguments de type double, `u0` la valeur de la suite au rang 0, `a` la valeur dont on veut calculer la racine carrée et `eps` la valeur de la précision de calcul. Le principe de calcul consiste à calculer la nouvelle valeur de `un`, en fonction de l'ancienne. Afin de calculer la précision, il faut conserver pour le calcul l'ancienne valeur (`un_1`), que l'on stocke dans la variable `un_1`. Dans la version itérative, l'évaluation de `un` est faite dans une boucle `do while` et le test de continuation

dans le `while`, la méthode `Math.abs` est une méthode de bibliothèque qui retourne la valeur absolue de l'argument passé. La méthode récursive calcule à chaque rappel la nouvelle valeur de `un`, si la précision convient cette valeur est retournée, sinon on retourne la valeur retournée par le prochain appel de la méthode.

Solution possible :

```
public class Test {
 static public void main(String [] args) {
  System.out.println(newtonRacine(5, 4, 0.001));
  System.out.println(newtonRaciner(5, 4, 0.001));
 }
 static double newtonRacine(double u0, double a, double eps){
  double un = u0, un_1;
  do {
   un_1 = un;
   un = (un_1 + a/un_1)/2;
  }while(Math.abs(un - un_1)> eps);
  return un;
 }
 static double newtonRaciner(double un, double a, double eps){
  double un_1 = un;
  un = (un_1 + a/un_1)/2;
  if(Math.abs(un - un_1) <= eps) return un;
  return newtonRacine(un, a, eps);
 }
}
```

Résultat de l'exécution :

```
2.000000000006711
2.000000000006711
```

7.4 Affichage en base deux

C'est un cas où la solution récursive est plus élégante et plus facile à mettre en œuvre qu'une version itérative. Une solution itérative consisterait à rechercher la plus grosse puissance juste inférieure au nombre à afficher, à la soustraire et à recommencer avec le reste. La difficulté justement consiste à traiter le terme de plus forte puissance.

Ici pour la solution récursive retenue, il faut bien ordonner le rappel de la fonction et l'affichage.

Etudions, le premier appel de `afficherBinaire` dans `main` :

On appelle `afficherBinaire` (n = 13)

On appelle `afficherBinaire` (n = 6)

On appelle `afficherBinaire` (n = 3)

On appelle `afficherBinaire` (n = 1)

On appelle `afficherBinaire`(n = 0), qui ne fait rien. On revient à l'appel précédent :

n = 1, n%2 = 1 que l'on affiche. On revient à l'appel précédent :

n = 3, n%2 = 1 que l'on affiche. On revient à l'appel précédent :

n = 6, n%2 = 0 que l'on affiche. On revient à l'appel précédent :

n = 13, n%2 = 1 que l'on affiche et on sort définitivement de `afficherBinaire`.

Solution possible :

```
public class Test {
 static public void main(String [] args) {
  afficherBinaire(13);
  afficherBinaire(6);
  afficherBinaire(8);
 }
 static void afficherBinaire(int n){
  if(n == 0) {
   System.out.println();
   return;
  }
  afficherBinaire(n/2);
  System.out.print(n%2);
 }
}
```

Résultat de l'exécution :

```
1101
110
1000
```

On peut constater que si, on appelle la méthode avec `0` rien ne s'affiche, car la condition `n == 0` est fausse, pour éviter ce problème, on peut passer par une méthode intermédiaire uniquement destinée à traiter ce cas particulier qui aurait l'allure suivante (et remplacer dans main les appels de `afficherBinaire par afficherBinaireInter`) :

```
static void afficherBinaireInter(int n) {
  if(n==0)System.out.print("\n" + 0);
  else afficherBinaire(n);
 }
```

7.5 Calculatrice sur la ligne de commande

L'objectif de cet exercice est d'écrire un programme Java qui permet de faire une opération arithmétique simple sur deux nombres à partir de la ligne de commande et d'afficher le résultat. Le programme proposé récupère trois arguments sur la ligne de commande, si il n'y en a pas trois, il affiche un message d'erreur et arrête l'exécution (méthode `exit` de `System`). Si le nombre d'arguments est correct, il convertit en double (méthode `parseDouble` de `Double`) les deux arguments de type chaîne de caractères (`args[0]` et `args[2]`) correspondant aux nombres sur lesquelles l'opération doit être effectuées, l'argument correspondant à l'opération (`args[1]`) est converti en `char` (méthode `charAt` de `String`, qui extrait le premier caractère de la chaîne). Le calcul, à proprement parler, est effectué par la méthode `calcul`. Celle-ci sélectionne la bonne opération, à l'aide de l'instruction de branchement `switch`, chaque `case` représentant une opération, le résultat du calcul est stocké dans la variable locale `r` qui est retournée par la méthode. Il est à noter que si un mauvais symbole d'opération est rentré, alors, l'erreur est traitée dans le `default` du `switch`, on affecte alors à `r` la valeur symbolique `NaN` qui signifie « Not A Number ». On a ainsi défini ici quelques traitements simples d'erreurs.

Solution possible :

```
public class Test {
```

```
  static public void main(String [] args) {
   if(args.length != 3) {
    System.out.println("Nombre d'arguments incorrects");
    System.exit(0);
   }
   double v1 = Double.parseDouble(args[0]);
   char op = args[1].charAt(0);
   double v2 = Double.parseDouble(args[2]);
   double r = calcul(v1, v2, op);
   System.out.println(r + " = "+v1 +" "+ op + " "+ v2);
  }
  static double calcul(double v1, double v2, char op){
   double r;
   switch(op){
    case '+' : r = v1 + v2; break;
    case '-' : r = v1 - v2; break;
    case '*' : r = v1 * v2; break;
    case '/' : r = v1 / v2; break;
    default : r = Double.NaN;
   }
   return r;
  }
 }
```

Résultat des deux exécutions :

```
>Java Test 5 + 3
8.0 = 5.0 + 3.0
>Java Test 5 ? 3
NaN = 5.0 ? 3.0
>Java Test 5 -
Nombre d'arguments incorrects
```

7.6 Copie, comparaison et concaténation de tableaux

Le programme qui suit permet de copier un tableau dans un autre, de comparer deux tableaux, de concaténer deux tableaux et enfin d'afficher un tableau. Le programme manipule des tableaux d'`int`.

La méthode `copier` recopie le contenu d'un tableau `ta`, dont la référence est passée en argument, dans le tableau dont la référence `tb` est retournée. Le tableau `tb`, dans un premier temps, créé dynamiquement de taille identique à celui qui est passé en argument, puis, dans la boucle `for`, on effectue une copie élément par élément de tous les éléments du tableau passé dans le tableau créé. Puis la référence, du tableau créé qui vient d'être mis à jour est retournée. Afin d'éviter une erreur d'exécution (`Java.lang.NullPointerException`), on vérifie en début de méthode que `ta` est non `null` (non vide).

Rappelons, préalablement que la comparaison directe `t1==t3` donnerait un résultat false, car les valeurs de `t1` et `t3` correspondent à deux références distinctes alors que les tableaux ont le même contenu. La méthode `compare` vérifie dans la boucle `for` l'égalité élément par élément des éléments de `ta` et `tb` de même indice, dès que deux éléments sont distincts, on retourne la valeur `false`. Lorsque l'on arrive à la fin de la boucle, la valeur `true` est retournée. Ici aussi, on vérifie que les tableaux ne sont pas `null` (non

vides). On vérifie aussi que les tableaux font la même taille, afin d'éviter, en particulier, un problème de débordement d'indice (« `Java.lang.ArrayIndexOutOfBoundsException` »), les tableaux pouvant, dans le cas général, être de tailles différentes.

La méthode de concaténation fonctionne un peu sur le même principe que celle de copie. Le tableau créé `tc` devant pouvoir contenir les éléments de `ta` et de `tb`, il faut lui allouer une taille égale à la somme des tailles de `ta` et `tb (ta.length + tb.length)`. On commence par recopier les éléments de `ta`, puis ceux de `tb`, pour la seconde partie, la copie dans `tc` s'effectue à partir de l'indice `i` mis à jour dans la boucle précédente.

Solution possible :

```
public class Test {
 static public void main(String [] args) {
  int[]t1 = {1,2,3};
  int[]t2 = {1,2,3,4,5};
  int[]t3 = copier(t1);
  afficher(t3);
  System.out.println(comparer(t1, t2));
  System.out.println(comparer(t1, t3));
  int[] t4 = concatener(t1, t2);
  afficher(t4);
 }
 static int[] copier(int[] ta){
  if(ta == null) return null;
  int[] tb = new int[ta.length];
  for(int i = 0; i < ta.length; i++)
   tb[i] = ta[i];
  return tb;
 }
 static boolean comparer(int[] ta, int[] tb){
  if(ta == null || tb == null) return false;
  if(ta.length != tb.length) return false;
  for(int i = 0; i < ta.length; i++)
   if (tb[i] != ta[i]) return false;
  return true;
 }
 static int[] concatener(int[] ta, int[] tb){
  if(ta == null) return tb;
  if(tb == null) return ta;
  int[] tc = new int[ta.length + tb.length];
  int i;
  for(i = 0; i < ta.length; i++)
   tc[i] = ta[i];
  for(int j = 0; j < tb.length; j++)
   tc[i+j] = tb[j];
  return tc;
 }
 static void afficher(int[] ta){
  if(ta == null) return ;
  for(int i = 0; i < ta.length; i++)
```

```
    System.out.print(ta[i] + " ");
   System.out.println();
  }
 }
```

Résultat de l'exécution :

```
1 2 3
false
true
1 2 3 1 2 3 4 5
```

7.7 Calcul matriciel

Il s'agit ici d'écrire des méthodes simples de calcul matriciel, la première méthode `somme` a pour l'objet d'effectuer la somme de deux matrices et la seconde `produit` effectue le produit de deux matrices. Pour chacune de ces méthodes la matrice résultat est retournée, elle est donc allouée dynamiquement par un `new` dans chaque méthode. Par ailleurs, avant d'effectuer le calcul, on vérifie la cohérence des dimensions des matrices passées.

Pour la somme, on se base sur la formule $c_{i,j} = a_{i,j} + b_{i,j}$, il est donc nécessaire d'effectuer deux boucles imbriquées (parcours sur les lignes et les colonnes).

Pour le produit, on se base sur la formule $c_{i,j} = \sum a_{i,k} \times b_{k,j}$ il est donc nécessaire d'effectuer trois boucles imbriquées (parcours sur les lignes et les colonnes et la somme des produits).

On pourra constater que la méthode d'affichage `print` utilise la méthode `printf`, tout à fait semblable à celle du langage C (introduite en même temps que les méthodes à nombre d'arguments variables avec le JDK 5.0, qui permet d'utiliser de nombreux formats d'affichage).

Solution possible :

```
public class Test {
 static public void main(String [] args) {
  double [][] ma = {{1,2},{2,1},{1,1}};
  double [][] mb = {{1,2, 1}, {2,1,2}};
  double [][] mc = produit(ma, mb);
  print(mc);
  double [][] md = {{2,1},{1,2},{2,2}};
  mc = somme(ma, md);
  print(mc);
 }
 static double [][] somme(double [][] m1, double [][] m2){
    if(m1.length == 0 || m1[0].length == 0) return null;
    if(m1.length != m2.length || m1[0].length != m2[0].length) return
null;

    double [][] mr = new double [m1.length][m1[0].length];
    for(int i=0; i<mr.length; i++) {
            for(int j=0; j< mr[0].length; j++) {
                mr[i][j] = m1[i][j] + m2[i][j];
            }
```

```
        }
        return mr;
    }
    static double [][] produit(double [][] m1, double [][] m2){
     if(m1.length == 0 || m1[0].length == 0) return null;
     if(m2.length == 0 || m2[0].length == 0) return null;
     if(m1[0].length != m2.length) return null;

     double [][] mr = new double [m1.length][m2[0].length];
     for(int i=0; i<mr.length; i++) {
            for(int j=0; j<mr[0].length; j++) {
               for(int k = 0; k < m1[0].length; k++)
                     mr[i][j] += (m1[i][k] * m2[k][j]);
            }
        }
        return mr;
    }
    static void print(double [][] matrice)
     {
       for(int i=0; i<matrice.length; i++) {
           if (i == 0)System.out.print("[");
            for(int j=0; j<matrice[i].length; j++) {
               if (i == 0 && j == 0)
                     System.out.printf("%5.2f", matrice[i][j]);
                else
                     System.out.printf(" %5.2f", matrice[i][j]);
            }
            if (i != matrice.length - 1)System.out.println(",");
            else System.out.println("]");
       }
     }
}
```

Résultat de l'exécution :

```
 [ 5,00  4,00  5,00,
  4,00  5,00  4,00,
  3,00  3,00  3,00]
[ 3,00  3,00,
  3,00  3,00,
  3,00  3,00]
```

Classes et Objets

Le corps et l'âme sont des vues prises du même objet à l'aide de méthodes différentes, des abstractions faites par notre esprit d'un être unique.

Alexis Carrel

1. Introduction

1.1 Exemple introductif

Considérons l'exemple suivant, dans lequel sont définies les données et les fonctions permettant de représenter un point et son comportement. On envisage ici des points dont les coordonnées sont entières et positives (pixels sur un écran, repérage de zones dans une usine, …). Le programme qui suit est écrit en langage C, langage structuré et procédural dont la syntaxe de base est proche de celle du Java. Afin de pouvoir décrire complètement un point, il est nécessaire de définir :

- Une structure de données `struct Point` représentant les informations relatives à un point (coordonnées)
- Des fonctions correspondant aux opérations possibles sur un point (initialisation, déplacements, …) : `initPoint`, `translatePoint`

A partir de ces définitions, il est possible de créer des variables de type `Point` en les déclarant et de les manipuler en utilisant les fonctions qui peuvent opérer dessus. Cependant, on peut noter que l'on a une séparation « syntaxique » entre les données qui caractérisent l'état du point et les fonctions qui décrivent son comportement, le regroupement est physique (même fichier). Du reste, on aurait pu séparer les fonctions de la structure, en les mettant dans des fichiers distincts. Or conceptuellement, l'état d'un `Point` est indissociable de son comportement. Par ailleurs, il est possible de définir des points incohérents, `p2` est initialisé avec une valeur négative et `p1` subit une translation qui donne à son abscisse une valeur négative.

Exemple :

```
struct Point {
    int x;
    int y;
};
void initPoint(int, int, struct Point *);
void translatePoint(int, int, struct Point *);
void affichePoint(struct Point);
void initPoint(int x, int y, struct Point *p) {
    if(x < 0 || y < 0) return;
    p->x = x;
    p->y = y;
}
```

```
void translatePoint(int dx, int dy, struct Point *p) {
    int nx = p->x + dx;
    int ny = p->y + dy;
    if(nx < 0 || ny < 0) return;
    p->x = nx;
    p->y = ny;
}
void affichePoint(struct Point p) {
    printf("(%d, %d)", p.x, p.y);
}
void main() {
    struct Point p1, p2;
    initPoint(1, 1, &p1);
    affichePoint(p1);
    p2.x = -1;
    p1.x -= 3;
}
```

1.2 Première classe et premiers objets

La première idée va consister à regrouper l'état et le comportement, afin d'obtenir une abstraction plus conforme à l'entité du monde réel qu'elle représente. Ce regroupement s'appelle un objet. Un objet est donc une abstraction[10] d'une entité du monde réel. Il possède :

- Une identité qui permet de le désigner de manière absolue (il est identifiable et différenciable des objets semblables ou non)
- Un état qui est l'ensemble des données qui vont le caractériser, état qui peut varier au cours du temps (lorsque les données évoluent)
- Un comportement constitué par un ensemble d'opérations que l'objet peut faire, ces opérations peuvent modifier l'état de l'objet

Une classe regroupe les objets qui ont les mêmes états (possibles) et les mêmes comportements. La classe constitue la spécification complète de ces objets, elle a une nature descriptive, c'est un modèle (encore une abstraction) qui permet de construire des objets, seuls les objets existent. Une classe, est un type de données, caractérisé par ses données (attributs) et des opérations décrivant le comportement commun aux objets de la classe, qui permet de créer des objets possédant ces propriétés.

Reprenons l'exemple du point que nous avons vu précédemment, mais cette fois ci en Java. On peut constater que le programme est constitué de deux classes, la classe qui contient la méthode `main` et celle qui définit la structure des points. Cette seconde classe, nommée `class Point` est composée de deux parties qu'elle encapsule. La première partie est constituée des définitions des coordonnées `x` et `y` d'un point. Ces attributs sont nommés champs ou encore données membres de la classe. La seconde partie est constituée de plusieurs fonctions qui décrivent le comportement d'un point. Ces fonctions sont appelées les méthodes ou fonctions membres de la classe.

Dans la classe `Test`, on déclare deux variables `p1` et `p2` de type `Point`. Comme pour les tableaux, il ne s'agit pas de `Point`, mais de référence de `Point`. La création d'un

[10] Représentation simplifiée

point s'effectue en utilisant l'opérateur `new` suivi de `Point()`. L'initialisation de `p1` s'effectue en invoquant la méthode `init`. Pour spécifier qu'il s'agit de l'appliquer à `p1`, le sélecteur de membre (ici un champ) est utilisé « `.` » : `p1.init(1, 1)`, il s'agit d'initialiser le `Point p1`. On remarque aussi, lors de l'exécution de l'instruction `System.out.println(p1);`que l'on affiche bien une adresse, celle du `Point` référencé par `p1` (elle est suffixée par `Point@` ce qui signifie qu'il s'agit de l'adresse d'un objet de type `Point`). Enfin, il est à noter que `p2` possède une adresse différente de `p1` et que bien que l'objet de type `Point` référencé par `p1` soit modifié, son adresse ne change pas.

Comme on peut le constater, en généralisant notre exemple :

- Tout objet a une identité qu'il conserve tout au long de sa vie, c'est la valeur de son adresse (sa référence).
- Tout objet possède un état, ce sont les valeurs de ses champs
- Tout objet possède un comportement, c'est la définition de ses méthodes.

Exemple :

```
public class Test {
 static public void main(String [] args) {
  Point p1 = new Point(), p2 = new Point();
  System.out.println(p1);
  System.out.println(p2);
  p1.init(1, 1);
  p1.affiche();
  p1.translate(1, 2);
  p1.affiche();
  System.out.println(p1);
 }
}
class Point {
 int x, y;
 void init(int nx, int ny) {
  if(nx < 0 || ny < 0) return;
    x = nx;
    y = ny;
 }
 void translate(int dx, int dy) {
  int nx = x + dx;
    int ny = y + dy;
  if(nx < 0 || ny < 0) return;
    x = nx;
    y = ny;
 }
 void affiche() {
    System.out.println("("+ x+", "+ y+")");
 }
}
```

Résultat de l'exécution :

```
Point@19821f
Point@addbf1
```

```
(1, 1)
(2, 3)
Point@19821f
```

2. Bases sur les classes et Objets

2.1 Définition d'une classe et droits d'accès

Une classe Java se définit de manière simplifiée :

```
[public] class NomClasse {
    [public | private] définition des champs
    [public | private] définition des méthodes
}
```

`NomClasse` est le nom de la **classe** qui doit être identificateur licite du langage. Par convention, la première lettre de l'identificateur d'une classe est une majuscule (pour le reste, on adopte la même convention que pour les variables locales). Le **droit d'accès** optionnel `public`[11], indique que la classe est accessible de partout (Il n'y a qu'une classe publique par fichier). Le droit d'accès fait partie des **modificateurs** qui affinent la déclaration d'un champ ou d'une variable (on a vu précédemment le modificateur `final`). Une classe est constituée de membres : méthodes et champs, dont les définitions se trouvent entre les accolades de la classe.

Droits d'accès des membres : `public`, rien ou `private`

- `public` : les membres publics sont accessibles par tous, à partir de la classe, à partir d'une autre classe du même fichier ou à partir d'une classe d'un autre fichier. La partie publique est appelée « interface[12] ».
- rien : tous ces membres sont accessibles à partir de la classe, à partir d'une autre classe du même fichier ou d'un fichier du même répertoire (package anonyme voir §10 p115), c'est le modificateur par défaut, on le qualifie d'accès **friendly**
- `private` : les membres privés ne sont accessibles que par les méthodes (fonctions membres) de la classe. La partie privée est aussi appelée implémentation ou réalisation.

Les **champs** se déclarent comme des variables locales, à la différence, qu'ils peuvent être précédés par un modificateur.

Une déclaration de champ a toujours la forme suivante :

```
[public | private] [final] type identificateur [= valeur initiale];
```

Le type d'un champ correspond à un type primitif ou à une classe (ou un tableau ou une énumération). A la différence des variables locales (voir §2.3 p15) qui doivent aussi être initialisées avant d'être utilisées, les champs sont initialisés avec une valeur par défaut (avec la valeur par défaut correspondant au type), si ils ne sont pas explicitement initialisés. Seuls les champs constants doivent obligatoirement être initialisés, même de manière différée[13]. Il est conseillé de rendre privés les champs, ainsi, on y accède au

[11] Seules les classes internes peuvent être privées

[12] Le terme interface a une autre définition en Java que nous verrons dans la suite

[13] On verra plusieurs possibilités dans la suite

travers des méthodes qui assurent les cohérences des données membres par rapport à leur définition (si les méthodes sont bien écrites). Chaque objet, étant stocké dans une zone mémoire qui lui est propre, possède ses propres valeurs de champs[14].

Les **méthodes** se définissent exactement comme nous l'avons vu dans le §6 p37. Bien évidemment, les modificateurs peuvent être exploités. Il est conseillé de rendre publiques les méthodes car ce sont elles qui permettent de manipuler les objets. Une classe peut disposer de méthodes privées ; dans ce cas, elles ne sont utilisables que par les autres méthodes de la classe, il s'agit en général de méthodes utilitaires.

Les définitions des membres peuvent être placées n'importe où dans la définition de la classe, nous conseillons de regrouper les méthodes avec les méthodes et les champs avec les champs.

2.2 Création et cycle de vie d'un objet

Le cycle de vie d'un objet est identique à celui d'un tableau, il va de la création jusqu'à la destruction de l'objet :

- **Déclaration** (de la référence) d'un objet[15] (l'objet n'existe pas encore)
- **Allocation dynamique** ou **création** d'un objet en utilisant l'opérateur `new NomClasse()`, l'objet créé est de type `NomClasse`, on dit que l'objet créé est une **instance** de la classe `NomClasse`. L'opérateur `new` retourne la référence de l'objet créé (adresse de la mémoire allouée qui contient l'objet)
- **Initialisation** qui initialise chaque champ avec sa valeur initiale (si elle est explicitée), ou sinon, sa valeur par défaut (qui dépend du type du champ). Les champs sont toujours initialisés. Cette initialisation s'effectue au moment de la création (la création et l'initialisation sont indissociables).
- **Mise à jour** et manipulation de l'objet (normalement au travers de ses méthodes). Lorsque l'on appelle une méthode en dehors de la classe à laquelle elle appartient, il faut la préfixer par le nom d'une référence de l'objet (généralement une variable) à laquelle on veut l'appliquer `nomObjet.nomMéthode(var1, var2, ..., vark)` en utilisant le sélecteur de membre « . ». L'appel d'une méthode peut conduire à la modification des champs de l'objet.
- **Destruction** (mort) de l'objet par le « ramasse miettes » après que l'objet ne soit plus référencé (pas forcément dès que)

Il est possible, comme dans le cas des tableaux, de créer des objets anonymes qui sont passés à des méthodes.

Précisions de vocabulaire :

`NomClasse nomVariable ;` la variable `nomVariable` est une variable de type référence d'un objet de type `NomClasse`. Nous nous permettrons l'abus de vocabulaire `nomVariable` est une variable (ou une référence) de type `NomClasse`.

`nomVariable = new nomClasse(…);` nous affectons à la variable `nomVariable` la référence d'un objet de type `NomClasse` (ou la référence d'une instance de `NomClasse`).

[14] On verra dans la suite qu'il existe des données partagées par tous les objets d'une classe qui sont qualifiées de variables de classe et déclarées `static`

[15] Dans la suite, il nous arrivera de faire l'abus de langage « déclaration d'un objet »

Nous nous permettrons l'abus de vocabulaire nous affectons à la variable (ou à la référence) `nomVariable` l'objet de type `NomClasse`.

Reprenons l'exemple précédent, en modifiant légèrement la classe `Point`, cette fois ci, on suppose qu'un point est inscrit dans un rectangle ($0 \leq x < 200$, $0 \leq y < 100$). Par rapport, à l'exemple précédent, les méthodes sont déclarées publiques et les champs privés. Désormais, il est impossible de modifier les champs `x` et `y` en dehors de la classe `Point`. La tentative : `p1.x = -1`, dans `main` conduirait à l'erreur de compilation : « `x has private access in Point` ». Bien évidemment, ces champs sont accessibles dans la classe `Point`. Les méthodes qui sont susceptibles de les modifier vérifient préalablement que les mises à jour sont licites (les méthodes doivent laisser les objets dans un état cohérent). On peut du reste noter que la méthode `coincide` a aussi accès au champ d'un point passé en argument, en effet c'est aussi un objet de type `Point`. Si, on oublie d'initialiser le champ `XMAX`, déclaré comme `final` (constante), alors, cela provoque l'erreur de compilation : variable « `XMAX might not have been initialized` », une constante devant être forcément initialisée.

La Figure 10 illustre le cycle de vie de l'objet de type `Point` (référencé par) `p1` (la variable `p1` est une variable de type référence d'un objet de type `Point`). Dans un premier temps, la variable `p1` est déclarée, elle n'a pas encore de valeur (aucun objet de type `Point` n'existe). Puis, cette variable est initialisée avec la référence d'un objet de type `Point` retourné par `new`, on vient donc de créer une instance de la classe `Point` (objet de type `Point`). Lors de cette création, les différents champs de cet objet sont initialisés, `x` et `y` avec la valeur par défaut `0`, `XMAX` et `YMAX` avec les constantes littérales `200` et `100` (initialisation explicite). Ensuite, l'objet référencé par `p1` va subir quelques modifications de ses champs lors des appels des méthodes `init` et de `translate` qui lui sont appliquées. Puis, lorsque l'on affecte à `p1` la valeur `null`, l'objet précédemment référencé par `p1` ne l'est plus (et par aucune autre variable), il devient inaccessible et peut être considéré comme détruit (même si cette destruction a lieu effectivement « beaucoup plus tard »). Le second appel de la méthode `coincide` s'effectue en passant la référence d'un objet anonyme créé par un `new`. Comme, cette objet possède les mêmes valeurs de ses champs `x` et `y` que `p2` (valeur par défaut), `true` est retournée.

Exemple :

```
public class Test {
 static public void main(String [] args) {
  Point p1 , p2;
  p1 = new Point();
  p2 = new Point();
  System.out.println(p1);
  System.out.println(p2);
  System.out.println(p1.coincide(p2));
  p1.init(1, 1);
  p1.affiche();
  p2.affiche();
  p1.translate(1, 2);
  p1.affiche();
  System.out.println(p1);
  System.out.println(p1.coincide(p2));
  System.out.println(p2.coincide(new Point()));
  p1 = null;
```

```
   System.out.println(p1);
  }
 }
 class Point {
  private int x, y;
  private final int XMAX = 200;
  private final int YMAX = 100;
  public void init(int nx, int ny) {
   if(nx < 0 || ny < 0) return;
   if(nx > XMAX || ny > YMAX) return;
     x = nx;
     y = ny;
  }
  public void translate(int dx, int dy) {
   int nx = x + dx;
     int ny = y + dy;
     if(nx < 0 || ny < 0) return;
   if(nx > XMAX || ny > YMAX) return;
     x = nx;
     y = ny;
  }
  public boolean coincide(Point p) {
   if(p.x == x && p.y == y) return true;
   return false;
  }
  public void affiche() {
     System.out.println("("+ x+", "+ y+")");
  }
 }
```

Résultat de l’exécution :

```
Point@19821f
Point@addbf1
true
(1, 1)
(0, 0)
(2, 3)
Point@19821f
false
true
null
```

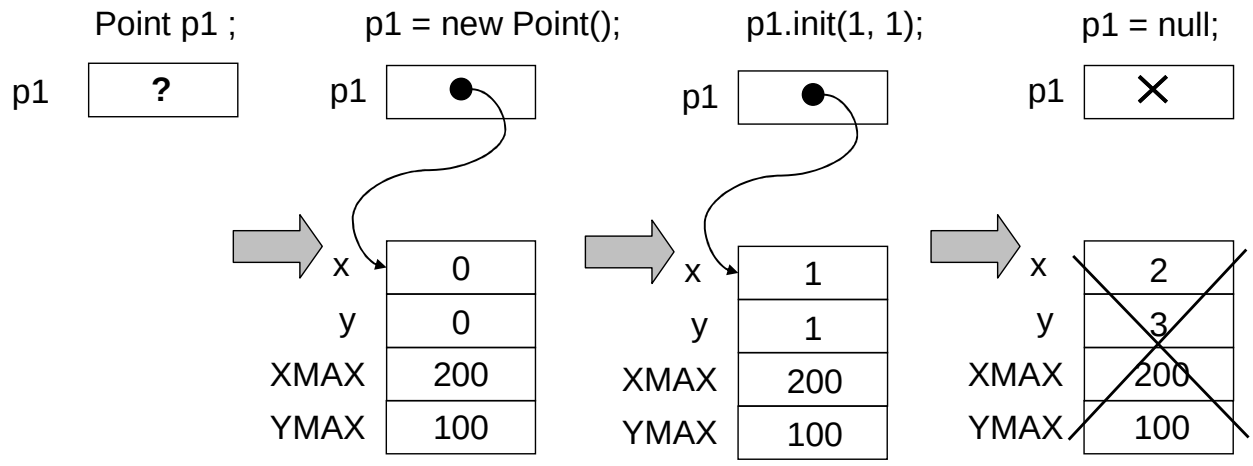


Figure 10 : Cycle de vie d'un objet

Remarque :

On aurait pu écrire la méthode privée « utilitaire » `validePosition` dont l'objectif est de contrôler que les coordonnées d'un point sont correctes, uniquement utilisée dans les méthodes `init` et `translate` (ou toute autre méthode modifiant les coordonnées d'un point), qui ne fait pas partie de l'interface de la classe `Point`. Ceci évite de dupliquer du code.

Exemple :

```
private boolean validePosition(int nx, int ny){
  if((nx < 0 || ny < 0
  || nx > XMAX || ny > YMAX)) return false;
  return true;
 }
public void init(int nx, int ny) {
  if(!validePosition(nx, ny)) return;
…
public void translate(int dx, int dy) {
  int nx = x + dx;
    int ny = y + dy;
    if(!validePosition(nx, ny)) return;
```

2.3 Objets arguments de méthodes

Comme pour les tableaux, il est possible de passer comme arguments une ou plusieurs références d'objet. Dans ce cas, il est possible d'en modifier le contenu. Bien évidemment, la méthode doit avoir les droits d'accès sur les champs concernés.

Le programme suivant illustre la passation de paramètre d'objets. Les objets (référencés par les variables) `p1` et `p2` sont créés et initialisés dans la méthode `main` et leurs références passés à la méthode `echange`. Les paramètres de cette méthode sont aussi des références d'objet de type `Point`, lors de l'appel, les valeurs de ces références sont recopiées respectivement dans `pa` et `pb`, qui vont donc désigner les mêmes objets que `p1` et `p2`. Ainsi, toutes les modifications des objets effectuées dans `echange` modifient aussi les objets référencés dans `main`. La Figure 11 illustre l'évolution de la configuration mémoire (seuls les champs `x` et `y` des objets de type `Point` sont représentés).

On peut observer, que l'on utilise la variable `p3` pour préfixer l'appel de `echange`, comme `p3` n'est pas impacté, on aurait pu prendre n'importe quel objet de type `Point` comme `p1` ou `p2`, on verra dans la suite comment écrire cela de manière plus élégante (méthode de classe).

Exemple :

```
public class Test {
 static public void main(String [] args) {
  Point p1 , p2, p3;
  p1 = new Point();
  p2 = new Point();
  p3 = new Point();
  p1.init(1,2);
  p2.init(3,4);
  p3.echange(p1, p2);
  p1.affiche();
  p2.affiche();
 }
}
class Point {
 private int x, y;
 private final int XMAX = 200;
 private final int YMAX = 100;
 public void init(int nx, int ny) {
  if(nx < 0 || ny < 0) return;
  if(nx > XMAX || ny > YMAX) return;
    x = nx;
    y = ny;
 }
 public void translate(int dx, int dy) {
  int nx = x + dx;
    int ny = y + dy;
    if(nx < 0 || ny < 0) return;
  if(nx > XMAX || ny > YMAX) return;
    x = nx;
    y = ny;
 }
 public boolean coincide(Point p) {
  if(p.x == x && p.y == y) return true;
  return false;
 }
 public void affiche() {
    System.out.println("("+ x+", "+ y+")");
 }
 public void echange(Point pa, Point pb) {
  int t = pa.x;
  pa.x = pb.x;
  pb.x = t;
  t = pa.y;
  pa.y = pb.y;
```

```
   pb.y = t;
  }
 }
```

Résultat de l'exécution :

```
(3, 4)
(1, 2)
```

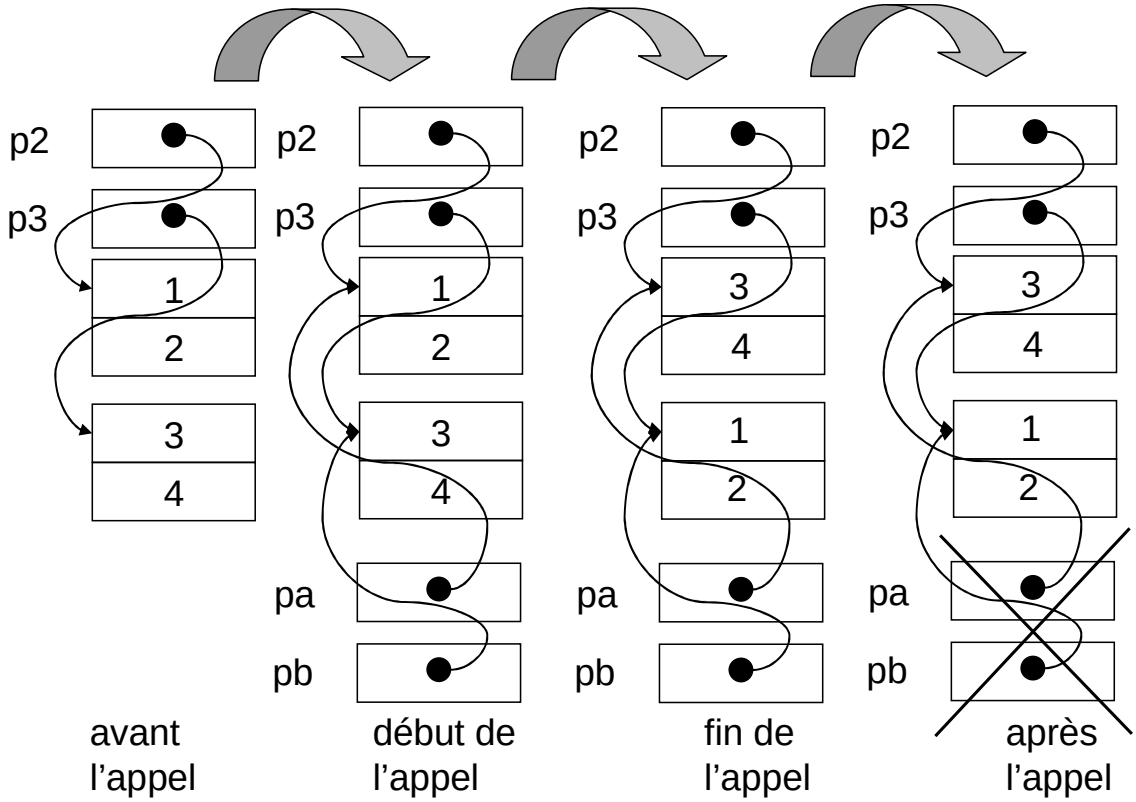


Figure 11 : Passation d'objets à une méthode

Les paramètres d'une méthode peuvent être déclarés `final`, ce qui signifie que les valeurs de ces paramètres ne peuvent pas être modifiées dans le corps de la méthode. Lorsqu'il s'agit de paramètres de type primitif, par définition de la passation de paramètres par valeur, il n'y a aucun impact, au niveau des arguments correspondants passés qui ne peuvent pas être modifiés (c'est une copie de leur valeur qui est passée). Cependant, lorsque l'on passe un objet, le mot clé `final` s'applique bien à la référence passée qui ne peut pas être modifiée, mais le contenu (l'objet effectivement référencé), lui peut l'être (il en va de même avec les tableaux).

L'exemple suivant reprend la méthode `echange` précédente, mais en ajoutant le modificateur `final` devant les paramètres de type `Point`, le résultat est identique, les objets passés sont bien modifiés. Par contre, si dans la méthode, nous avions cherché à modifier `pa`, en écrivant, par exemple, `pa = new Point()`, nous aurions déclenché l'erreur de compilation « `final parameter pa may not be assigned` ».

Exemple :

```
public class Test {
 static public void main(String [] args) {
  Point p1 , p2, p3;
  p1 = new Point();
```

```
   p2 = new Point();
   p3 = new Point();
   p1.init(1,2);
   p2.init(3,4);
   p3.echange(p1, p2);
   p1.affiche();
   p2.affiche();
  }
 }
 class Point {
  private int x, y;
  private final int XMAX = 200;
  private final int YMAX = 100;
  public void init(int nx, int ny) {
   if(nx < 0 || ny < 0) return;
   if(nx > XMAX || ny > YMAX) return;
     x = nx;
     y = ny;
  }
  public void echange(final Point pa, final Point pb) {
   int t = pa.x;
   pa.x = pb.x;
   pb.x = t;
   t = pa.y;
   pa.y = pb.y;
   pb.y = t;
  }
  public void affiche() {
     System.out.println("("+ x+", "+ y+")");
  }
 }
```

Résultat de l'exécution :

```
(3, 4)
(1, 2)
```

2.4 Règles de portées des champs et variables locales

Une méthode peut posséder des variables locales de même nom que les champs de sa classe. Dans ce cas, ce sont des variables homonymes. La règle de portée qui s'applique est le masquage du champ par la variable locale, entre le moment où la variable locale est déclarée, et sa fin de vie (fin de bloc). Pour éviter des confusions, on ne peut que déconseiller de nommer des champs et des variables locales de la même façon.

Dans le programme suivant, la classe `ClasseX` possède trois champs `t`, `x`, et `y`, respectivement initialisés avec les valeurs `0`, `1` et `2`. La méthode `testPortee` possède deux variables locales, la première `x` qui correspond à son unique paramètre (portée méthode), la seconde `y` déclarée dans un bloc (portée bloc). Dès que l'on créé l'objet de type `Point` (référencé par) `cx`, les trois champs sont créés et leur durée de vie correspond à celui de `cx` (fin de `main`). Lorsque l'on rentre dans la méthode `testPortee`, on crée la variable locale (paramètre) `x` qui masque alors le champ `x` qui devient alors inaccessible,

le champ `y` l'est toujours. Dès que l'on rentre dans le bloc, on crée la variable locale `y` qui masque alors le champ `y` qui devient alors inaccessible (`x` et `y` existent toujours, mais ne sont plus accessibles). Dès que l'on sort du bloc, la variable `y` disparaît, le champ `y` est à nouveau accessible, et dès que l'on sort de `testPortee`, `x` est à nouveau accessible.

Exemple :

```
public class Test {
 static public void main(String [] args) {
  ClasseX cx = new ClasseX();
  cx.testPortee(3);
 }
}
class ClasseX {
 private int t = 0, x = 1, y = 2;
 public void testPortee(int x) {
  System.out.println(t + " " + x + " " + y);
  {
   int y = 4;
   System.out.println(t + " " + x + " " + y);
  }
  System.out.println(t + " " + x + " " + y);
 }
}
```

Résultat de l'exécution :

```
0 3 2
0 3 4
0 3 2
```

2.5 Surcharge et droit d'accès

Les droits d'accès peuvent influencer le mécanisme de surcharge, en dissimulant certaines méthodes surchargées. En effet, un objet qui n'a pas accès à la méthode correspondant « le mieux » aux types d'arguments passés provoquera l'exécution de la « meilleure » méthode surchargée à laquelle il a accès (au pire à une erreur de compilation).

Le programme suivant illustre ce phénomène, des affichages des entêtes de méthodes sont ajoutés dans les méthodes concernées. L'appel de `init` avec deux arguments de type `int` déclenche l'exécution de la méthode `init` possédant des paramètres de type `double`. Il en va de même pour l'appel de la méthode `translate`. Dans l'exemple proposé, les méthodes possédant un comportement identique, cela ne pose pas de réel problème.

Exemple :

```
public class Test {
 static public void main(String [] args) {
  Point p1;
  p1 = new Point();
  p1.init(1,2);
  p1.translate(2,2);
  p1.affiche();
```

```
  }
 }
 class Point {
  private double x, y;
  public void init(double nx, double ny) {
   System.out.println("init(double nx, double ny)");
     x = nx;
     y = ny;
  }
  private void init(int nx, int ny) {
   System.out.println("init(int nx, int ny)");
     x = nx;
     y = ny;
  }
  public void translate(double nx, double ny) {
   System.out.println("translate(double nx, double ny)");
     x += nx;
     y += ny;
  }
  private void translate(int nx, int ny) {
   System.out.println("translate(int nx, int ny)");
     x += nx;
     y += ny;
  }
  public void affiche() {
     System.out.println("("+ x+", "+ y+")");
  }
 }
```

Résultat de l'exécution :

```
init(double nx, double ny)
translate(double nx, double ny)
(3.0, 4.0)
```

Si les méthodes `init` et `translate` avaient été déclarées `public`, le résultat aurait été :

```
init(int nx, int ny)
translate(int nx, int ny)
(3.0, 4.0)
```

3. Constructeurs et initialisation des objets

3.1 Rôle et définition

Les constructeurs d'une classe sont des méthodes particulières de cette classe qui définissent la manière dont les objets appartenant à cette classe sont créés et initialisés. Le constructeur est appelé lors de la création d'un nouvel objet par `new`. Les constructeurs ont pour caractéristiques principales :

- ils ne retournent aucune valeur (aucun type de retour ne doit être spécifié, même pas `void`)

- ils ne peuvent pas être appelés explicitement ailleurs qu'au moment de la création d'un objet
- ils ont le même nom que leur classe d'appartenance.

Sinon, ils ont les mêmes caractéristiques générales que les autres méthodes : ils peuvent posséder un nombre quelconque d'arguments, ils peuvent être surchargés, les arguments sont passés par valeur, ils peuvent posséder plusieurs return, …

La forme générale d'un constructeur de la classe `NomClasse` est :

```
[modificateurs] NomClasse(type1 nom1, ..., typek nomk) {
 instructions
 [return; ]
}
```

Comme pour une méthode chaque couple de la liste `type1 nom1`, ..., `typek nomk` correspond à la déclaration de chaque paramètre. En général, le modificateur d'un constructeur est public (un constructeur privé ne pourra pas être utilisé pour instancier des objets).

L'appel effectif du constructeur se trouve tout de suite après un `new` :

```
RéférenceObjet = new NomClasse(var1, var2, ... , vark)
```

La liste de valeurs `var1`, `var2`, ..., `vark` constituent les arguments passés au constructeur, les valeurs de ces arguments doivent être de types compatibles avec les paramètres correspondants du constructeur.

Les constructeurs vont donc faciliter et automatiser le processus d'initialisation des objets lors de leur création.

Dans le cas de la classe `Point` précédemment définie, l'utilisateur de cette classe doit appeler explicitement la méthode d'initialisation `init` pour qu'un point déjà créé soit initialisé avec des valeurs autres que les valeurs initiales de ces champs. L'utilisateur doit donc opérer en deux temps.

Dans le programme suivant, un constructeur dont l'objectif est d'initialiser les valeurs des champs `x` et `y` des objets de type `Point` est défini. On peut vérifier que ce constructeur ne retourne rien et qu'il porte le même nom que la classe (aucun type de retour). Ce constructeur de `Point` est appelé lors de la création de nouveaux objets de type `Point`. Par exemple, l'instruction `p1 = new Point(1,2);` va créer directement un nouvel objet `Point` dont les champs `x` et `y` vaudront `1` et `2` et en retourner la référence qui est affectée à `p1`. On peut observer que ce constructeur a un comportement identique à celui de la méthode `init`.

Exemple :

```
public class Test {
 static public void main(String [] args) {
  Point p1 , p2, p3;
  p1 = new Point(1,2);
  p2 = new Point(3,4);
  p1.translate(5, 5);
  p1.affiche();
  p2.affiche();
 }
}
```

```
class Point {
 private int x, y;
 private final int XMAX = 200;
 private final int YMAX = 100;
 public  Point(int nx, int ny) {
  if(nx < 0 || ny < 0) return;
  if(nx > XMAX || ny > YMAX) return;
    x = nx;
    y = ny;
 }
 public void translate(int dx, int dy) {
  int nx = x + dx;
    int ny = y + dy;
    if(nx < 0 || ny < 0) return;
    if(nx > XMAX || ny > YMAX) return;
    x = nx;
    y = ny;
 }
 public boolean coincide(Point p) {
  if(p.x == x && p.y == y) return true;
  return false;
 }
 public void affiche() {
    System.out.println("("+ x+", "+ y+")");
 }
}
```

Résultat de l'exécution :

```
(6, 7)
(3, 4)
```

Remarques :

Un constructeur ne pouvant pas être appelé directement, la tentative d'appel `p2.Point(5,5) ;` aurait donné lieu à l'erreur de compilation « `cannot find symbol method Point(int,int )` »

Bien qu'un constructeur ait pour objectif d'initialiser les champs des objets ceux-ci le sont tout de même avant, soit explicitement avec les valeurs initiales, soit implicitement avec les valeurs par défaut. Même dans le cas des champs constants, les constructeurs peuvent être aussi utilisés pour faire de l'initialisation différée (comme pour les autres champs). Nous avons écrit une seconde version du constructeur de la classe `Point`, en initialisant les deux champs constants `XMAX` et `XMAX` dans le constructeur plutôt qu'au niveau de la déclaration de ces champs. Il faut bien évidemment vérifier que l'on n'a pas besoin d'utiliser ces champs avant, le test `if(nx > XMAX || ny > YMAX)` n'aurait pas pu être remonté avant l'initialisation des champs constants.

```
class Point {
 private int x, y;
 private final int XMAX = 200;
 private final int YMAX = 100;
```

```
class Point {
 private int x, y;
 private final int XMAX ;
 private final int YMAX;
```

```
public  Point(int nx, int ny) {
  if(nx < 0 || ny < 0) return;
  if(nx > XMAX || ny > YMAX)
return;
  x = nx;
  y = ny;
}
```

```
public  Point(int nx, int ny) {
  XMAX = 200;
  YMAX = 100;
  if(nx < 0 || ny < 0) return;
  if(nx > XMAX || ny > YMAX)
return;
  x = nx;
  y = ny;
}
```

Tableau 6 : Initialisation des champs constants

3.2 Surcharge et constructeur par défaut

Un constructeur peut être surchargé comme toute méthode usuelle. Ce qui laisse au programmeur une grande liberté pour initialiser les objets de ses classes. Ainsi, plusieurs constructeurs possédant des signatures distinctes peuvent coexister au sein d'une même classe.

Dans l'exemple qui suit, plusieurs constructeurs sont définis dans la classe `Point`. Afin de visualiser le comportement du programme, un affichage est ajouté comme première instruction de chaque constructeur, il indique l'entête du constructeur appelé[16]. On remarque que c'est bien le bon constructeur qui est sélectionné, comme dans le cas des autres méthodes, la signature est bien discriminante. Le constructeur qui reçoit comme argument la référence d'un objet de type `Point` (`Point(Point p)`) est appelé constructeur de recopie, le constructeur sans paramètre est appelé le constructeur par défaut (`Point()`).

Exemple :

```
public class Test {
 static public void main(String [] args) {
  Point p1, p2, p3, p4, p5;
  p1 = new Point(1,2);
  p2 = new Point(3);
  p3 = new Point();
  p4 = new Point(5.1,6.2);
  p5 = new Point(p1);
  p1.affiche();
  p2.affiche();
  p3.affiche();
  p4.affiche();
  p5.affiche();
 }
}
class Point {
```

[16] A ne pas faire dans un programme, c'est uniquement pour illustrer

```
  private int x, y;
  private final int XMAX = 200;
  private final int YMAX = 100;
  public  Point(int nx, int ny) {
   System.out.println("public  Point(int nx, int ny)");
   if(nx < 0 || ny < 0) return;
      if(nx > XMAX || ny > YMAX) return;
      x = nx;
      y = ny;
  }
  public  Point(double nx, double ny) {
   System.out.println("public  Point(double nx, double ny)");
   if(nx < 0 || ny < 0) return;
      if(nx > XMAX || ny > YMAX) return;
      x = (int)nx;
      y = (int)ny;
  }
  public  Point(int nx) {
   System.out.println("public  Point(int nx)");
   if(nx < 0 ) return;
      if(nx > XMAX ) return;
      x = nx;
  }
  public  Point() {
   System.out.println("public  Point()");
      x = y = 0;
  }
  public Point(Point p) {
   System.out.println("public Point(Point p)");
      x = p.x;
      y = p.y;
  }
  public void affiche() {
      System.out.println("("+ x+", "+ y+")");
  }
 }
```

Résultat de l'exécution :

```
public  Point(int nx, int ny)
public  Point(int nx)
public  Point()
public  Point(double nx, double ny)
public Point(Point p)
(1, 2)
(3, 0)
(0, 0)
(5, 6)
(1, 2)
```

En l'absence de tout constructeur, le compilateur Java en synthétise un de type constructeur par défaut (sans paramètre). Ce constructeur, qui ne fait rien, permet

d'utiliser l'écriture `Point p1 = new Point()` (voir le premier programme du §1.2 p59). Cependant, dès que l'on définit un constructeur, le compilateur ne synthétise plus de constructeur par défaut (le programmeur prend la responsabilité de l'initialisation des objets). Ainsi, le programme suivant donne lieu à l'erreur de compilation « `cannot find symbol constructor Point()` », le compilateur ne trouve plus le constructeur `Point()`.

Exemple :

```
public class Test {
 static public void main(String [] args) {
  Point p1;
  p1 = new Point();
 }
}
class Point {
 private int x, y;
 private final int XMAX = 200;
 private final int YMAX = 100;
 public  Point(int nx, int ny) {
  System.out.println("public  Point(int nx, int ny)");
  if(nx < 0 || ny < 0) return;
    if(nx > XMAX || ny > YMAX) return;
    x = nx;
    y = ny;
 }
}
```

Par contre, le programme suivant est bien syntaxiquement correct. Le compilateur trouve le constructeur `Point()`, c'est lui qui l'a synthétisé, ce constructeur « implicite » ayant certainement « l'allure suivante ».

```
public Point() {
}
```

Exemple :

```
public class Test {
 static public void main(String [] args) {
  Point p1;
  p1 = new Point();
 }
}
class Point {
 private int x, y;
 private final int XMAX = 200;
 private final int YMAX = 100;
}
```

3.3 Mot clé this et constructeurs

Le mot clé `this` désigne la référence de l'objet courant c'est l' « autoréférence » de l'objet. Ce mot clé n'a de sens que dans la définition de la classe. Dans la définition d'une classe, les membres (champs ou méthodes) d'une classe sont implicitement référencés par rapport à `this` (désignation relative).

L'exemple suivant montre à quoi correspond la valeur de `this` qui est affichée dans la méthode `affiche`. Cette valeur correspond exactement à la référence de l'objet. Pour l'objet `p1`, on constate bien que la valeur de la référence affichée dans `main` et la valeur de l'autoréférence `this` de `p1` qui est affichée dans l'appel de la méthode `affiche` `p1.affiche();` sont bien égales et valent `@19821f`. On peut remarquer aussi, l'utilisation de l'écriture `this.x` qui désigne le champ `x` de l'objet courant, cette désignation des champs est implicite, ce qui permet, dans notre cas, d'écrire de façon indifférenciée `x` ou `this.x`.

Exemple :

```
public class Test {
 static public void main(String [] args) {
  Point p1, p2;
  p1 = new Point(1,2);
  p2 = new Point(3, 4);
  System.out.println(p1);
  System.out.println(p2);
  p1.affiche();
  p2.affiche();
 }
}
class Point {
 private int x, y;
 private final int XMAX = 200;
 private final int YMAX = 100;
 public  Point(int nx, int ny) {
  if(nx < 0 || ny < 0) return;
     if(nx > XMAX || ny > YMAX) return;
     this.x = nx;
     y = ny;
 }
 public void affiche() {
     System.out.println("("+ x+", "+ y+")");
     System.out.println(this);
 }
}
```

Résultat de l'exécution :

```
Point@19821f
Point@addbf1
(1, 2)
Point@19821f
(3, 4)
Point@addbf1
```

L'autoréférence `this` a de nombreuses utilités. On peut utiliser « `this.` » comme technique de résolution de portée afin de différencier une variable locale et un champ homonymes. Bien évidemment, il faut éviter de donner le même nom aux variables locales et aux champs.

Dans l'exemple suivant, les paramètres du constructeur de la classe `Point` ont le même identificateur que ceux des champs, champ `x` et paramètre `x`, champ `y` et paramètre `y`. Par définition des règles de portée, le paramètre `x` masque le champ `x`, `x` désigne donc sans ambiguïté le paramètre et pas le champ. Lors de l'initialisation des champs `x` et `y`, on utilise l'autoréférence `this`, ainsi `this.x` désigne le champ `x`, ceci résout donc le problème.

Exemple :

```
public class Test {
 static public void main(String [] args) {
  Point p1, p2;
  p1 = new Point(1,2);
  p2 = new Point(3, 4);
  p1.affiche();
  p2.affiche();
 }
}
class Point {
 private int x, y;
 private final int XMAX = 200;
 private final int YMAX = 100;
 public  Point(int x, int y) {
  if(x < 0 || y < 0) return;
    if(x > XMAX || y > YMAX) return;
    this.x = x;
    this.y = y;
 }
 public void affiche() {
    System.out.println("("+ x+", "+ y+")");
 }
}
```

Mais, la principale utilisation de `this` est effectuée dans les constructeurs. Elle va permettre d'appeler un constructeur au sein d'un autre constructeur d'une même classe. Dans ce cas particulier, `this` est utilisé comme une méthode et reçoit des arguments. Ainsi, le constructeur dont la signature correspond est appelé. Pour ce cas d'utilisation de `this`, il ne peut y avoir qu'un unique appel de `this`(...) et, dans cas, cet appel doit constituer la première instruction du constructeur appelant.

```
[modificateurs] NomClasse(type1 nom1, ..., typek nomk) {
 this(val1, …, valn) ;
 instructions (sans this(…))
 [return; ]
}
```

Lors de l'appel, le constructeur référencé par `this` est appelé avant que les instructions qui suivent dans le constructeur courant ne soient exécutées. Si cet appel ne constitue pas la première instruction, cela provoque l'erreur de compilation « `call to this must be first statement in constructor` ».

Le gros avantage de l'utilisation de cette écriture est de permettre la factorisation du code au niveau d'un (ou d'un nombre réduit) constructeur(s), afin d'éviter de dupliquer du code inutilement (ce qui augmente les risques d'erreurs et complique la mise à jour du code).

Reprenons, le premier exemple, développé pour le 3.2, dans lequel plusieurs constructeurs de la classe `Point` sont définis. On peut remarquer que le code qui consiste à vérifier que tout point créé se situe dans la bonne zone est dupliqué à plusieurs reprises. Dans cette nouvelle version, le constructeur, le plus général `Point(int nx, int ny)`, est appelé, directement ou indirectement, par tous les autres constructeurs. C'est lui qui contrôle la conformité des valeurs à affecter aux champs `x` et `y`.

`Point(double nx, double ny)`, `Point(int nx)` et `Point(Point p)` appellent `Point(int nx, int ny)`, `Point()` appelle `Point(int nx)`. Les affichages des entêtes de méthodes montrent bien que le constructeur appelé est exécuté avant les instructions de l'appelant. L'appel `p3 = new Point()` provoque celui de `Point(int nx)` qui provoque celui de `Point(int nx, int ny)` qui retourne dans `Point(int nx)` qui ensuite retourne dans `Point()`. La Figure 12 illustre ce qui se passe pour l'appel `p4 = new Point(5.1,6.2)`.

Exemple :

```
public class Test {
 static public void main(String [] args) {
  Point p1, p2, p3, p4, p5;
  p1 = new Point(1,2);
  p2 = new Point(3);
  p3 = new Point();
  p4 = new Point(5.1,6.2);
  p5 = new Point(p1);
  p1.affiche();
  p2.affiche();
  p3.affiche();
  p4.affiche();
  p5.affiche();
 }
}
class Point {
 private int x, y;
 private final int XMAX = 200;
 private final int YMAX = 100;
 public  Point(int nx, int ny) {
  System.out.println("public  Point(int nx, int ny)");
    if(nx < 0 || ny < 0) return;
    if(nx > XMAX || ny > YMAX) return;
    x = nx;
    y = ny;
 }
 public  Point(double nx, double ny) {
  this((int)nx, (int)ny);
  System.out.println("public  Point(double nx, double ny)");
 }
```

```
  public  Point(int nx) {
   this(nx, 0);
   System.out.println("public  Point(int nx)");
  }
  public  Point() {
   this(0);
   System.out.println("public  Point()");
  }
  public Point(Point p) {
   this(p.x, p.y);
   System.out.println("public Point(Point p)");
  }
  public void affiche() {
     System.out.println("("+ x+", "+ y+")");
  }
 }
```

Résultat de l'exécution :

```
public  Point(int nx, int ny)
public  Point(int nx, int ny)
public  Point(int nx)
public  Point(int nx, int ny)
public  Point(int nx)
public  Point()
public  Point(int nx, int ny)
public  Point(double nx, double ny)
public  Point(int nx, int ny)
public Point(Point p)
(1, 2)
(3, 0)
(0, 0)
(5, 6)
(1, 2)
```

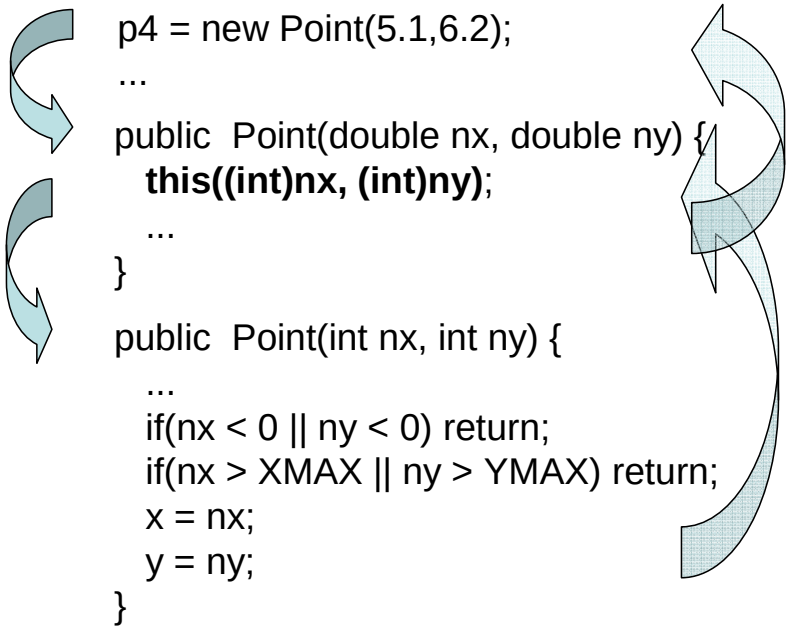


Figure 12 : Appel de constructeur et retour avec this

3.4 Blocs d'initialisation

Java offre une dernière possibilité pour initialiser les champs d'une classe : les blocs d'initialisation. Un bloc d'initialisation est un bloc d'instructions défini dans une classe. Une classe peut en posséder plusieurs.

```
[public] class NomClasse {
 [public | private] définition des champs
     [public | private] définition des méthodes
     {
          Instructions ;
     }
 }
```

Les blocs d'initialisation sont exécutés, lors de la création de chaque nouvel objet, après les initialisations explicites des champs (ou leur initialisation par défaut) et avant l'exécution du ou des constructeurs. Ces blocs peuvent être utilisés pour initialiser des champs ou exécuter n'importe quelles instructions. Lorsqu'une classe possède plusieurs blocs d'initialisations, ils sont exécutés dans leur ordre de définition dans la classe. Il vaut mieux éviter de les utiliser et plutôt privilégier les constructeurs.

L'exemple suivant montre l'ordre d'exécution lors de la création d'un objet de type Point. Là aussi, des affichages sont ajoutés afin d'illustrer le comportement du programme. Lors de la création de l'objet de type `Point new Point(2,2)`, les champs `x` et `y` sont initialisés avec la valeur par défaut `0` (`XMAX` et `YMAX` sont aussi initialisés), puis, le bloc d'initialisation est exécuté, on affecte à `x` et `y` la valeur `1`, puis le constructeur est appelé et on affecte à `x` et `y` la valeur `2`.

Exemple :

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

```
   Point p1;
   p1 = new Point(2,2);
   p1.affiche();
  }
 }
 class Point {
  private int x, y;
  private final int XMAX = 200;
  private final int YMAX = 100;
  {
   affiche();
   x = y = 1;
  }
  public  Point(int nx, int ny) {
   affiche();
   if(nx < 0 || ny < 0) return;
      if(nx > XMAX || ny > YMAX) return;
      x = nx;
      y = ny;
  }

  public void affiche() {
      System.out.println("("+ x+", "+ y+")");
  }
 }
```

Résultat de l'exécution :

```
{...}
(0, 0)
public Point(int nx, int ny)
(1, 1)
(2, 2)
```

4. Tableau d'objets

Un tableau d'objets est un cas particulier de tableau dans lequel le type de base est une classe. Ainsi, les valeurs composantes d'un tableau d'objets sont des références d'objet, mais pas des objets. Les tableaux d'objets se comportent comme des tableaux à deux dimensions.

Le programme suivant décompose les étapes nécessaires pour créer un tableau d'objets de type `Point`, la Figure 13 représente la mémoire après la création complète du tableau d'objets (seuls les champ `x` et `y` des objets de type `Point` sont représentés).

Déclaration d'un tableau (d'objets de type `Point`) :

```
Point []tp;
```

`tp` est la référence d'un tableau d'objets de type `Point`.

Allocation dynamique ou **création** du tableau en utilisant l'opérateur `new`

```
tp = new Point[5];
```

Chaque composante du tableau créé dynamiquement est de type référence d'un objet de type `Point`, ces 5 références valent toutes `null` (valeur par défaut des objets).

Allocation dynamique ou **création** de chaque objet référencé par une composante du tableau utilisant l'opérateur `new`

```
tp[i] = new Point(i, i);
```

Chaque composante du tableau référence un objet de type `Point` qui existe, les champs `x` et `y` de chaque objet sont initialisés par le constructeur de `Point`.

Manipulation d'un objet référencé par une composante du tableau, on sélectionne la composante donc la référence de l'objet correspondant, puis on sélectionne la méthode (ou le champ) avec l'opérateur de sélection de membre « . ».

```
tp[i].affiche();
```

Exemple :

```
public class Test {
 static public void main(String [] args) {
  Point []tp;
  tp = new Point[5];
  int i;
  for(i = 0; i < tp.length; i++)
   tp[i] = new Point(i, i);
  for(i = 0; i < tp.length; i++)
   tp[i].affiche();
 }
}
class Point {
 private int x, y;
 private final int XMAX = 200;
 private final int YMAX = 100;
 public  Point(int nx, int ny) {
  if(nx < 0 || ny < 0) return;
    if(nx > XMAX || ny > YMAX) return;
    x = nx;
    y = ny;
 }
 public void affiche() {
    System.out.println("("+ x+", "+ y+")");
 }
}
```

Résultat de l'exécution :

```
(0, 0)
(1, 1)
(2, 2)
(3, 3)
(4, 4)
```

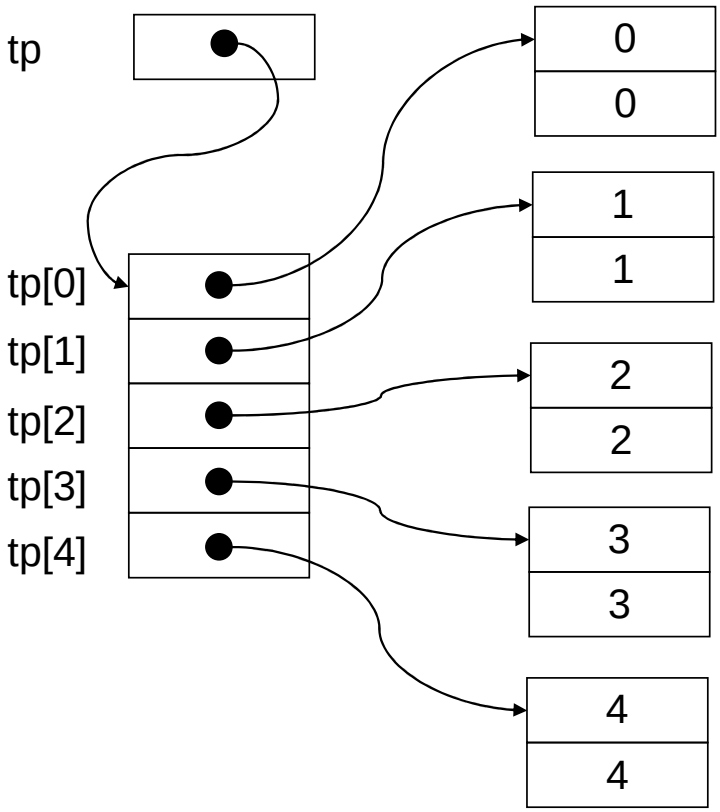


Figure 13 : tableau d'objets après sa création

Il est aussi possible de créer un tableau d'objets en utilisant une liste d'initialisations, comme l'exemple suivant l'illustre :

Exemple :

```
…
p1 = new Point(1,2);
p2 = new Point(2,3);
p3 = new Point(3,4);
Point [] tp = {p1, p2, p3};
…
```

5. Membres statiques

5.1 Champs statiques

Chaque objet d'une classe possède ses propres champs (données membres). Cependant, il peut être intéressant que tous les objets d'une classe possèdent une ou plusieurs données partagées par l'ensemble de ces instances. Par exemple, le nombre d'objets créés de la classe (nombre de produits stockés), le nombre maximum d'objets que l'on peut créer (capacité d'un stock), ... Ces données sont appelées **variables de classe** par opposition aux données membres qui sont propres à chaque objet qui sont qualifiées de **variables d'instance**. Une variable de classe a plusieurs caractéristiques :

- Elle est partagée par toutes les instances de la classe à laquelle elle appartient
- Elle est indépendante d'une instance particulière
- Elle n'existe qu'un seul exemplaire

Une variable de classe, appelée aussi **champ statique**, se déclare, dans une classe `nomClasse` comme un champ, auquel on ajoute le modificateur[17] `static` :

```
static [public | private] [final] type identificateur [= valeur initiale];
```

Le champ statique `identificateur`, s'il est public, peut être manipulé à l'extérieur de sa classe d'appartenance, soit comme un champ non statique : `nomObjet.identificateur` (si bien sur `nomObjet` référence un objet de la classe), ou bien directement `nomClasse.identificateur`. Cette seconde écriture est plus élégante, en effet, ce champ étant commun à tous les objets de la classe, il vaut mieux le désigner par rapport à sa classe que par rapport à un objet particulier.

Dans le programme suivant, la classe `Point` possède deux champs statiques, le premier `nombrePoints` permet de stocker le nombre d'objets de type `Point` créés dans le programme. Pour cela, le constructeur de la classe `Point` incrémente d'une unité la valeur de `nombrePoints` initialisée à `0` (cette initialisation est optionnelle, car un champ variable de type `int` est initialisé à `0`, par défaut qu'il soit statique ou non). La constante statique `NB_PT_MAX`, correspond au nombre maximum d'objets de type `Point` dont on peut fixer les valeurs des champs `x` et `y`. On peut remarquer que la valeur de cette constante est accessible, avant la création du premier objet de type `Point`. Dès que l'on a créé 5 (=`NB_PT_MAX`) objets de type `Point`, les champs `x` et `y` de tous les prochains conserveront leur valeur initiale à savoir `0` (mais ils seront quand même créés !) et `nombrePoints` n'évoluera plus. On peut remarquer que lorsque l'on déclare un `Point` sans créer d'instance, la valeur du champ statique `nombrePoints` n'est logiquement pas modifiée (après la déclaration `Point p2`, `nombrePoints` vaut toujours `1`). On peut constater qu'il est possible de désigner, dans `main`, la constante statique `NB_PT_MAX` soit relativement au nom d'un objet de type Point, soit relativement au nom de la classe (`p1.NB_PT_MAX` ⇔ `Point.NB_PT_MAX`), cette seconde écriture étant plus logique.

Exemple :

```
public class Test {
 static public void main(String [] args) {
  System.out.println(Point.NB_PT_MAX);
  Point []tp;
  Point p1;
  p1 = new Point(1, 2);
  System.out.println(p1.NB_PT_MAX);
  System.out.println(Point.NB_PT_MAX);
  p1.affiche();
  Point p2;
  p1.affiche();
  tp = new Point[5];
  int i;
  for(i = 0; i < tp.length; i++)
   tp[i] = new Point(i, i);
  for(i = 0; i < tp.length; i++)
```

[17] L'ordre des modificateurs (static, public, private, final, …) n'a pas importance, ils doivent cependant précéder le type du membre qu'ils qualifient

```
    tp[i].affiche();
  }
 }
 class Point {
  private int x, y;
  static private int nombrePoints = 0;
  static public final int NB_PT_MAX = 5;
  private final int XMAX = 200;
  private final int YMAX = 100;
  public  Point(int nx, int ny) {
   if(nx < 0 || ny < 0) return;
     if(nx > XMAX || ny > YMAX) return;
     if(nombrePoints >= NB_PT_MAX) return;
     x = nx;
     y = ny;
     nombrePoints ++;
  }

  public void affiche() {
     System.out.println("("+ x+", "+ y+")");
     System.out.println(" nombrePoints = " + nombrePoints);
  }
 }
```

Résultat de l'exécution :

```
5
5
5
(1, 2)
 nombrePoints = 1
(1, 2)
 nombrePoints = 1
(0, 0)
 nombrePoints = 5
(1, 1)
 nombrePoints = 5
(2, 2)
 nombrePoints = 5
(3, 3)
 nombrePoints = 5
(0, 0)
 nombrePoints = 5
```

5.2 Méthodes statiques

Une méthode statique est une méthode qui n'appartient à aucun objet particulier de la classe dans laquelle elle est définie. Une telle méthode peut être appelée à n'importe quel moment, dès lors qu'elle est accessible, même si aucune instance de sa classe d'appartenance n'est créée (comme les champs statiques). Les deux utilisations principales des méthodes statiques sont :

- L'accès et la manipulation des champs statiques

- Des traitements qui ne sont pas liés à un objet particulier, mais qui nécessitent d'accéder aux champs des objets de la classe

Parfois, elle sont aussi utilisées pour accéder à des informations sur la classe.

Une méthode statique ne peut pas manipuler les membres non statiques, méthodes ou champs, ainsi que l'autoréférence `this`, une telle tentative de manipulation conduit à l'erreur de compilation « `non-static variable y cannot be referenced from a static context` ».

Comme pour les champs, il faut ajouter à la liste des modificateurs de la méthode statique le mot clé `static`.

Le programme qui suit illustre la définition et l'utilisation de méthodes statiques. La méthode `afficheNombrePoints` affiche uniquement la valeur du champ statique `nombrePoints`. Cette méthode est appelée au début de la méthode main, alors qu'aucun objet de type `Point` n'est encore créé, on peut l'appeler soit en la préfixant par le nom de la classe `Point.afficheNombrePoints()` (plus élégant), soit par le nom d'une instance `p1.afficheNombrePoints()`. La seconde méthode statique, `coincide(Point pa, Point pb)`, teste si deux objets de type `Point` possèdent les mêmes coordonnées. On passe à cette méthode la référence de deux objets quelconques de type `Point`, elle est bien indépendante d'un objet `Point` particulier. Par contre, comme elle fait partie de la classe `Point`, elle peut accéder aux données privées des instances de cette classe, ici `x` et `y`. On peut comparer cette méthode avec la méthode non statique `coincide(Point p)` qui elle dépend de l'objet courant (ici référencé par `p1`) auquel elle est appliquée. Dans la comparaison `p.x == this.x`, `this` (le `this` est optionnel) référence l'objet courant qui est le même que celui référencé par `p1` dans `main`.

Exemple :

```
public class Test {
 static public void main(String [] args) {
  Point.afficheNombrePoints();
  Point p1, p2, p3;
  p1 = new Point(1, 2);
  p2 = new Point(2, 3);
  p3 = new Point(3, 4);
  Point.afficheNombrePoints();
  p1.afficheNombrePoints();
  System.out.println(Point.coincide(p1, p2));
  System.out.println(p1.coincide(p2));
 }
}
class Point {
 private int x, y;
 static private int nombrePoints = 0;
 private final int XMAX = 200;
 private final int YMAX = 100;
 public  Point(int nx, int ny) {
  if(nx < 0 || ny < 0) return;
  if(nx > XMAX || ny > YMAX) return;
  x = nx;
  y = ny;
```

```
   nombrePoints ++;
  }
  public void affiche() {
     System.out.println("("+ x+", "+ y+")");
  }
  static public void afficheNombrePoints() {
     System.out.println(nombrePoints);
  }
  public boolean coincide(Point p) {
   if(p.x == this.x && p.y == this.y) return true;
   return false;
  }
  static public boolean coincide(Point pa, Point pb) {
   if(pa.x == pb.x && pa.y == pb.y) return true;
   return false;
  }
 }
```

Résultat de l'exécution :

```
0
3
3
false
false
```

5.3 Bloc d'initialisation statique

Java offre la possibilité d'utiliser des blocs d'initialisation statiques. Un bloc d'initialisation est un bloc d'instructions défini dans une classe préfixé par le mot clé `static`. Ces blocs peuvent être utilisés pour effectuer des traitements nécessaires à la classe, comme par exemple initialiser des variables statiques ou tester la disponibilité d'une ressource. Bien évidemment, ils ne peuvent pas accéder aux membres non statiques. Une classe peut posséder plusieurs blocs d'initialisation statiques.

```
[public] class NomClasse {
 [public | private] définition des champs
     [public | private] définition des méthodes
     static {
          Instructions ;
     }
 }
```

Les blocs d'initialisation sont exécutés, lors de la création du premier objet de sa classe de la classe ou dès que l'on utilise une méthode ou un champ statique de sa classe. Un bloc d'initialisation statique ne peut être exécuté qu'une fois (à l'opposé des blocs d'initialisation non statiques qui sont exécutés à la création de tout nouvel objet). Lorsqu'une classe possède plusieurs blocs d'initialisation statiques, ils sont exécutés dans leur ordre de définition dans la classe.

Dans l'exemple qui suit le bloc d'initialisation statique permet d'initialiser à `0` la variable statique `nombrePoints`.

Exemple :

```
public class Test {
 static public void main(String [] args) {
  Point p1, p2, p3;
  p1 = new Point(1, 2);
  p2 = new Point(2, 3);
  p3 = new Point(3, 4);
  Point.afficheNombrePoints();
 }
}
class Point {
 private int x, y;
 static private int nombrePoints;
 static {
  nombrePoints = 0;
 }
 private final int XMAX = 200;
 private final int YMAX = 100;
 public  Point(int nx, int ny) {
  if(nx < 0 || ny < 0) return;
     if(nx > XMAX || ny > YMAX) return;
     x = nx;
     y = ny;
     nombrePoints ++;
 }

 public void affiche() {
     System.out.println("("+ x+", "+ y+")");
 }
 static public void afficheNombrePoints() {
     System.out.println(nombrePoints);
 }
}
```

Résultat de l'exécution :

```
3
```

Lorsqu'une classe contient des parties statiques celles-ci sont toujours exécutées avant les parties non statiques. L'exécution de la partie statique commence lors de la création du premier objet de la classe ou dès que l'on utilise une méthode ou un champ statique. Il faut retenir l'ordre d'exécution :

A la création du premier objet ou à l'utilisation du premier membre statique (une seule fois)

1. Initialisation des champs statiques (explicite ou par défaut)
2. Exécution des blocs d'initialisation statiques dans leur ordre

A chaque création d'un nouvel objet

3. Allocation dynamique de l'objet
4. Initialisation des champs non statiques (explicite ou par défaut)
5. Exécution des blocs d'initialisation non statiques dans leur ordre d'apparition

6. Exécution du ou des constructeurs

Le programme suivant illustre ce processus :

Dès que l'on crée la première instance de la classe `Point` (`ts1 = new TestStatique();`):

1. l'initialisation `vclasse`←1 est exécutée
2. le bloc statique est exécuté `vclasse`←2
3. l'initialisation `vinstance`←10 est exécutée
4. le bloc non statique est exécuté `vinstance`←11
5. le constructeur est exécuté `vclasse`←3 et `vinstance`←12.

Lors des créations suivantes des instances de la classe `TestStatique`, les parties statiques ne sont plus exécutées, seules les parties non statiques le sont.

Si dans la méthode `main`, nous avions mis l'unique instruction : `TestStatique.afficheVclasse()`, le résultat de l'exécution se serait limité à :

```
vclasse = 1
vclasse = 2
```

Seuls les deux premiers item précédents auraient été exécutés avant l'exécution de la méthode statique `afficheVclasse()`.

Exemple :

```
public class Test {
 static public void main(String [] args) {
  TestStatique ts1, ts2, ts3;
  System.out.println("------------------------------");
  ts1 = new TestStatique();
  System.out.println("------------------------------");
  ts2 = new TestStatique();
  System.out.println("------------------------------");
  ts3 = new TestStatique();
  System.out.println("------------------------------");
 }
}
class TestStatique {
 static private int vclasse = 1;
 private int vinstance = 10;
 static {
  afficheVclasse();
  vclasse = 2;
 }
 {
  System.out.println("vinstance = " + vinstance);
  vinstance = 11;
 }
 public TestStatique(){
  afficheVclasse();
  System.out.println("vinstance = " + vinstance);
  vinstance ++;
  vclasse ++;
```

```
   afficheVclasse();
   System.out.println("vinstance = " + vinstance);
  }
  public static void afficheVclasse() {
   System.out.println("vclasse = " + vclasse);
  }
 }
```

Résultat de l'exécution :

```
 ------------------------------
 vclasse = 1
 vinstance = 10
 vclasse = 2
 vinstance = 11
 vclasse = 3
 vinstance = 12
 ------------------------------
 vinstance = 10
 vclasse = 3
 vinstance = 11
 vclasse = 4
 vinstance = 12
 ------------------------------
 vinstance = 10
 vclasse = 4
 vinstance = 11
 vclasse = 5
 vinstance = 12
 ------------------------------
```

5.4 Champs constants statiques ou non statiques

Les champs statiques constants sont des champs précédés des deux modificateurs `final` et `static`. Ces champs peuvent être considérés comme des constantes globales. Du reste, certaines classes de l'API Java en utilisent. Par exemple la constante `Math.PI` a pour valeur une valeur (approchée) de π. Elle est déclarée comme `static final double` (voir 11.3).

Java met à la disposition du programmeur plusieurs possibilités pour définir des champs constants et les initialiser. Nous nous proposons d'en étudier quelques unes et de les comparer.

Reprenons l'exemple de la classe `Point`, dans cette version, tous les points sont inscrits dans un rectangle de même dimension. Or les champs constants `XMAX` et `YMAX` sont dupliqués à chaque fois que l'on crée une nouvelle instance de `Point`. On consomme donc de la mémoire inutilement (peut être problématique dans une grosse application). La seconde classe `PointB` évite cette duplication, les deux champs `XMAX` et `YMAX` sont, cette fois-ci, statiques donc créés en un seul exemplaire (voir Figure 14) quel que soit le nombre d'instances de `PointB`.

Exemple :

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

```
   Point p1, p2;
   p1 = new Point(1, 2);
   p2 = new Point(3, 4);
   PointB pb1, pb2;
   pb1 = new PointB(1, 2);
   pb2 = new PointB(3, 4);
  }
}
class Point {
 private int x, y;
 private final int XMAX = 200;
 private final int YMAX = 100;
 public  Point(int nx, int ny) {
  if(nx < 0 || ny < 0) return;
     if(nx > XMAX || ny > YMAX) return;
     x = nx;
     y = ny;
 }
}
class PointB {
 private int x, y;
 private static final int XMAX;
 private static final int YMAX;
 static {
  XMAX = 200;
  YMAX = 100;
 }
 public  PointB(int nx, int ny) {
  if(nx < 0 || ny < 0) return;
     if(nx > XMAX || ny > YMAX) return;
     x = nx;
     y = ny;
 }
}
```

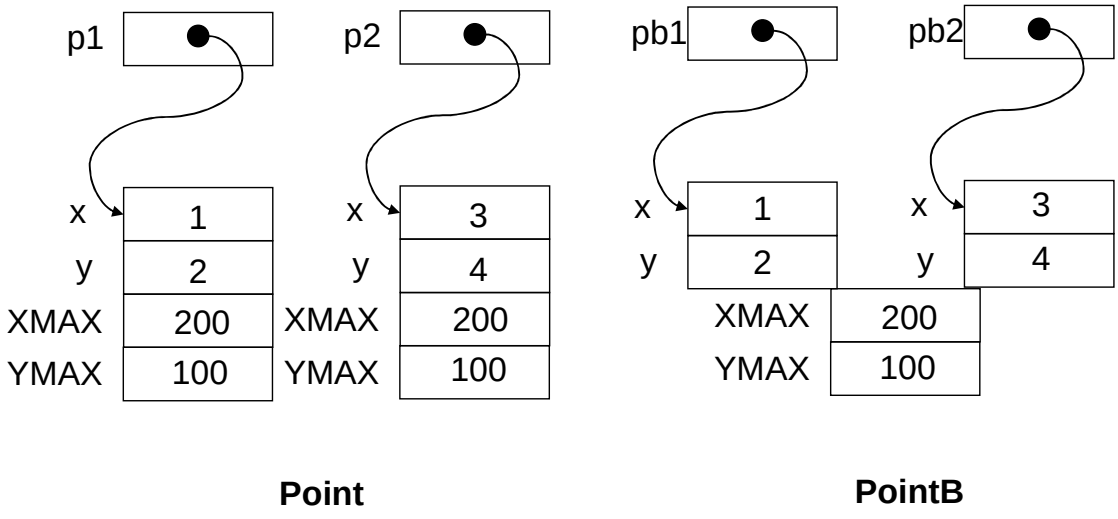


Figure 14 : Avec et sans duplication de constantes

Cependant, l'utilisation de champs statiques non constants peut se justifier, si leur valeur est dépendante de l'objet. Maintenant, si on décide que nos points sont circonscrits à un rectangle, mais que ce rectangle dépend du point, donc il doit être défini à la création de chaque nouvel objet de type `Point`. L'exemple suivant propose une classe `Point` supportant cette nouvelle définition, les champs constants `XMAX` et `YMAX` sont alors initialisés dans le constructeur de `Point`. Comme il s'agit d'une initialisation différée, il faut bien s'assurer que `XMAX` et `YMAX` sont, dans tous les cas initialisés, même si les valeurs des arguments passés ne conviennent pas. Ici nous avons choisi `200` pour le champ `XMAX` et `200` pour le champ `YMAX`.

Exemple :

```
public class Test {
 static public void main(String [] args) {
  Point p1, p2;
  p1 = new Point(1, 2, 100, 200);
  p2 = new Point(3, 4, 150, 225);
 }
}
class Point {
 private int x, y;
 private final int XMAX;
 private final int YMAX;
 public  Point(int nx, int ny, int nXmax, int nYmax) {
  if(nXmax < 0 || nYmax < 0) {
   XMAX = 200;
   YMAX = 100;
   return;
  }
  XMAX = nXmax;
  YMAX = nYmax;
  if(nx < 0 || ny < 0) return;
    if(nx > XMAX || ny > YMAX) return;
    x = nx;
    y = ny;
 }
}
```

6. Ecriture standard d'une classe

Dans cette partie, nous allons nous intéresser à la structure type d'une classe Java. Il s'agit avant tout de définir un certain nombre de règles et de conventions d'écriture. Leur application permet d'éviter un certain nombre d'écueils, d'améliorer la lisibilité et la maintenabilité des classes développées et, par ailleurs, de faciliter leur intégration et leur utilisation dans d'autres programmes. Par ailleurs, cela permet de changer le codage des méthodes sans changer leurs entêtes donc la manière dont elles sont appelées : même interface et implémentation différente.

6.1 Typologie des méthodes

On distingue plusieurs types de méthodes, faisant partie de l'interface de la classe (on l'appelle `uneClasse`), qui doivent avoir une entête standard :

Les **constructeurs** qui permettent d'initialiser les nouveaux objets lors de leur création. On doit définir le constructeur par défaut qui ne reçoit aucun paramètre, il doit avoir pour entête :

- `Public uneClasse()`

Le constructeur de recopie a comme unique paramètre un objet de la classe ; il affecte aux champs de l'objet courrant la valeur des champs correspondants de l'objet passé, il doit avoir pour entête :

- `Public uneClasse(uneClasse obj)`

Bien évidemment, généralement la classe possède d'autres constructeurs. Il faut privilégier un « constructeur général » qui effectue tous les traitements de validation et d'initialisation effective des champs non statiques de la classe, il sera appelé par les autres constructeurs via un `this(…)`. Généralement, on choisit le constructeur qui possède des paramètres correspondant aux champs de la classe (en particulier de même type).

Les **altérateurs** ou modificateurs ou mutateurs, ce sont des méthodes qui modifient la valeur d'un ou plusieurs champs de l'objet courant. Elles doivent vérifier que l'objet reste dans un état cohérent. Lorsque les méthodes d'altération modifient le champ `XXX` de type `T`, le modificateur doit avoir pour entête :

- `public void setXXX(T nomParamètre)`.

Si le champ est statique, le modificateur `static` doit être ajouté.

Les **accesseurs** ou observateurs ou interrogateurs, ce sont des méthodes qui retournent la valeur d'un champ de l'objet courant, sans le modifier[18]. Lorsque que ces méthodes retournent la valeur du champ `XXX` de type `T` (non booléen), l'accesseur doit avoir pour entête :

- `public T setXXX()`,

si le `XXX` est de type booléen doit avoir pour entête :

- `public boolean isXXX()`.

Si le champ est statique, le modificateur `static` doit être ajouté.

On définit souvent d'autres méthodes particulièrement utiles qui font aussi partie de l'interface (elles sont, en particulier, nécessaires si on désire exploiter nos classes avec certaines API Java). On verra dans la suite, lorsque l'on abordera l'héritage (la classe `Object`) et la notion d'interface (`Clonable`, `Comparable`, `Comparator`) que la forme de ces méthodes divergera légèrement de ce que nous proposons.

On peut définir une méthode de **clonage** dont l'objectif est de retourner une copie de l'objet courant, qui doit avoir pour entête :

- `public uneClasse clone()`

On peut définir des **comparateurs** permettant de comparer en profondeur des objets (si cela a un sens), en effet l'opérateur `==` ne fait que comparer la valeur de leurs références, il est en général nécessaire de comparer les champs des objet. La première méthode de comparaison retourne `true` si tous les champs de l'objet passé sont égaux à ceux de l'objet courant, `false` sinon, elle doit avoir pour entête :

[18] On verra dans la suite que quelques précautions s'imposent dans le cas des champs qui ne sont pas de type primitifs

- `boolean equals(uneClasse obj)`

On peut définir deux autres méthodes de comparaison qui retournent une valeur entière. Elles permettent également de comparer si un objet est plus petit ou plus grand qu'un autre (si cela a un sens). On doit s'assurer que si `equals` retourne `true` (égalité), alors, ces méthodes doivent retourner `0`.

La première retourne une valeur positive si l'objet courant est « plus grand » que l'objet passé, `0` si les deux objets sont égaux, une valeur négative sinon, elle doit avoir pour entête :

- `int compareTo(uneClasse obj)`

La seconde retourne une valeur positive si le premier objet passé est « plus grand » que le second objet passé, `0` si les deux objets sont égaux, une valeur négative sinon. Elle est statique car indépendante d'un objet particulier. elle doit avoir pour entête :

- `static int public compare(uneClasse obj1, uneClasse obj1)`

6.2 Exemple de la classe Point

Nous reprenons la version de la classe `Point` représentant un ensemble de points se trouvant tous contenus dans le même rectangle dont les dimensions sont stockées dans les champs constants statiques `XMAX` et `YMAX`.

Nous avons défini trois constructeurs :

- `public Point()` : constructeur par défaut, sans paramètres
- `public Point(Point p)` : constructeur de recopie
- `public Point(int nx, int ny)` : constructeur général appelé par les autres constructeurs et qui vérifie la validité des coordonnées des points créés avant de les affecter aux champs.

Deux altérateurs publiques permettant de mettre à jour les coordonnées `x` et `y` des points créés sont définis, ces altérateurs vérifient la validité de la coordonnée concernée. Bien évidement, aucun altérateur concernant les constantes `XMAX` et `YMAX` n'est défini :

- `public void setX(int nx)` : mise à jour du champ `x`
- `public void setY(int ny)` : mise à jour du champ `y`

Quatre accesseurs publiques permettant d'obtenir les coordonnées `x` et `y` des points créés et les dimensions du rectangle contenant les points sont définis. Les accesseurs concernant les constantes statiques `XMAX` et `YMAX` sont déclarés `static` :

- `public int getX()` : retourne la valeur du champ `x`
- `public int getY()` : retourne la valeur du champ `y`
- `static public int getXMAX()` : retourne la valeur champ `XMAX`
- `static public int getYMAX()` : retourne la valeur champ `YMAX`

Nous avons défini la méthode de clonage publique qui retourne une copie de l'objet auquel elle est appliquée. Cette méthode utilise le constructeur de recopie `new Point(this)` afin de créer une nouvelle instance de `Point` identique au `Point` courant (mêmes valeurs des champs, mais référence différente). Dans main, on peut remarquer que `p5`, le clone de `p2` a bien une référence différente (`@addbf1`≠`@190d11`) et pourtant son contenu est identique à celui de `p2` (`p2.equals(p5)`→`true`) :

- `public Point clone()` : retourne un clone de l'objet courant

Enfin, nous avons définie trois comparateurs, celui d'égalité (`equals`) ne fait que tester l'égalité des champs `x` et `y` de l'objet auquel il est appliqué avec ceux de l'objet passé en paramètre. Les deux autres utilisent la distance euclidienne comme méthode de comparaison. Elles appellent la méthode statique `Math.abs(…)` de la classe `Math` qui retourne la valeur absolue de la valeur qui lui est passée[19] :

- `public boolean equals(Point p)` : vérifie l'égalité du point courant et du point passé
- `public int compareTo(Point p)` : compare le point courant avec le point passé
- `static public int compare(Point p1, Point p2)` : compare les deux points passés.

Afin de s'assurer de l'identité comportementale entre ces deux dernières méthodes, on aurait pu définir, par exemple, la méthode `compare` de la façon suivante :

```
static public int compare(Point p1, Point p2) {
  return p1.compareTo(p2);
 }
```

Exemple :

```
public class Test {
 static public void main(String [] args) {
  Point p1, p2, p3, p4, p5;
  p1 = new Point(1,2);
  p2 = new Point(2,3);
  p3 = new Point(p1);
  p4 = new Point();
  p5 = p2.clone();
  System.out.println(p1);
  System.out.println(p3);
  System.out.println(p2);
  System.out.println(p4);
  System.out.println(p5);
  System.out.println(p1.equals(p3));
  System.out.println(p1.equals(p2));
  System.out.println(p2.equals(p5));
  System.out.println(p1.compareTo(p2));
  System.out.println(Point.compare(p1, p2));
  System.out.println(p1.getX());
  p1.setX(5);
  System.out.println(p1.getX());
  p1.setX(300);
  System.out.println(p1.getX());
  System.out.println(Point.getXMAX());
 }
}
class Point {
```

[19] Cette méthode surchargée existe en plusieurs exemplaires : int, float, double, …

```
  private int x, y;
  private static final int XMAX = 200;
  private static final int YMAX = 100;
  public  Point(int nx, int ny) {
   if(nx < 0 || ny < 0) return;
     if(nx > XMAX || ny > YMAX) return;
     x = nx;
     y = ny;
  }
  public  Point() {
   this(0, 0);
  }
  public Point(Point p) {
   this(p.x, p.y);
  }
  public void translate(int dx, int dy) {
   int nx = x + dx;
     int ny = y + dy;
     if(nx < 0 || ny < 0) return;
     if(nx > XMAX || ny > YMAX) return;
     x = nx;
     y = ny;
  }
  public Point clone() {
   return new Point(this);
  }
  public boolean equals(Point p) {
   if(p.x == x && p.y == y) return true;
   return false;
  }
  public int compareTo(Point p) {
   return Math.abs(x*x + y*y) - Math.abs(p.x*p.x + p.y*p.y);
  }
  static public int compare(Point p1, Point p2) {
   return Math.abs(p1.x*p1.x + p1.y*p1.y) - Math.abs(p2.x*p2.x +
p2.y*p2.y);
  }
  public int getX() {
   return x;
  }
  public int getY() {
   return y;
  }
  static public int getXMAX() {
   return XMAX;
  }
  static public int getYMAX() {
   return YMAX;
  }
  public void setX(int nx) {
```

```
   if(nx < 0 || nx > XMAX) return;
   x = nx;
  }
  public void setY(int ny) {
   if(ny < 0 || ny > YMAX) return;
   y = ny;
  }
}
```

Résultat de l'exécution :

```
Point@19821f
Point@addbf1
Point@42e816
Point@9304b1
Point@190d11
true
false
true
-8
-8
1
5
5
200
```

7. Objet, champs mutables et dissimulation

Un objet est « **mutable** » s'il est possible de modifier son état après sa création. C'est le cas de la classe `Point` que nous avons défini précédemment dont les champs ne sont pas déclarés `final` et qui possède des altérateurs. On peut définir des objets dits « non mutables » ou « **immuables** » qui, une fois créés, ne peuvent plus être modifiés, en particulier, ils ne possèdent aucune méthode d'altération.

Le fait d'utiliser des objets mutables pose des difficultés par rapport au principe de dissimulation et nécessite un mode de programmation rigoureux. Nous allons commencer par un exemple introductif.

Considérons l'exemple suivant dans lequel sont définis des cercles dont les caractéristiques, le centre et le rayon, sont fixés une fois qu'ils sont créés. Le champ `centre` est une référence d'un objet `Point` et le champ `rayon` est un entier de type `int`. On cherche donc à créer des cercles non mutables, les champs sont déclarés `final` et aucun altérateur n'est défini. Dès lors, on pourrait penser que c'est suffisant, il n'en est rien. On peut remarquer que si on modifie l'objet de type `Point`, référencé par `p1`, qui sert à construire l'objet de type `Cercle`, référencé par `c1`, l'objet référencé par le champ `centre` est modifié (`p1.setX(2)`). De même, si on affecte la référence du champ `centre` à une autre référence d'objet de type `Point` (`p2 = c1.getCentre()`) qu'on modifie (`p2.setY(2)`), l'objet référencé par le champ `centre` est modifié. Par contre, les opérations analogues sur le champ `rayon` ne modifient pas la valeur de ce dernier. Ce qui se passe est normal, c'est la valeur de la référence du champ `centre` qui ne peut pas être modifiée, cependant l'objet référencé lui peut l'être (on a rencontré un cas similaire avec les objets passés aux méthodes voir 2.3). La Figure 15 montre la configuration mémoire à

la fin de l'exécution de `main`, dans laquelle, `c1.centre`, `p1` et `p2` référencent le même objet de type `Point`.

La rupture du principe de dissimulation des champs privés va au-delà du fait que l'on désire qu'un ou plusieurs champs d'un objet ne soient pas modifiés. En effet, même si le champ `centre` n'était pas déclaré `final`, on n'aurait dû pouvoir y accéder, et en particulier le modifier, uniquement au travers des méthodes de la classe `Cercle`, ce qui n'est pas le cas. Le problème aurait été identique avec la déclaration du champ centre de `Cercle` suivante :

```
class Cercle {
 private Point centre;
…
```

Exemple :

```
public class Test {
 static public void main(String [] args) {
  int r1 = 10;
  Point p1 = new Point(1,1);
  Cercle c1 = new Cercle(p1, r1);
  c1.print();
  r1 = 11;
  c1.print();
  p1.setX(2);
  c1.print();
  int r2 = c1.getRayon();
  Point p2 = c1.getCentre();
  r2 = 12;
  p2.setY(2);
  c1.print();
 }
}
class Point {
 private int x, y;
 public  Point(int nx, int ny) {
    x = nx;
    y = ny;
 }
 public  Point() {
  this(0, 0);
 }
 public Point(Point p) {
  this(p.x, p.y);
 }
 public Point clone() {
  return new Point(this);
 }
 public void print() {
    System.out.println("("+ x+", "+ y+")");
 }
 public void setX(int nx) {
```

```
   x = nx;
  }
  public void setY(int ny) {
   y = ny;
  }
  public int getX() {
   return x;
  }
  public int getY() {
   return y;
  }
}
class Cercle {
 private final Point centre;
 private final int rayon;
 public Cercle(Point nCentre, int nRayon) {
  rayon = nRayon;
  centre = nCentre;
 }
 public Point getCentre() {
  return centre;
 }
 public int getRayon() {
  return rayon;
 }
 public void print() {
  centre.print();
  System.out.println(rayon);
 }
}
```

Résultat de l'exécution :

```
(1, 1)
10
(1, 1)
10
(2, 1)
10
(2, 2)
10
```

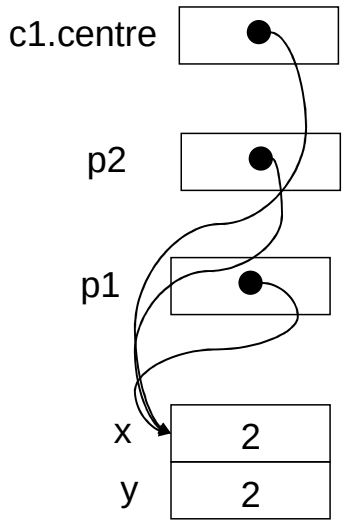


Figure 15 : Cercle mutable

Donc le modificateur `final` devant les champs de type objet et l'absence d'altérateurs concernant ces champs ne sont pas suffisants. La première solution consiste à rendre la classe `Point` immuable (en déclarant les champs `x` et `y` final et en supprimant les altérateurs). Cependant, il arrive fréquemment que l'on soit amené à utiliser des classes toutes faites ou simplement que l'on veuille que l'accès à des données privées non constantes ne puissent pas violer le principe d'encapsulation. Pour contourner le problème, on utilise le principe dit de « copie défensive ». Lors de l'initialisation d'un champ de type objet, on lui affecte la référence d'une copie de l'objet dont la référence est passée au constructeur. De même, l'accesseur concernant un champ de type objet doit retourner la référence d'une copie. Pour faciliter la mise en œuvre de cette approche, la classe de l'objet champ doit posséder un cloneur et un constructeur de recopie.

Dans l'extrait de programme qui suit, nous avons redéfini la classe `Cercle`, l'objet correspondant au champ `centre` est initialisé avec une copie de celui passé dans le constructeur, on utilise le constructeur de recopie de la classe `Point` (`centre = new Point(nCentre)`). L'accesseur, quant à lui, retourne la référence d'une copie de l'objet correspondant au champ `centre`, la méthode `clone` de `Point` est appelée (`return centre.clone()`). En remplaçant, la classe `Cercle` dans le programme précédent, le résultat de l'exécution est conforme à ce que nous voulions. La Figure 16 montre la configuration mémoire à la fin de l'exécution de `main`, dans laquelle `c1.centre`, `p1` et `p2` référencent chacune un objet distinct de type `Point`. La configuration est bien différente de celle du cas précédent.

Exemple (programme partiel) :

```
class Cercle {
 private final Point centre;
 private final int rayon;
 public Cercle(Point nCentre, int nRayon) {
  rayon = nRayon;
  centre = new Point(nCentre);
 }
 public Point getCentre() {
  return centre.clone();
 }
```

```
public int getRayon() {
 return rayon;
}
public void print() {
 centre.print();
 System.out.println(rayon);
}
```

Résultat de l'exécution :

```
(1, 1)
10
(1, 1)
10
(1, 1)
10
(1, 1)
10
```

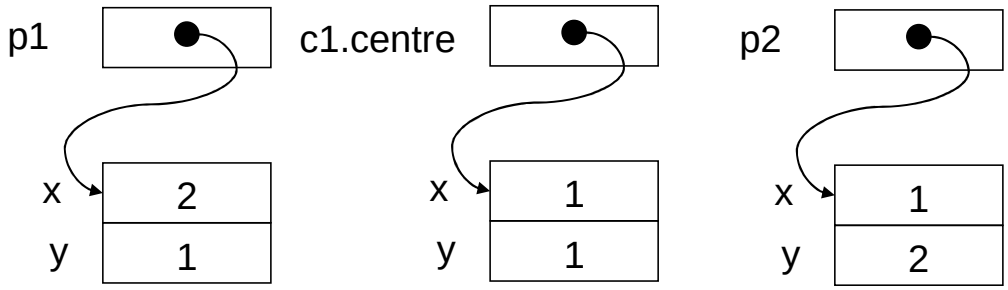


Figure 16 : Cercle non mutable

8. Classes internes

Une **classe interne** (inner class) ou **classe imbriquée** est une classe qui est définie à l'intérieur d'une autre classe (outer class), qualifiée de **classe externe**. Une classe interne n'est qu'une déclaration, ces champs ne sont pas ajoutés à ceux de sa classe englobante (comme pour les structures en langage C), il faut l'instancier pour créer des objets.

Comme une classe est encapsulée dans une classe externe, elle peut donc accéder même aux membres privés de celle-ci. En général, ce sont des classes relativement simples dont l'objectif est d'offrir des fonctionnalités qui n'ont pas de sens dans un contexte indépendant de la classe externe. Ce mécanisme permet de définir une classe à l'endroit où cela est nécessaire et de la cacher à l'extérieur de la classe externe, il peut être considéré comme un mécanisme d'encapsulation. Bien que généralement, une classe interne hérite d'une classe ou implémente une interface, notions que nous verrons lorsque nous aborderons l'héritage, nous les étudierons dans cette partie. En Java, il existe quatre catégories de classe interne :

- Non statique
- Statique
- Anonyme**,** étudiée dans la partie relative à l'héritage (§9 p208)
- Locale (nommée)

Interne classe non statique	Interne classe statique	Interne classe locale à une méthode

`class A {`
` …`
` class B {`
`  …`
` }`
` …`
`}`
	`class A {`
	` …`
	` static class B {`
	`  …`
	` }`
	` …`
	`}`
		`class A {`
		` …`
		` typeR maMethode(…) {`
		`  class B {`
		`  …`
		`  }`
		` }`
		` …`
		`}`

Tableau 7 : Types de classes internes (hors anonyme)

8.1 Classe interne non statique

Une classe interne non statique est une classe interne qui est définie au même niveau que les membres de cette classe et qui n'a pas de modificateur `static`.

Une classe interne est schématiquement déclarée :

```
[public] class outerClasse {
 [public | private] définition des champs
     [public | private] définition des méthodes
     [public | private] class innerClasse {
          [public | private] définition des champs
          [public | private] définition des méthodes
     }
 }
```

Les droits d'accès ont les mêmes caractéristiques que ceux des membres (en cas d'omission de droits d'accès, il s'agit d'un accès « friendly »), mais il s'agit d'une définition de classe qui ne peut être utilisée que pour déclarer des objets. Tout objet d'une classe interne non statique possède la référence de l'objet classe englobante. Une classe interne non statique ne peut pas posséder de membres statiques.

Dans l'exemple suivant, une classe `Pile` est incluse à la classe `Point` afin d'historiser les anciennes valeurs des coordonnées des objets de type `Point` et de pouvoir les récupérer (système très simple d'annulation suite à une erreur). Pour réaliser cela, la classe interne `Pile` sert à créer un champ `stock` (contenant les anciennes valeurs des champs `x` et `y`). A chaque fois que les champs d'un objet de type `Point` sont modifiés par la méthode `set`, les valeurs courantes de ces champs sont empilées par la méthode `push` de la classe `Pile` avant d'effectuer la mise à jour. Pour récupérer les dernières valeurs de ces champs, il suffit d'appeler la méthode `undo` de `Point` qui utilise la méthode `pop` de la classe `Pile` qui affecte aux champs `x` et `y` leurs anciennes valeurs après les avoir dépilées. La classe interne Pile a accès aux champs privés `x` et `y`. De même, les méthodes de la classe `Point` accèdent aussi aux membres de la classe `Pile` `push` et `pop` par l'intermédiaire du champ stock, bien que la classe `Pile` soit privée (normal c'est le même niveau d'imbrication), l'accès aurait été possible même si ces méthodes avaient été privées (ce qui peut sembler moins logique car ces méthodes sont d'un niveau d'imbrication supérieur).

Exemple :

```
public class Test {
 static public void main(String [] args) {
  Point p = new Point(1,1);
  int i;
  for(i = 0; i < 3; i++){
   p.set(i+2,i+2);
  }
  p.print();
  p.undo();
  p.print();
  p.undo();
  p.print();
  for(i = 5; i < 6; i++){
   p.set(i+2,i+2);
  }
  p.print();
  p.undo();
  p.print();
  p.undo();
  p.print();
 }
}
class Point {
 private int x, y;
 private Pile stock;
 public  Point(int nx, int ny) {
    x = nx;
    y = ny;
    stock = new Pile(10);
 }
 private class Pile {
  private int [][]txy;
  private int index;
  public Pile(int dim) {
   txy = new int[dim][2];
   index = 0;
  }
  public boolean push(){
   if(index == txy.length) return false;
   txy[index][0] = x;
   txy[index][1] = y;
   index ++;
   return true;
  }
  public boolean pop(){
   if(index == 0) return false;
   index --;
   x = txy[index][0];
```

```
    y = txy[index][1];
    return true;
   }
  }
  public void print() {
     System.out.println("("+ x+", "+ y+")");
  }
  public void set(int nx, int ny) {
   stock.push();
   x = nx;
   y = ny;
  }
  public boolean undo() {
   return stock.pop();
  }
 }
```

Résultat de l'exécution :

```
(4, 4)
(3, 3)
(2, 2)
(7, 7)
(2, 2)
(1, 1)
```

Il est possible de référencer l'objet de classe englobante dans une classe interne. Dans la classe interne, `this` constitue la référence de l'objet courant de la classe interne. Pour désigner la référence de la classe externe, il faut le préfixer par le nom de celle-ci suivi du séparateur de champ « . » : `outerClasse.this`. Lorsque l'on fait référence un membre de la classe interne, il est implicitement préfixé (ce mécanisme de désignation peut être utilisé pour lever des ambiguïtés de nom : membres homonymes de la classe interne et de la classe englobante).

Ainsi, si on reprend l'exemple précédent, la méthode `push` aurait pu s'écrire comme dans l'extrait suivant où :

- `Point.this.x` ⇔ `x` : champ `x` de la classe encapsulante
- `this.index` ⇔ `index` : champ `index` de la classe interne courante

Exemple (extrait) :

```
public boolean push(){
   if(index == txy.length) return false;
   txy[this.index][0] = Point.this.x;
   txy[this.index][1] = Point.this.y;
   index ++;
   return true;
 }
```

Nous aurions pu reprendre l'exemple de la classe `Cercle` (§7 p97), dans laquelle la déclaration de la classe `Point` utile pour la définition du `centre` aurait pu être cachée en la rendant interne (traité en exercice §12.3 p148).

Il est possible de créer des objets instances d'une classe interne. Il faut nécessairement que la classe interne ne soit pas déclarée `private` (sinon elle est inaccessible à l'extérieur de sa classe externe). Leur utilisation est assez peu pertinente, car une autre classe peut jouer avantageusement le même rôle qu'une classe interne publique. En reprenant l'exemple précédent, il suffit de déclarer une variable de type `Pile`, comme la classe `Pile` n'est pas connue à l'extérieur de `Point`, il faut la désigner par rapport à la classe `Point` en utilisant le séparateur de champ « . » :

- `Point.Pile pi;` déclaration d'un objet de type `Pile` d'un objet de type `Point`.

Puis pour créer un objet de type `Pile`, il faut nécessairement créer préalablement un objet de type Point :

- `Point pt = new Point(1,1);`

Puis créer un objet de type `Pile` par rapport à l'objet de type `Point` déjà créé, en utilisant le séparateur de champ « . » :

- `pi = pt.new Pile(5);` création de l'objet `pi` de type `Pile` de l'objet `pt` de type `Point`.

L'objet référencé par `pi` est distinct de l'objet référencé par `stock`. L'objet référencé par `pi` est associé à l'objet `pt`, ainsi, les méthodes `push` et `pop` de `pi` accèdent directement aux champs `x` et `y` de l'objet de type `Point` référencé par `pt`, de la même façon que `stock`. L'utilisation des classes internes publiques doit l'être dans un contexte où leur utilisation est pertinente tant leur mise en œuvre syntaxique et comportementale peut être déroutante.

Exemple (extrait) :

```
public class Test {
 static public void main(String [] args) {
  Point.Pile pi;
  Point pt = new Point(1,1);
  pi = pt.new Pile(5);
  pt.print();
  pi.push();
 }
}
class Point {
 private int x, y;
 private Pile stock;
 …
 public class Pile {
  …
```

8.2 Classe interne statique

Il est possible de définir une classe interne statique, comme on définit une classe interne, à la condition que cette classe ne manipule pas de champs non statiques de sa classe externe. Une classe interne statique est découplée des membres d'instances qu'elle ne peut pas manipuler (bien évidemment, l'inverse est possible), elle ne peut pas non plus en manipuler la référence. Un objet d'une classe interne statique peut exister sans qu'aucune instance de sa classe englobante n'ait été crée et il ne peut avoir accès à

la référence d'un objet de sa classe englobante. La classe interne `Pile` n'aurait pas pu être statique car elle manipulait les variables d'instance `x` et `y` de sa classe externe.

Dans l'exemple suivant, la classe interne statique `Max` a pour objet de calculer et stocker la distance de l'objet de type `Point` le plus éloigné de l'origine. Pour cela, la classe englobante `Point` possède un champ statique `max` référençant un objet de type `Max`. Dans la classe `Max`, la distance est stockée par le champ entier `dmax` qui est mis à jour par l'altérateur `setDMax` et dont la valeur peut être lue par l'accesseur `getDMax`. Dès qu'un objet de type `Point` est créé ou modifié, l'altérateur `setDMax` est appelé qui, éventuellement, met à jour le champ `dmax` (donc indirectement `max`).

Dans l'exemple qui suit, on a directement récupéré la référence du champ statique `max` de `Point` en l'affectant à la variable locale de `main` `max`.

- `Point.Max max = Point.getMax() ;`

Concernant les déclarations et les créations, comme le contexte est statique et que la classe `Max` est publique, il aurait été possible de créer des objets de type `Pile` sans qu'il existe préalablement d'objets de type `Point` :

- `Point.Max m`; déclaration (de la référence) d'un objet de type `Max` de la classe `Point`.
- `m = new Point.Max();` création de l'objet `m` de type `Max` de la classe `Point`.

Exemple :

```
public class Test {
 static public void main(String [] args) {
  Point.Max max = Point.getMax();
  System.out.println(max.getDMax());
  Point p = new Point(1,1);
  System.out.println(max.getDMax());
  Point p2 = new Point(2,3);
  System.out.println(max.getDMax());
  p.set(4,3);
  System.out.println(max.getDMax());
 }
}
class Point {
 private int x, y;
 static Max max = new Max();
 static public class Max {
  private int dmax ;
  public Max() {
   dmax = 0;
  }
  public int getDMax() {
   return dmax;
  }
  public void setDMax(int nx, int ny) {
   int d = (int)Math.sqrt(nx*nx + ny*ny);
   if(d > dmax) dmax = d;
  }
 }
```

```
  public  Point(int nx, int ny) {
     x = nx;
     y = ny;
     max.setDMax(x, y);
  }
  public static Max getMax(){
   return max;
  }
  public static void setMax(Max nmax){
   max = nmax;
  }
  public void print() {
     System.out.println("("+ x+", "+ y+")");
  }
  public void set(int nx, int ny) {
   x = nx;
   y = ny;
   max.setDMax(x, y);
  }
 }
```

Résultat de l'exécution :

```
0
1
3
5
```

Une classe interne statique peut être utilisée pour déclarer des champs non statiques dans la classe qui l'encapsule. Sa seule limitation est qu'elle ne peut pas manipuler les membres ou la référence d'un objet courant. L'exemple suivant, purement syntaxique, illustre quelques possibilités :

`private Inner outin ;` : une classe externe peut posséder des champs non statiques du type de sa classe interne statique

`return new Inner(n);` : une classe externe peut créer des objets du type de sa classe interne statique

`public int methodeInner(Outer out)` : une classe interne peut manipuler (référencer, créer, modifier, …) des objets du type de sa classe externe (elle pourrait même posséder des champs de ce type)

`System.out.println(intChamp + " " + cpt);` : une classe interne peut manipuler les membres statiques de sa classe externe

Exemple :

```
public class Test {
 static public void main(String [] args) {
  Outer.Inner nin = new Outer.Inner(0);
  nin.print();
  Outer nout = new Outer(1);
  nout.print();
  nin = nout.methodeOuter(3);
  nin.print();
```

```
   nin = nout.methodeOuter();
   nin.print();
  }
 }
 class Outer {
  private int outChamp;
  private Inner outin;
  private static int cpt = 0;
  static public class Inner {
   private int intChamp;
   public Inner(int n) {
    intChamp = n;
   }
   public int methodeInner(Outer out) {
    return out.outChamp;
   }
   public void print() {
    System.out.println(intChamp + " " + cpt);
   }
  }
  public Outer(int n) {
   outChamp = n;
   outin = new Inner(n);
   cpt++;
  }
  public Inner methodeOuter(int n) {
   return new Inner(n);
  }
  public Inner methodeOuter() {
   return outin;
  }
  public void print() {
    System.out.println(outChamp);
  }
 }
```

Résultat de l'exécution :

```
0 0
1
3 1
1 1
```

8.3 Classe interne locale

Dans ce dernier cas, il s'agit de déclarer une classe dans une méthode. La classe ne peut être utilisée que pour déclarer des variables locales à la méthode (comme elle est dissimulée, les paramètres et les valeurs de retour ne peuvent pas être concernés). Une classe locale à une méthode ne possède pas de droit d'accès et elle ne peut pas être statique.

La classe locale peut accéder à tous les membres de la classe englobante même privés. Par contre, seule la méthode dans laquelle la classe locale est définie peut accédé

aux membres, même privés, de celle-ci. La classe locale ne peut accéder qu'aux variables locales déclarées `final` de la méthode dans laquelle elle est déclarée.

L'utilisation de ce type de classe est plutôt rare. L'exemple suivant illustre leur mise en oeuvre. La classe locale `Format` de la méthode `print` a uniquement pour objet de mettre en forme les champs `x` et `y` des objets de type `Point` avant leur affichage. Elle accède bien aux champs privés x et y de la classe encapsulante. Par ailleurs, bien que le champ `s` soit privé, il aurait été possible de substituer à l'instruction `System.out.println(nf.getS())`l'instruction `System.out.println(s)`, la méthode `print` pouvant accéder même aux champs privés de sa classe locale `Format`.

Exemple :

```
public class Test {
 static public void main(String [] args) {
  Point p1 = new Point(1, 1);
  Point p2 = new Point(2, 2);
  Point p3 = new Point(3, 3);
  p1.print(0);
  p2.print(1);
  p3.print(2);
 }
}
class Point {
 private int x, y;
 public  Point(int nx, int ny) {
    x = nx;
    y = ny;
 }
 public void set(int nx, int ny) {
  x = nx;
  y = ny;
 }
 public void print(int tf) {
  class Format {
   private String s;
   public Format(int tFormat) {
    if (tFormat == 0) s = "["+ x+"|"+ y+"]";
    else if (tFormat == 1) s = "<"+ x+"..."+ y+">";
    else s = "("+ x+", "+ y+")";
   }
   public String getS(){
    return s;
   }
  }
  Format nf = new Format(tf);
    System.out.println(nf.getS());
 }
}
```

Résultat de l'exécution :

```
[1|1]
```

```
<2...2>
(3, 3)
```

9. Enumération

Depuis Java 5.0, le mot clé `enum` permet de créer des **énumérations**. Jusqu'à cette version, il fallait contourner cette absence, par exemple, en définissant, des classes contenant des valeurs constantes statiques.

Un type `enum` en Java est un type composé d'un ensemble fixe d'objets constants (`static` et `final`), mais potentiellement mutables, ces éléments uniques sont désignés par des valeurs symboliques, ces valeurs sont implicitement publiques. Les objets de l'énumération sont créés en même temps que l'énumération elle-même. On ne peut, du reste, ajouter aucun modificateur devant cette liste de valeurs. Donc, une fois définie, on ne peut pas ajouter de nouveaux objets à l'énumération ou changer la valeur symbolique référençant un de ses objets. De plus, tous les objets de l'énumération sont distincts deux à deux.

En Java, les types énumérés sont beaucoup plus complets que leurs homologues dans d'autres langues, en effet, ce sont des classes particulières, dites de type énuméré, qui peuvent posséder des méthodes et des champs complémentaires à la liste des valeurs symboliques. Par contre, ces classes ont quelques limites, elles peuvent uniquement être utilisées comme des classes internes statiques, elles ne peuvent pas être dérivées (elles sont `final`, voir partie sur l'héritage), elles sont obligatoirement publiques ou « friendly ».

Lors de la création d'une énumération, le compilateur synthétise (ajoute automatiquement) certaines méthodes spécifiques qui peuvent être exploitées (extraction d'une valeur, de la liste de toutes les valeurs, …).

La forme la pus simple d'une énumération est la suivante :

```
[public] enum Enumeration {
 VAL1, VAL2,…, VALI …, VALN;
}
```

Par convention, les valeurs symboliques : `VAL1, VAL2, …, VALN` sont en majuscules, comme toutes les constantes en Java. Le seul modificateur qui peut qualifier une énumération est le modificateur `public`.

Si on duplique une des valeurs de la liste cela provoque l'erreur de compilation « `VALI is already defined in Jour` »

Les méthodes synthétisées associées à l'énumération `Enumeration` sont :

- `public int ordinal()` : retourne une valeur entière unique qui désigne de manière unique chaque constante (les constantes sont classées dans l'ordre de leur création, la première a comme ordinal 0, la seconde 1, ….)
- `public String name()` : retourne une chaîne de caractères dont la valeur correspond à l'identificateur de la constante (on obtient le même résultat avec `toString`)
- `public static Enumeration []values(String)` : retourne la référence de l'objet de l'énumération dont la valeur correspond à la chaîne de caractères passée
- `public static Enumeration values()` : retourne la référence d'un tableau contenant toutes les références des objets de l'énumération (valeurs de l'énumération)

Un constructeur par défaut est aussi synthétisé.

Les objets de l'énumération sont désignés par des valeurs symboliques. Ainsi, pour définir la référence d'un objet d'une énumération, il ne faut pas utiliser l'opérateur new (une énumération ne peut pas être instanciée), mais directement affecter à cette référence la valeur de la constante préfixée par le nom de l'énumération, en utilisant le séparateur de champ « . ». En conséquence, pour pouvoir utiliser une énumération, il faut que l'énumération possède au moins une valeur (l'absence de valeur ne provoque pas d'erreur de compilation, mais rend l'énumération inutile). Ainsi, pour déclarer une référence et la définir, on peut opérer des deux façons :

`Enumeration a = Enumeration.VALI;` : affectation directe

Ou

```
Enumeration a;
…
a = Enumeration.VALI; : affectation différée
```

Le principe est le même que pour l'affectation d'une constante statique à une variable :

- `double x = Math.PI ;`.

Des objets de type énuméré peuvent être comparés avec les opérateurs == (on obtient le même résultat avec `equals`) et `!=`.

L'exemple suivant illustre l'utilisation de l'énumération `Jour` constituée de la liste des jours de la semaine. Pour créer une variable de type `Jour` (`Jour j`), il faut la déclarer, et lui affecter une valeur de l'énumération (`j= Jour.LUNDI;`), `j` constitue bien la référence de la constante `LUNDI` de `Jour` (du reste, l'instruction `j = null`; est parfaitement licite). Par contre, il est impossible d'instancier un `Jour`, l'instruction `j = new Jour();` provoque l'erreur de compilation « `enum types may not be instantiated` ». On peut remarquer que les valeurs sont numérotées dans leur ordre d'apparition, à partir de 0, (`LUNDI` → `0`, `MARDI` → `1`, … `DIMANCHE` → `6`). Si on duplique une des valeurs de la liste, par exemple `SAMEDI`, cela provoque l'erreur de compilation « `SAMEDI is already defined in Jour` ». On peut remarquer l'utilisation de la boucle « for each » pour afficher le tableau dont les valeurs correspondent aux références des champs de l'énumération.

Exemple :

```
public class Test {
   static public void main(String [] args) {
   Jour j = Jour.LUNDI;
   System.out.println(j.name());
   j = Jour.MARDI;
   System.out.println(j.ordinal());
   Jour []tj = Jour.values();
   for(Jour ja : tj) {
    System.out.println(ja.name() + " : " +ja.ordinal());
    if(ja == j) System.out.println("   c'est j !!!");
   }
   String s= j.name();
   System.out.println(Jour.valueOf(s));
  }
}
```

```
enum Jour {
 LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE;
}
```

Résultat de l'exécution :

```
LUNDI
1
LUNDI : 0
MARDI : 1
   c'est j !!!
MERCREDI : 2
JEUDI : 3
VENDREDI : 4
SAMEDI : 5
DIMANCHE : 6
```

Java permet d'utiliser les énumérations dans les `switch`, de manière particulière. Les valeurs des constantes dans les `case` ne doivent pas être préfixées par le nom de l'énumération, sinon cela provoque l'erreur de compilation : « `an enum switch case label must be the unqualified name of an enumeration constant` ». Il faut considérer cette possibilité comme une simplification d'écriture (déroutante vis-à-vis de l'encapsulation !).

```
Jour j = Jour.LUNDI;
switch(j) {
 case Jour.LUNDI : … // erreur de compilation
 …
}
```

Il faut donc écrire

```
Jour j = Jour.LUNDI;
switch(j) {
 case LUNDI : …
 …
}
```

Alors que dans les autres cas (différents de `case`), il faut obligatoirement préfixer la valeur par le nom de l'énumération : `if(j == LUNDI)` provoque l'erreur de compilation : « `cannot find symbol variable LUNDI` ».

Il est possible d'ajouter des membres à l'énumération : des champs et des méthodes qui peuvent être précédés de modificateurs (`static`, `public`, `final`, …) sauf les constructeurs qui doivent être « friendly » ou privés (si on les déclare `public`, cela déclenche l'erreur de compilation « `modifier public not allowed here` »). En effet, il est impossible de créer une instance d'énumération.

Lorsque l'on associe aux valeurs symboliques d'une énumération des valeurs qui correspondent aux caractéristiques de chaque objet qui la compose, il faut définir un constructeur compatible qui est automatiquement appelé pour initialiser l'énumération (en cas d'absence cela provoque l'erreur de compilation « `cannot find symbol constructor Enumeration(type_1, type_2)` »). Si le nombre de caractéristiques diffèrent ou qu'elles sont de types distinctes, il faut prévoir d'autres constructeurs compatibles.

```
[public] enum Enumeration {
```

```
     VAL1(val11, val12), …, VALN(valn1, valn2);
     [private] Enumeration(type_1 v1, type_2 v2) {
      …
     }
    }
```

Pour manipuler ces valeurs, il faut prévoir des champs de type compatible pour les désigner. L'association champ valeur s'effectue au travers du constructeur.

Pour illustrer ces possibilités, nous allons définir une énumération correspondant à l'ensemble des planètes du système solaire. Chaque planète est caractérisée par sa masse et son rayon équatorial qui correspondent aux deux champs constant `masse` et `rayon`. Ces valeurs sont transmises au constructeur lorsque l'objet constant correspondant est créé. La Figure 17 illustre la création de l'objet `TERRE` de l'énumération `Planete`, lorsque le compilateur lit : `TERRE (5.976e+24, 6.37814e6)`, il appelle automatiquement le constructeur en lui transmettant les deux valeurs que le constructeur va affecter aux champs `masse` et `rayon`.

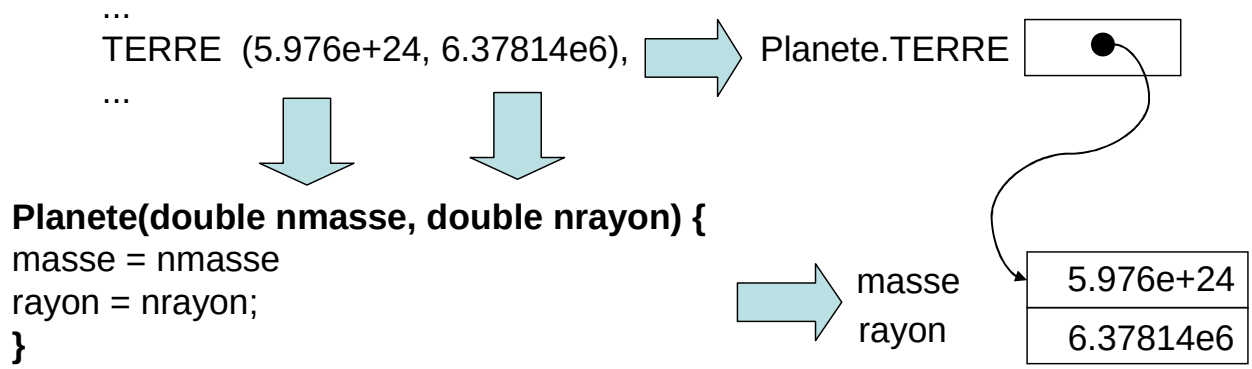


Figure 17 : Création d'un objet de l'énumération

En plus de ses propriétés et du constructeur, la classe `Planete` possède des méthodes qui permettent de calculer la gravité en surface : `graviteEnSurface`, le poids d'un objet en surface : `poidsEnSurface` et la masse volumique de la planète : `masseVolumique` (tous les résultats sont donnés dans le système SI).

Exemple :

```
public class Test {
 public static void main(String[] args) {
        Planete p = Planete.TERRE;
        System.out.println(p.graviteEnSurface());
        System.out.println(p.poidsEnSurface(80));
        System.out.println(p.masseVolumique());
        p = Planete.JUPITER;
        System.out.println(p.graviteEnSurface());
        System.out.println(p.poidsEnSurface(80));
        System.out.println(p.masseVolumique());
        Planete []tp = Planete.values();
        for(Planete pp :tp)
          System.out.println(pp);
    }
}
```

```
enum Planete {
    MERCURE (3.303e+23, 2.4397e6),
    VENUS   (4.869e+24, 6.0518e6),
    TERRE  (5.976e+24, 6.37814e6),
    MARS    (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27,   7.1492e7),
    SATURNE  (5.688e+26, 6.0268e7),
    URANUS  (8.686e+25, 2.5559e7),
    NEPTUNE (1.024e+26, 2.4746e7);

    private final double masse;
    private final double rayon;
    public static final double G = 6.67300E-11;

    private Planete(double nmasse, double nrayon) {
        masse = nmasse;
        rayon = nrayon;
    }
    private double masse()   { return masse; }
    private double rayon() { return rayon; }
    double graviteEnSurface() {
        return G * masse / (rayon * rayon);
    }
    double poidsEnSurface(double masse) {
        return masse * graviteEnSurface();
    }
    double masseVolumique() {
     return masse / (4. / 3. * rayon * rayon * rayon * Math.PI);
    }
    public String toString() {
      return name() + " " + masse + " " + rayon;
    }
}
```

Résultat de l'exécution :

```
9.802652743337129
784.2122194669703
5498.444379312064
24.80617666947324
1984.4941335578592
1241.3454352805627
MERCURE 3.303E23 2439700.0
VENUS 4.869E24 6051800.0
TERRE 5.976E24 6378140.0
MARS 6.421E23 3397200.0
JUPITER 1.9E27 7.1492E7
SATURNE 5.688E26 6.0268E7
URANUS 8.686E25 2.5559E7
NEPTUNE 1.0239999999999999E26 2.4746E7
```

Il aurait été possible de déclarer les champs `masse` et `rayon` non constants, ils auraient alors été modifiables (dans ce cas, l'objet aurait été mutable), par contre la référence de l'objet planète, désigné par sa valeur symbolique, lui reste constant.

Supposons que la masse d'une planète soit inconnue, par exemple Jupiter, dans ce cas, l'objet correspondant aurait eu comme spécification : `JUPITER (7.1492e7)`, le constructeur prévu n'est pas compatible (erreur de compilation « `cannot find symbol constructor Planete(double)` »), il faut en prévoir un second qui pourrait être (`Double.NaN` signifie « Not a Number » voir §11.2.3 p130) :

```
  private Planete(double nrayon) {
         this(Double.NaN, nrayon);
  }
```

10.Paquetages

Une application Java peut être constituée par des centaines de classes (des énumérations ou des interfaces). Les paquetages (`package`) Java permettent de structurer les applications en ensembles de classes logiquement ou fonctionnellement reliées (structuration laissée à l'appréciation du concepteur). Un paquetage définit un espace de noms : deux classes peuvent porter le même nom dès lors qu'elles appartiennent à des paquetages différents. L'API du langage Java est structurée de la sorte.

Lorsque rien n'est précisé, toutes les classes se trouvant dans un même répertoire (dossier) appartiennent au même paquetage, le paquetage par défaut ou paquetage anonyme. Dans ce paquetage, les membres non privés sont accessibles par toutes les méthodes du même paquetage. Mais seules les méthodes publiques des classes publiques le sont à l'extérieur de ce paquetage. Or, Java impose une contrainte forte sur l'accès aux classes, il n'y a qu'une classe publique par fichier (qui porte le même nom que la classe publique). Une structuration par fichier est trop limitée.

A l'exception des classes définies dans le paquetage anonyme et des classes définies dans le `package java.lang`., tous les membres de toutes les autres classes sont inaccessibles quel que soit leur droit d'accès. Pour rendre une classe accessible, il faut l'importer ou importer toutes les classes de son `package`.

Le mécanisme de gestion des paquetages va plus loin, car il offre aussi un mécanisme de désignation des classes, une classe n'est pas seulement désignée par son nom mais aussi par le package à laquelle elle appartient. On peut ainsi définir deux classes de même nom dans des paquetages différents. Mais il faut garantir des noms de paquetages tous distincts deux à deux.

Par exemple la classe `Random` qui fait partie du paquetage `package java.util` ne peut pas être utilisée si on ne le spécifie pas (l'absence de la commande d'importation provoque l'erreur de compilation : « `cannot find symbol class Random` »). La désignation effective de cette classe n'est pas `Random` mais `java.util.Random`. Pour le spécifier, il faut utiliser le mot clé `import` suivi du nom complet de la classe ou directement nommer complètement la classe. Le tableau suivant montre ces deux possibilités.

```
import java.util.Random;
public class Test {
 static public void main(String []
```

```
public class Test {
     static public void main(String
[] a) {
```

```
a) {
   Random r = new Random();
  }
}
```

```
java.util.Random r =
     new java.util.Random();
    }
}
```

Tableau 8 : Importation ou nommage relatif

Concernant l'importation, Java offre une facilité d'écriture qui permet d'importer toutes les classes d'un package en utilisant à la place du nom de la classe le joker « * ». C'est en général cette écriture qui est utilisée.

Exemple :

```
import java.util.*;
public class Test {
 static public void main(String [] a) {
   Random r = new Random();
  }
}
```

L'utilisation du nommage relatif n'est utilisé que pour lever des ambiguïtés de désignation (éviter les collisions de nom). Dans l'exemple qui suit il faut différencier la classe locale `Random` et la classe `Random` du package `java.util` (l'utilisation d'un simple `import` provoquerait l'erreur de compilation : « `Random is already defined in this compilation unit` »). Ici, il n'y a pas de problème, `r` désigne bien l'objet de type `java.util.Random()`et `ra` un objet de type `Random` (classe locale).

Exemple :

```
public class Test {
 static public void main(String [] a) {
   java.util.Random r = new java.util.Random();
   Random ra = new Random();
  }
}
class Random {
}
```

Il existe un autre mécanisme d'importation depuis le Java 1.5, l'import statique, qui permet d'importer un membre ou tous les membres publiques et statiques d'une classe. Il permet d'alléger l'écriture lors de l'utilisation de variables ou de méthodes de classe en omettant le nom de classe associée. Bien évidemment, si on abuse de cette facilité d'écriture, on provoque des ambiguïtés de noms (collisions de noms). Le tableau suivant illustre la différence entre une version de code Java sans l'utilisation d'un import statique et une autre avec. A la différence d'un import non statique, le nom de la classe est précisé ; dans le cas où on se limite à un membre statique, il faut là encore tout préciser, par exemple ; pour importer uniquement `Math.PI`, il faut écrire `import static java.lang.Math.PI;`.

```
    .
public class Test {
 static public void main(String [] a) {
```

```
import static java.lang.Math.*;
public class Test {
```

```
                                                 static public void main(String
System.out.println(Math.cos(Math.PI/2));  [] a) {
  }
}                                                System.out.println(cos(PI/2));
                                                   }
                                                 }
```

Tableau 9 : Import statique

Le concepteur de classe peut définir ces propres packages. Il doit s'assurer de plusieurs choses :

- Le nom du package est distinct de celui de packages existants (au moins ceux utilisés dans l'application cible)
- Le répertoire dans lequel les classes du package sont regroupées a le même nom que celui du package
- Chaque classe du package doit avoir en entête « package nomDuPackage ; ».

Ainsi à un paquetage qui correspond à une structuration logique, on associe une structuration physique au travers du nom du répertoire.

A titre d'exemple, supposons que l'on veuille définir une bibliothèque de classes permettant de définir et gérer des objets géométriques. Il va regrouper tous les fichiers contenant les classes dans le répertoire (ou dossier) `geometrie`. Chaque fichier devra posséder en entête la spécification de package `package geometrie;`. Toutes les classes définies dans ce répertoire ont automatiquement accès aux membres non privés des autres classes du `package geometrie`.

```
package geometrie;          package geometrie;      package geometrie;
public class Point {        public class Cercle {   public class Carre {
…                           …                       …
public void methode() {     }                       }
 …
 }
}
```

Tableau 10 : Importation ou nommage relatif

Toutes les classes externes auront accès aux membres publiques des classes publiques du `package geometrie` si elles l'importent. C'est le cas de la classe `Test` qui peut manipuler la méthode `methode` publique de la classe `Point`.

Exemple :

```
import geometrie.*;
public class Test {
 static public void main(String[] args)  {
  Point p = new Point();
```

```
    p.methode();
  }
 }
```

Un package peut être constitué de sous packages, les sous packages correspondant alors physiquement à des sous répertoires du répertoire du package. Si un package `nomPackage` rassemble plusieurs sous-packages, la clause import `nomPackage.*` n'importe pas tous les sous-packages. Il faut explicitement importer chacun d'eux (avec par exemple import `nomPackage.nomSousPackage1.*`).

Lors de l'importation d'un package, le compilateur recherche dans les packages de l'API et dans les sous répertoires du répertoire du fichier courant. Pour élargir la recherche, on peut utiliser une variable système `classpath` spécifiant l'ensemble des chemins prédéfinis à utiliser. Ce chemin peut être spécifié lors de la compilation, certains environnements permettent de le spécifier pour chaque projet.

11.Classes de base de l'API Java

L'API Java se compose de nombreuses classes regroupées par fonctionnalité en packages. Ces packages offrent des services couvrant la gestion de fichiers, la conception d'interfaces graphiques, l'exploitation des bases de données, l'exploitation des réseaux, l'animation en 3D, …

Dans l'ensemble de ces packages, il existe un package particulièrement important, qui est automatiquement importé dans les programmes Java, le `package java.lang`. Il contient les classes fondamentales. Dans cette partie, nous détaillerons certaines de ces classes. Nous nous limitons à des classes que l'on peut pratiquement considérer comme des types de données ou à des utilitaires basiques.

11.1 Chaînes de caractères

Les classes `String` et `StringBuffer` faisant partie du package `java.lang`, elles sont automatiquement importées dans les programmes Java. On verra, du reste, dans la suite, qu'elles sont utilisées implicitement.

Une chaîne de caractères est une succession de caractères. Ces chaînes sont, dans certains langages, traitées comme de simples tableaux de caractères. En Java, les chaînes sont représentées par des objets de type `String` ou `StringBuffer`. Les objets de type `String` sont construits et initialisés automatiquement par le compilateur quand il rencontre une chaîne littérale dans le programme source.

L'opérateur « + » permet de concaténer les chaînes de type `String`. Du reste, lorsque l'on utilise cet opérateur entre une chaîne de caractères et une variable d'un type primitif, il y a conversion automatique de l'opérande de type primitif en chaîne de caractères, puis concaténation. On peut considérer les chaînes comme faisant partie des types primitifs du langage, hormis le fait que les chaînes sont des objets.

Le programme suivant illustre les possibilités de la concaténation et le mécanisme de promotion automatique associé. Dans le premier cas, il s'agit d'une simple concaténation de trois chaînes de caractères. Dans les autres cas, on concatène une chaîne avec des entiers convertis en chaînes juste avant la concaténation. Il faut faire attention aux règles d'associativité, de priorité et de conversion lors de l'évaluation des expressions correspondantes.

- "1 + 1 = " + 1 + 1 → ("1 + 1 = " + 1) + 1→"1 + 1 = + 1" + 1→"1 + 1 = 11"
- "1 + 1 = " + (1 + 1) →"1 + 1 = " + 2→"1 + 1 = 2"
- 1 + 1 + " = 1 + 1"→(1 + 1) + " = 1 + 1"→2 + " = 1 + 1"→"2 = 1 + 1"

- "2 * 2 = " + 2 * 2→"2 * 2 = " + (2 * 2) →"2 * 2 = " + 4→"2 * 2 = 4"

Exemple :

```
public class Test {
 static public void main(String [] args) {
  System.out.println("bonjour " + "le " + "monde");
  System.out.println("1 + 1 = " + 1 + 1);
  System.out.println("1 + 1 = " + (1 + 1));
  System.out.println(1 + 1 +  " = 1 + 1" );
  System.out.println("2 * 2 = " + 2 * 2);
 }
}
```

Résultat de l'exécution :

```
bonjour le monde
1 + 1 = 11
1 + 1 = 2
2 = 1 + 1
2 * 2 = 4
```

11.1.1 String (chaînes immuables)

La classe `String` est la classe privilégiée pour manipuler des chaînes de caractères en Java. C'est une classe dont les objets sont immuables, une fois créés et initialisés par un constructeur, ils ne peuvent plus changer de valeur. Cette classe dispose d'une quinzaine de constructeurs et de plusieurs dizaines de méthodes.

Le programme suivant illustre l'utilisation de quelques méthodes de la classe `String`. Il existe des méthodes très surchargées, comme les constructeurs et la méthode statique de conversion `valueOf`. On retrouve des méthodes de comparaison, de recherche et d'extraction (caractères et sous chaînes). Il faut remarquer que les objets de type `String` ne sont jamais modifiés (immuables).

- `String()` → **`String()`** : le constructeur par défaut qui construit une chaîne vide.
- `String("Vive la Mécanique")`→ **`String(String)`**: le constructeur de recopie qui est implicitement appelé lors l'initialisation d'un objet de type `String` avec une chaîne littérale
- `String(tc)` → **`String(char[] value)`** : le constructeur qui initialise une chaîne avec un tableau de caractères
- `s2.length()`→ **`int length()`**: retourne la longueur de la chaîne de caractères
- `s2.charAt(4)` → **`char charAt(int)`** : retourne le caractère dont l'indice est passé en argument (ici celui d'indice 4, donc le cinquième caractère)
- `s4.compareTo("ABC")` → **`int compareTo(String)`**: retourne le résultat de la comparaison lexicographique. Si les deux chaînes sont égales, `0` est retourné sinon c'est la différence des codes des premiers caractères différents qui est renvoyée (celui de la chaîne courante moins celui de la chaîne passée)

- `s4.compareToIgnoreCase("ABC")`→ **`int compareToIgnoreCase(String)`** : retourne le même résultat que `compareTo`, mais sans prendre en compte la casse.
- `s4.equals("ABC")`→ **`boolean equals(Object)`** : retourne `true` si la chaîne courante et la chaîne passée sont identiques, `false` sinon.
- `"AbC".equalsIgnoreCase(s4)` **`boolean equalsIgnoreCase(String)`** : retourne le même résultat que `equals`, mais sans prendre en compte la casse.
- `s2.indexOf('t')` → **`int indexOf(int)`** : retourne la position de la première occurrence du code du caractère passé, `-1` si le caractère n'est pas trouvé
- `s2.indexOf("une")` → **`int indexOf(String)`**: retourne la position de la première occurrence de la chaîne de caractères passée, `-1` si cette chaîne n'est pas trouvée
- `s2.isEmpty()`→ **`boolean isEmpty()`** : retourne `true` si la chaîne est vide, `false` sinon
- `s3.replace('e', 'a')` → **`String replace(char, char)`** : retourne une chaîne égale à la chaîne courante, mais en remplaçant toutes les occurrences du premier caractère passé par le second caractère passé
- `s3.toCharArray()`→ **`char[] toCharArray()`** : retourne un tableau de caractères constitué des caractères de la chaîne
- `s3.toLowerCase()`→ **`String toLowerCase()`** : retourne une chaîne de caractères égale à la chaîne courante convertie en minuscules
- `s3.toUpperCase()`→ **`String toUpperCase()`** : retourne une chaîne de caractères égale à la chaîne courante convertie en majuscules
- `" Avada Kedavra ".trim()`→ **`String trim() :`** retourne une chaîne de caractères égale à la chaîne courante en supprimant les espaces de début et de fin.
- `String.valueOf(new char[] {'a', 'e', 'i', 'o', 'u', 'y'})` → **`static String valueOf(char[] data)`**: retourne une chaîne correspondent au tableau de caractères passé (méthode statique)
- `String.valueOf(3.141159)` → **`static String valueOf(double d)`**: retourne une chaîne correspondant à la valeur du réel de type double passé (méthode statique)

Exemple :

```
public class Test {
 static public void main(String [] args) {
  int i;
  char []tc = {'a', 'b','c'};
  String s1, s2, s3, s4, sa;
  s1 = new String();
  s2 = "Supméca est une école d'ingénieur";
  s3 = new String("Vive la Mécanique");
  s4 = new String(tc);
  System.out.println(s1);
  System.out.println(s2);
```

```
   System.out.println(s3);
   System.out.println(s4);
   System.out.println(s1.length());
   System.out.println(s2.length());
   System.out.println(s2.charAt(4));
   System.out.println(s4.compareTo("ABC"));
   System.out.println(s4.compareToIgnoreCase("ABC"));
   System.out.println(s4.equals("ABC"));
   System.out.println("AbC".equalsIgnoreCase(s4));
   System.out.println(s2.indexOf('t'));
   System.out.println(s2.indexOf("une"));
   System.out.println(s2.isEmpty());
   System.out.println(s1.isEmpty());
   System.out.println(s3.replace('e', 'a'));
   tc = s3.toCharArray();
   for(i = 0; i< tc.length; i++)
    System.out.print(tc[i]);
   System.out.println();
   System.out.println(s3.toLowerCase());
   System.out.println(s3.toUpperCase());
   System.out.println("   Avada Kedavra   ".trim());
   System.out.println(String.valueOf(new char[] {'a', 'e', 'i', 'o',
'u', 'y'}));
   System.out.println(String.valueOf(3.141159));
  }
}
```

Résultat de l’exécution :

```
Supméca est une école d'ingénieur
Vive la Mécanique
abc
0
33
é
32
0
false
true
10
12
false
true
Viva la Mécaniqua
Vive la Mécanique
vive la mécanique
VIVE LA MÉCANIQUE
Avada Kedavra
aeiouy
3.141159
```

11.1.2 StringBuffer (chaînes mutables)

Le défaut principal de la classe `String` vient de son caractère immuable. Si des chaînes de caractères doivent subir de nombreuses transformations, utiliser des chaînes de caractères de type `String` nécessitent des créations inutiles : créer une nouvelle chaîne pour stocker le résultat de la transformation. En conséquence, les opérations sur ces chaînes entraînent la création de chaînes temporaires et des opérations coûteuses en temps d'exécution.

Ces chaînes sont aussi implicitement utilisées par le compilateur Java, lorsque l'on effectue des concaténations de chaînes de caractères. Le code Java suivant illustre l'équivalence qui consiste à effectuer la concaténation en utilisant la méthode `append` (qui concatène à la chaîne courante la chaîne passée en argument) de la classe `StringBuffer` puis la conversion du résultat en `String` :

```
String s1 = "bonjour ", s2 = "Monsieur ", s3 = "Java", s4;
s4 = s1+s2+s3;
s4 = new StringBuffer(s1).append(s2).append(s3).toString();
```

Les objets de la classe `StringBuffer` contiennent des chaînes de caractères variables, la taille effective est gérée de manière transparente pour l'utilisateur. Les objets de cette classe possède une propriété relative à la capacité de stockage de la chaîne et une autre qui correspond à la longueur effective de la chaîne. A chaque fois que l'on ajoute des éléments à la chaîne et que la nouvelle taille va dépasser la valeur de la capacité, une capacité supérieure (de l'ordre du double) est allouée, ce mécanisme limite le nombre d'opérations d'allocation dynamique (`new`), mais consomme plus de mémoire que nécessaire. Les objets de cette classe sont donc particulièrement utiles pour manipuler des chaînes de caractères dont la taille varient souvent. Concernant les méthodes non statiques qui retournent un objet de type `StringBuffer`, l'objet retourné correspond à la référence de l'objet courant.

- `StringBuffer()` → **`StringBuffer()`** : le constructeur par défaut qui construit une chaîne vide de capacité initiale de 16 caractères
- `String("Vive la Mécanique")`→ **`StringBuffer (String)`**: le constructeur qui initialise un objet de type `StringBuffer` avec une chaîne de caractère de type `String`
- `StringBuffer(10)`→ **`StringBuffer(int)`**: le constructeur qui initialise un objet de type `StringBuffer` de capacité égal à la valeur passée en argument
- `s1.capacity()` → **`int capacity() :`** retourne la capacité de la chaîne de caractères courante
- `s1.length()` → **`int length()`**: retourne la longueur de la chaîne de caractères courante
- `s3.insert(i, (char)('c' + i))` → **`StringBuffer insert(int, char)`** : insère en i^{ème} (premier argument) position le caractère passé en second argument (si la chaîne n'est pas vide, les caractères suivant sont décalés vers la droite)
- `s2.indexOf("et")` → **`int indexOf(String)`**: retourne la position de la chaîne de caractères passée, `-1` si cette chaîne n'est pas trouvée.
- `s2.insert(i,"ou")` → **`StringBuffer insert(int, String)`** insère en i^{ème} (premier argument) position la chaîne de caractères passée en second argument, les caractères qui suivent sont décalés vers la droite

- `s2.delete(n, m)` → **`StringBuffer delete(int, int)`** : supprime de la chaîne de caractères courante les caractères se trouvant entre la position dont la valeur correspond au premier argument et la position dont la valeur correspond au second argument-1
- `s2.reverse()`→ **`StringBuffer reverse()`** : inverse l'ordre des caractères de la chaîne de caractères courante
- `s2.toString()`→ **`String toString()`** : Retourne la référence de la chaîne elle même
- `s4.lastIndexOf("ique")` → **`int lastIndexOf(String)`** retourne la position de la dernière occurrence de la chaîne de caractères passée, `-1` si cette chaîne n'est pas trouvée
- `s4.indexOf("ique")` → **`int indexOf(String)`** : retourne la position de la première occurrence de la chaîne de caractères passée, `-1` si cette chaîne n'est pas trouvée
- `s4.setCharAt(0, 'V')` → **`void setCharAt(int , char)`** remplace le caractère dont la position correspond au premier argument par le caractère correspondant au second argument
- `s4.setLength(10)` → **`void setLength(int)`** : force la longueur de la chaîne de caractères courante à la longueur passée en argument, tous les caractères dont la position est supérieure ou égal à cette valeur sont détruits
- `s4.delete(3, 6)` → **`StringBuffer delete(int, int )`** : supprime tous les caractères compris entre les deux positions passées en arguments
- `s4.deleteCharAt(3)` → **`StringBuffer deleteCharAt(int)`** : supprime le caractère dont la position est passée en argument
- `s4.charAt(3)` → **`char charAt(int)`** : retourne le caractère dont l'indice est passé en argument.
- s4.append(s3) → **`StringBuffer append(StringBuffer)`** : cette méthode concatène la chaîne courante avec la chaîne dont la référence est passée en argument, il existe plusieurs versions de cette méthode obtenues par surcharge (plusieurs sont utilisées ici)

Exemple :

```
public class Test {
 static public void main(String [] args) {
  int i;
  char []tc = {'a', 'b','c'};
  StringBuffer s1, s2, s3, s4;
  s1 = new StringBuffer();
  s2 = new StringBuffer("Vive la Mécanique");
  s3 = new StringBuffer(10);
  System.out.println(s1);
  System.out.println(s2);
  System.out.println(s3);
  for(i = 0; i < tc.length; i ++)
   s1.append(tc[i]);
  s2.append(" et l'informatique");
  System.out.println(s1);
```

```
   System.out.println(s2);
   System.out.println(s3);
   System.out.println(s1.capacity() + " " +s1.length());
   System.out.println(s2.capacity() + " " +s2.length());
   System.out.println(s3.capacity() + " " +s3.length());
   int l = s3.capacity();
   for(i = 0; i < l; i ++)
    s3.insert(i, (char)('c' + i));
   System.out.println(s3);
   i = s2.indexOf("et");
   s2.insert(i,"ou");
   System.out.println(s2);
   int n = i + "ou".length(), m = i +  "et".length()+ "ou".length();
   s2.delete(n, m);
   System.out.println(s2);
   s2.reverse();
   System.out.println(s2);
   String sa = s2.toString();
   s4 = new StringBuffer(sa.toLowerCase());
   s4.reverse();
   System.out.println(s4);
   System.out.println(s4.lastIndexOf("ique"));
   System.out.println(s4.indexOf("ique"));
   s4.setCharAt(0, 'V');
   System.out.println(s4);
   s4.setLength(10);
   System.out.println(s4);
   s4.delete(3, 6);
   System.out.println(s4);
   s4.deleteCharAt(3);
   System.out.println(s4);
   System.out.println(s4.charAt(3));
   s3 = new StringBuffer("à Supmeca depuis ");
   s4 = new StringBuffer("Vive la mécanique ");
   s4.append(s3);
   s4.append(50);
   s4.append(' ');
   s4.append('a');
   s4.append('n');
   s4.append('s');
   System.out.println(s4);
  }
 }
```

Résultat de l’exécution :

```
 Vive la Mécanique

 abc
 Vive la Mécanique et l'informatique
```

```
16 3
68 35
10 0
cdefghijkl
Vive la Mécanique ouet l'informatique
Vive la Mécanique ou l'informatique
euqitamrofni'l uo euqinacéM al eviV
vive la mécanique ou l'informatique
31
13
Vive la mécanique ou l'informatique
Vive la mé
Viva mé
Viv mé

Vive la mécanique à Supmeca depuis 50 ans
```

11.1.3 Méthode toString

Cette méthode est une méthode particulière qui retourne une chaîne de caractère représentant un objet. Elle est implicitement appelée lorsque l'on passe la référence d'un objet à une méthode comme `print` ou `println`. En définitive, c'est une méthode de conversion de l'objet en chaîne de caractères. Par défaut, lorsque l'on cherche à afficher la référence d'un objet, c'est son adresse qui est affichée, en réalité une méthode `toString` est appelée par défaut (Lorsqu'on abordera la redéfinition de méthode, ce mécanisme sera détaillé), mais il est conseillé de redéfinir systématiquement cette méthode (cela fait partie de l'écriture standard d'une classe, voir §6 p92).

La version suivante de la classe `Point`, très simplifiée, montre, comme nous l'avons déjà vu, que lorsque l'on passe une référence de `Point` à la méthode `println`, le résultat est la concaténation du nom de la classe et de l'adresse de l'objet passé.

Exemple :

```
public class Test {
 static public void main(String [] args) {
  Point p = new Point(1,1);
  System.out.println(p);
 }
}
class Point {
 private int x, y;
 public Point(int nx, int ny){
  x = nx;
  y = ny;
 }
}
```

Résultat de l'exécution :

```
Point@19821f
```

Dans cette seconde version, la méthode `toString` est définie (en fait redéfinie). Elle retourne une chaîne de type `String` représentant les coordonnées de l'objet `Point`. Dès

lors, il n'est plus utile de définir de méthode d'affichage spécifique comme nous l'avions fait pour la plupart des classes Point précédentes.

Exemple :

```
public class Test {
 static public void main(String [] args) {
  Point p = new Point(1,1);
  System.out.println(p);
 }
}
class Point {
 private int x, y;
 public Point(int nx, int ny){
  x = nx;
  y = ny;
 }
 public String toString() {
  return "(" + x + ", " + y + ")";
 }
}
```

Résultat de l'exécution :

```
(1, 1)
```

11.2 Wrapper

11.2.1 Généralités

Les objets de type wrappers (enveloppeurs) représentent des objets qui encapsulent une donnée de type primitif et qui fournissent un ensemble de méthodes qui permettent notamment de faire des conversions et qui possèdent des données relatives aux domaines de valeurs des types primitifs. Bien évidemment, en fonction du type primitif, il y a des différences : manipulation de bits avec les entiers (`byte`, `short`, `int` et `long`), manipulation des valeurs infinies avec les réels (`float` et `double`), gestion de la nature des caractères avec les char, …. Il est à noter que les wrappers sont très utiles lorsque l'on veut manipuler des listes ou des tableaux de références d'objets représentant un type primitif.

Les classes wrappers faisant partie du package `java.lang`, elles sont automatiquement importées dans les programmes Java.

Type primitif	Wrapper
void	*java.lang.Void*
boolean	java.lang.Boolean
char	java.lang.Character
byte	java.lang.Byte
short	java.lang.Short
int	java.lang.Integer
long	java.lang.Long
float	java.lang.Float

Type primitif	Wrapper
`double`	`java.lang.Double`

Tableau 11 : Correspondance type primitif wrapper

Le programme suivant illustre l'utilisation des champs et des méthodes communes à l'ensemble des wrappers, ici, nous nous sommes limités aux deux classes `Integer` et `Double`.

- `Integer(10)` → **`Integer(int)`** : le constructeur qui permet une instanciation à partir du type primitif
- `Integer("11")`→ **`Integer(String)`** : le constructeur qui permet une instanciation à partir d'un objet `String`
- `Double.MAX_VALUE` : valeur maximum d'un réel de type `double`
- `Double.MIN_VALUE` : valeur minimum d'un réel de type `double`
- `Double.SIZE` : taille en bits d'un réel de type `double`
- `nn.longValue()`→**`long longValue()`**: retourne la valeur convertie dans le type primitif long (une de ces méthodes de conversion existe pour chaque type primitif numérique `longValue()` à `doubleValue()`)
- `Double.parseDouble("1.3")`→ **`static double parseDouble(String)`** : retourne la valeur correspondant à la chaîne passée dans le type primitif (méthode statique)
- `Double.valueOf(2.2)` → **`static Double valueOf(double)`** : retourne la référence d'un objet créé dont la valeur correspond à celle de l'argument passé, le type primitif correspondant à celui de la classe enveloppe (méthode statique)
- `Double.valueOf("1.4")` → **`static Double valueOf(String s)`** : retourne la référence d'un objet créé dont correspond à la chaîne passée dans le type primitif (méthode statique)

Il est à noter que les méthodes de conversion de chaînes de caractères sont successibles de provoquer une erreur d'exécution si la valeur de la chaîne n'a pas le bon format. Par exemple, l'appel `Integer.parseInt("17.1")` provoque l'erreur : « `java.lang.NumberFormatException` ».

Exemple :

```
public class Test {
 static public void main(String [] args) {
  Integer nn = new Integer(10);
  Integer mm = new Integer("11");
  Double xx = new Double(2.1);
  Double yy = new Double("1.2");
  System.out.println(Double.MAX_VALUE);
  System.out.println(Integer.MAX_VALUE);
  System.out.println(Double.MIN_VALUE);
  System.out.println(Integer.MIN_VALUE);
  System.out.println(Integer.SIZE);
  System.out.println(Double.SIZE);
  System.out.println(nn.longValue());
```

```
   System.out.println(mm.floatValue());
   System.out.println(xx.longValue());
   System.out.println(yy.floatValue());
   double z = Double.parseDouble("1.3");
   int p = Integer.parseInt("12");
   System.out.println(z);
   System.out.println(p);
   Double zz = Double.valueOf(2.2);
   Double tt = Double.valueOf("1.4");
   System.out.println(zz);
   System.out.println(tt);
   Integer pp = Integer.valueOf(14);
   Integer qq = Integer.valueOf("15");
   System.out.println(pp);
   System.out.println(qq);
  }
}
```

Résultat de l'exécution :

```
1.7976931348623157E308
2147483647
4.9E-324
-2147483648
32
64
10
11.0
2
1.2
1.3
12
2.2
1.4
14
15
```

11.2.2 Integer et traitement de bits

L'exemple suivant exploite les méthodes orientées traitement de bits de la classe `Integer`, des méthodes identiques existent pour les classes `Short` et `Long`. Pour tous les affichages, on convertit un entier en une chaîne de caractères au format binaire qui le représente avec la méthode statique `Integer.toBinaryString(int)`. Toutes ces méthodes sont des méthodes statiques que l'on peut considérer comme des utilitaires complétant les opérateurs binaires du langage Java.

- `Integer.rotateRight(n, 2)`→ **`static int rotateRight(int, int)`**: retourne la valeur correspondant au premier argument après un nombre de rotations à droite de la valeur du second argument
- `Integer.rotateLeft(n, 2)`→ **`static int rotateLeft(int, int) :`** retourne la valeur correspondant au premier argument après un nombre de rotations à gauche de la valeur du second argument

- `Integer.reverse(n)`→ **`static int reverse(int)`**: retourne la valeur de l'argument après avoir échangé les bits de poids forts avec les bits de poids faibles
- `Integer.highestOneBit(n)`→ **`static int highestOneBit(int)`** **:** retourne la valeur du premier argument après avoir mis à zéro tous les bits sauf le bit égal à 1 de poids le plus fort
- `Integer.lowestOneBit(n)`→ **`static int lowestOneBit(int)`** **:** retourne la valeur du premier argument après avoir mis à zéro tous les bits sauf le bit égal à 1 de poids le plus faible

Exemple :

```
public class Test {
 static public void main(String [] args) {
  int n = 11, r;
  String s = Integer.toBinaryString(n);
  System.out.println(s);
  s = Integer.toHexString(n) ;
  System.out.println(s);
  r = Integer.rotateRight(n, 2);
  s = Integer.toBinaryString(r);
  System.out.println(s);
  r = Integer.rotateLeft(n, 2);
  s = Integer.toBinaryString(r);
  System.out.println(s);
  r = Integer.reverse(n);
  s = Integer.toBinaryString(r);
  System.out.println(s);
  r = Integer.bitCount(n);
  s = Integer.toBinaryString(r);
  System.out.println(s);
  r = Integer.highestOneBit(n);
  s = Integer.toBinaryString(r);
  System.out.println(s);
  r = Integer.lowestOneBit(n);
  s = Integer.toBinaryString(r);
  System.out.println(s);
 }
}
```

Résultat de l'exécution :

```
1011
b
11000000000000000000000000000010
101100
11010000000000000000000000000000
11
1000
1
```

11.2.3 Double et infinis

La classe `Double`, ainsi que la classe `Float` disposent de membres permettant de manipuler des valeurs infinies. Ces valeurs n'ont de sens que pour les types réels et aucun pour les autres types entiers, par exemple `1.0/0` a une valeur symbolique en Java et `1/0` provoque l'erreur d'exécution « `java.lang.ArithmeticException: / by zero` ». Ces valeurs sont :

- La constante statique `Double.POSITIVE_INFINITY` (distincte de `Double.MAX_VALUE` qui a une valeur) correspond à +∞
- La constante statique `Double.NEGATIVE_INFINITY` correspond à -∞
- La constante statique `Double.NaN` correspond au résultat indéterminé d'une opération sur des réels (0/0, +∞-∞, …)

Quelques méthodes statiques que l'on peut, là encore, considérer comme des utilitaires permettant de tester les valeurs infinies. Le programme suivant en illustre l'utilisation. On peut constater que les règles sur les infinis sont respectées, par ailleurs, l'affichage de `Double.POSITIVE_INFINITY` donne `Infinity` et de `Double.NEGATIVE_INFINITY` donne `-Infinity`.

- `Double.isInfinite(1./0)`→ **`static boolean isInfinite(double)`**: retourne true si la valeur correspondant à l'argument vaut `Double.POSITIVE_INFINITY` ou `Double.NEGATIVE_INFINITY`
- `yy.isInfinite()`→**`boolean isInfinite()`**: retourne true si la valeur de l'objet courant vaut `Double.POSITIVE_INFINITY` ou `Double.NEGATIVE_INFINITY`
- `Double.isNaN(0./0)`→ **`static boolean isNaN(double)`**: retourne true si la valeur correspondant à l'argument vaut `Double.NaN`
- `yy.isNaN()`→**`boolean isNaN ()`**: retourne true si la valeur de l'objet courant vaut `Double.NaN`

Exemple :

```
public class Test {
 static public void main(String [] args) {
  double x = 1, y = -2;
  double z = Double.NaN;
  double t = Double.NEGATIVE_INFINITY;
  Double xx, yy;
  xx = new Double(x/0);
  System.out.println(xx);
  yy = new Double(y/0);
  System.out.println(yy);
  System.out.println(t);
  System.out.println(z);
  System.out.println(Double.NEGATIVE_INFINITY +
Double.POSITIVE_INFINITY);
  System.out.println(Double.POSITIVE_INFINITY +
Double.POSITIVE_INFINITY);
  System.out.println(Double.POSITIVE_INFINITY /
Double.POSITIVE_INFINITY);
  System.out.println(Double.MAX_VALUE*2);
```

```
   System.out.println(Double.isInfinite(1./0));
   System.out.println(yy.isInfinite());
   System.out.println(Double.isNaN(0./0));
   System.out.println(yy.isNaN());
  }
 }
```

Résultat de l'exécution :

```
Infinity
-Infinity
-Infinity
NaN
NaN
Infinity
NaN
Infinity
true
true
true
false
```

11.2.4 Character et types de caractères

La classe `Character` dispose un grand nombre de méthodes statiques permettant de tester le type de caractères (majuscule, minuscule, chiffre,...) et d'effectuer des conversions. Les méthodes de test sont de la forme `isFamille` si le caractère correspondant à la famille `true` est retourné, `false` sinon. Il existe des méthodes permettant de vérifier qu'une lettre ou un chiffre est un caractère pouvant faire partie d'un identifiant Java. Les valeurs de `MIN_VALUE` et de `MAX_VALUE` correspondent aux valeurs minimum (`'\u0000'`) et maximum (`'\FFFF'`) d'un Unicode. Comme il n'y a pas de difficulté, nous donnons simplement l'intitulé de ce qui est testé :

- `Character.isDigit(char)` : retourne `true` si le caractère passé est un chiffre, `false` sinon
- `Character.isLetter(char)` : retourne `true` si le caractère passé est une lettre, `false` sinon
- `Character.isLetterOrDigit(char)` : retourne `true` si le caractère passé est un chiffre ou une lettre, `false` sinon
- `Character.isUpperCase(char)` : retourne `true` si le caractère passé est une lettre majuscule, `false` sinon
- `Character. isLowerCase (char)` : retourne `true` si le caractère passé est une lettre minuscule, `false` sinon
- `Character.isWhitespace(char)` : retourne true si le caractère passé est un espace c'est-à-dire un des caractères `' '`, `'\t'`, `'\f'`, `'\n'` ou `'\r'`, `false` sinon

Exemple :

```
public class Test {
 static public void main(String [] args) {
  String s = "Abcé1-?+r ";
```

```
   for(int i = 0; i < s.length(); i++) {
    char c = s.charAt(i);
    System.out.println("-------------------------------------");
    System.out.println(c);
    System.out.println("isDigit " + Character.isDigit(c));
    System.out.println("isLetter " + Character.isLetter(c));
    System.out.println("isLetterOrDigit " +
 Character.isLetterOrDigit(c));
    System.out.println("isLowerCase " + Character.isLowerCase(c));
    System.out.println("isUpperCase " + Character.isUpperCase(c));
    System.out.println("isWhitespace " + Character.isWhitespace(c));
   }
  }
 }
```

Résultat de l'exécution :

```
-------------------------------------
A
isDigit false
isLetter true
isLetterOrDigit true
isLowerCase false
isUpperCase true
isWhitespace false
-------------------------------------
b
isDigit false
isLetter true
isLetterOrDigit true
isLowerCase true
isUpperCase false
isWhitespace false
-------------------------------------
c
isDigit false
isLetter true
isLetterOrDigit true
isLowerCase true
isUpperCase false
isWhitespace false
-------------------------------------
é
isDigit false
isLetter true
isLetterOrDigit true
isLowerCase true
isUpperCase false
isWhitespace false
-------------------------------------
1
```

```
isDigit true
isLetter false
isLetterOrDigit true
isLowerCase false
isUpperCase false
isWhitespace false
-------------------------------------
-
isDigit false
isLetter false
isLetterOrDigit false
isLowerCase false
isUpperCase false
isWhitespace false
-------------------------------------
?
isDigit false
isLetter false
isLetterOrDigit false
isLowerCase false

isUpperCase false
isWhitespace false
-------------------------------------
+
isDigit false
isLetter false
isLetterOrDigit false
isLowerCase false
isUpperCase false
isWhitespace false
-------------------------------------
r
isDigit false
isLetter true
isLetterOrDigit true
isLowerCase true
isUpperCase false
isWhitespace false
-------------------------------------

isDigit false
isLetter false
isLetterOrDigit false
isLowerCase false
isUpperCase false
isWhitespace true
```

11.2.5 Autoboxing

L'**autoboxing** ou **boxing** (emballage) automatique est un mécanisme de conversion automatique qui existe depuis la version 1.5 de Java. Il permet de convertir automatiquement une valeur de type primitif en objet du wrapper correspondant. Ce mécanisme allège l'écriture des programmes. Le processus inverse appelé **unboxing** (déballage) automatique existe aussi.

L'exemple suivant montre ce qu'il faudrait faire en l'absence d'un tel mécanisme.

- `Integer ii = new Integer(i)` : `int` → `Integer` (boxing)
- `n = ii.intValue()` : `Integer` → `int` (unboxing)

Exemple :

```
public class Test {
 static public void main(String [] args) {
 int i = 3, n;
 Integer ii = new Integer(i);
 n = ii.intValue() ;
 System.out.println(i + " " + n+ " " + ii);;
 }
}
```

Résultat de l'exécution :

```
3 3 3
```

L'exemple suivant utilise les mécanismes de boxing et unboxing automatiques.

- `ii = i` : `int` → `Integer` (boxing)
- `n = ii` : `Integer` → `int` (unboxing)

Exemple :

```
public class Test {
 static public void main(String [] args) {
 int i = 3, n;
 Integer ii = i;
 n = ii;
 System.out.println(i + " " + n+ " " + ii);;
 }
}
```

Résultat de l'exécution :

```
3 3 3
```

11.2.6 Synthèse sur les possibilités de conversion

Les tableaux suivants synthétisent les principales possibilités de conversion des wrappers et de la classe `String`. On utilise la classe `Integer` qui est par défaut la classe d'appartenance des méthodes du tableau, dans les autres cas, la classe de base est explicitée (on privilège la classe `Double`, sachant qu'un équivalent existe avec les autres wrappers).

↗	**Integer**
int	`Integer(int value)` `static Integer valueOf(int i)`

↗	**Integer**

String	`static Integer valueOf(String)`
	`static Integer valueOf(String, int)` à partir de la base
	`static Integer decode(String)`
	`Integer(String)`

autres	

Tableau 12 : Conversion vers wrapper (Integer)

↗	**int**
Integer	`int intValue()`
String	`static int parseInt(String)` `static int parseInt(String, int)` à partir de la base
autres	`int intValue()` de la classe `Double`

Tableau 13 : Conversion vers type primitif (int)

↗	**String**

int	`static String toString(int)`
	`static String toString(int, int)` vers la base
	`static String valueOf(int)` de la classe `String`

Integer	`String toString()`
autres	`static String toString(double)` de la classe `Double` `static String valueOf(double)` de la classe `String`

Tableau 14 : Conversion vers String

L'exemple suivant illustre l'utilisation des méthodes permettant d'effectuer des changements de base. Les deux méthodes `valueOf` et `parseInt` effectuent une conversion du nombre spécifié par le premier argument (représenté par une chaîne de caractères) exprimé dans la base spécifiée par le second argument vers un nombre en base 10. Si la base n'est pas cohérente avec la valeur du premier argument, cela provoque une erreur d'exécution, par exemple, l'instruction `ii = Integer.valueOf("12", 2);` aurait provoqué l'erreur : « `java.lang.NumberFormatException: For input string: "12"` ».

La méthode `toString` effectue une conversion du nombre en base 10 du premier argument dans la base spécifiée par le second argument, le résultat étant représenté par une chaîne de caractères.

Exemple :

```
public class Test {
 static public void main(String [] args) {
 int i ;
 Integer ii ;
 String s;
 ii = Integer.valueOf("12", 3);
 i = Integer.parseInt("43", 5);
```

```
  s = ii.toString(123, 8);
  System.out.println("ii = " + ii);
  System.out.println("i = " + i);
  System.out.println("s = " + s);
  }
 }
```

Résultat de l'exécution :

```
ii = 5
i = 23
s = 173
```

11.3 Math

La classe `Math` regroupe un ensemble de méthodes et constantes mathématiques. Elle comporte des fonctionnalités comparables aux bibliothèques mathématiques des autres langages, comme celle du langage C (associée au fichier math.h). Elle opère majoritairement sur des valeurs réelles de type double.

La classe `Math` faisant partie du package `java.lang`, elle est automatiquement importée dans les programmes Java.

Une des particularités de cette classe est qu'elle ne possède que des membres statiques et logiquement aucun constructeur (que des membre statiques dont aucun membres d'instance). C'est vraiment une « classe utilitaire ».

Les fonctions méthodes mathématiques retournent la valeur correspondant à la constante symbolique `NaN` lorsque le calcul demandé n'a pas de sens, par exemple :

- `Math.asin(2)` → `NaN`

De même elles gèrent aussi les infinis , par exemple :

- `Math.sqrt(Double.POSITIVE_INFINITY)` → `Infinity`

Elle possède les deux constantes :

- `static double E` : valeur approchée de la base des logarithmes népériens
- `static double PI` : valeur approchée du nombre π

Elle possède par ailleurs des méthodes que l'on peut classer en plusieurs catégories :

Les fonctions mathématiques de base : trigonométriques, trigonométriques inverses, hyperboliques, hyperboliques inverses, exponentielles, racines, puissances,…

- Les arrondis
- Les comparaisons
- La valeur absolue
- La génération de nombres aléatoires

11.3.1 Fonctions mathématiques

Le programme suivant illustre l'utilisation de certaines des fonctions mathématiques de base. Leur utilisation étant assez facile, nous ne les commentons pas.

Exemple :

```
public class Test {
 static public void main(String [] args) {
 double x = Math.PI/4, y;
```

```
  System.out.println("cos"+"("+x+") = "+ Math.cos(x));
  System.out.println("tan"+"("+x+") = "+ Math.tan(x));
  x = 1;
  System.out.println("asin"+"("+x+") = "+ Math.asin(x));
  System.out.println("atan"+"("+x+") = "+ Math.atan(x));
  x = Math.E;
  System.out.println("log"+"("+x+") = "+ Math.log(x));
  System.out.println("log10"+"("+x+") = "+ Math.log10(x));
  x = 1;
  System.out.println("exp"+"("+x+") = "+ Math.exp(x));
  System.out.println("cosh"+"("+x+") = "+ Math.cosh(x));
  x = 9;
  System.out.println("sqrt"+"("+x+") = "+ Math.sqrt(x));
  x = 8;
  System.out.println("cbrt"+"("+x+") = "+ Math.cbrt(x));
  x = 2;
  y = 3;
  System.out.println("pow"+"("+x+", "+ y +") = "+ Math.pow(x, y));
  }
}
```

Résultat de l'exécution :

```
cos(0.7853981633974483) = 0.7071067811865476
tan(0.7853981633974483) = 0.9999999999999999
asin(1.0) = 1.5707963267948966
atan(1.0) = 0.7853981633974483
log(2.718281828459045) = 1.0
log10(2.718281828459045) = 0.4342944819032518
exp(1.0) = 2.7182818284590455
cosh(1.0) = 1.543080634815244
sqrt(9.0) = 3.0
cbrt(8.0) = 2.0
oow(2.0, 3.0) = 8.0
```

11.3.2 Comparaisons

Le programme suivant illustre l'utilisation des méthodes de comparaison. Leur utilisation étant assez facile, nous ne les commentons pas. Il existe une version des méthodes `min` et `max` pour les `int`, les `long`, les `float` et les `double`.

Exemple :

```
public class Test {
  static public void main(String [] args) {
  double x = Math.PI/4, y = Math.E/3, v, w;
  v = Math.max(x , y);
  w = Math.min(x , y);
  System.out.println(w +" <= " +v);
  int a = 2, b = 3, c, d;
  c = Math.max(a, b);
  d = Math.min(a, b);
  System.out.println(a +" <= " +b);
```

```
 }
}
```

Résultat de l'exécution :

```
0.7853981633974483 <= 0.9060939428196817
2 <= 3
```

11.3.3 Arrondis

Le programme suivant illustre l'utilisation des méthodes d'arrondi.

- `static double ceil(double a)` : retourne le plus petit entier, converti en double, supérieur ou égal à la valeur de l'argument (arrondi par excès)
- `static double floor(double a)` : retourne le plus grand entier, converti en `double`, inférieur ou égal à la valeur de l'argument (arrondi par défaut)
- `static long round(double a)` : retourne l'entier de type `long` dont la valeur est la plus proche de l'argument (arrondi au plus prêt)
- `static long round(double a)` : retourne l'entier, converti en `double`, dont la valeur est la plus proche de l'argument
- `static double nextUp(double d)` : retourne la valeur du plus petit `double` dont la valeur est strictement supérieure à l'argument (l'argument et la valeur retournée sont des nombres adjacents, leur écart dépend de la précision des doubles)

Exemple :

```
public class Test {
 static public void main(String [] args) {
 double x = Math.PI, y = Math.E;
 System.out.println("ceil"+"("+x+") = "+ Math.ceil(x));
 System.out.println("ceil"+"("+y+") = "+ Math.ceil(y));
 System.out.println("floor"+"("+x+") = "+ Math.floor(x));
 System.out.println("floor"+"("+y+") = "+ Math.floor(y));
 System.out.println("round"+"("+x+") = "+ Math.round(x));
 System.out.println("round"+"("+y+") = "+ Math.round(y));
 System.out.println("rint"+"("+x+") = "+ Math.rint(x));
 System.out.println("rint"+"("+y+") = "+ Math.rint(y));
 System.out.println("nextUp"+"("+x+") = "+ Math.nextUp(x));
 }
}
```

Résultat de l'exécution :

```
ceil(3.141592653589793) = 4.0
ceil(2.718281828459045) = 3.0
floor(3.141592653589793) = 3.0
floor(2.718281828459045) = 2.0
round(3.141592653589793) = 3
round(2.718281828459045) = 3
rint(3.141592653589793) = 3.0
rint(2.718281828459045) = 3.0
nextUp(3.141592653589793) = 3.1415926535897936
```

11.3.4 Génération de nombres aléatoires

La génération d'un nombre aléatoire de type `double` compris entre `0.0` et `1.0` s'effectue avec la méthode statique `random()`. La classe `Random`, du package `java.util`, est plus spécialisée et permet de travailler plus finement.

Le programme suivant affiche 10 nombres réels de type `double` générés aléatoirement par la méthode `random()`.

Exemple :

```
public class Test {
 static public void main(String [] args) {
 for(int i = 0; i < 10; i++)
  System.out.println(Math.random());
 }
}
```

Résultat de l'exécution :

```
0.4243156575520196
0.5741880870524587
0.7338883111717018
0.6235159182933898
0.3260900295863459
0.656229814936521
0.09295279462865558
0.31243906469279403
0.1176212841007035
0.8766417439900582
```

11.4 Arrays

Java dispose d'une classe `Arrays` (du package `java.util`) qui possède quelques fonctions permettant d'effectuer le tri et la recherche sur des tableaux d'éléments de types de base[20] et d'autres opérations comme des comparaisons, des copies partielles, ...

Dans cette partie, nous nous limitons à l'étude d'un exemple illustrant quelques possibilités sur des tableau d'entiers. Toutes les méthodes de cette classe sont statiques, il s'agit d'un ensemble d'utilitaires facilitant la gestion des tableaux.

- `Arrays.copyOf(ta,5);`→ **`static int[] copyOf(int[], int)`** : retourne la référence d'une copie partielle des premiers éléments (de la référence) du tableau correspondant au premier argument. La valeur du second argument spécifie le nombre d'éléments copiés, si cette valeur est plus grande que la taille du tableau en argument, le tableau retourné est complété avec des 0
- `Arrays.copyOfRange(ta,3,9);` → **`static int[] copyOfRange(int[], int, int)`** : retourne la référence d'une copie partielle des éléments du tableau correspondant au premier argument. Il s'agit des éléments dont

[20] Cette classe permet de traiter aussi des objets, mais ces objets doivent être comparables. Java dispose de *collections* permettant d'effectuer des opérations sur des structures de données complexes : ensembles, arbres, ou listes d'éléments (recherche, tri, insertion, concaténation,...).

l'indice est supérieur ou égal au second argument et dont l'indice est strictement inférieur au dernier argument

- `Arrays.equals(ta tb);` → **`static boolean equals(int[], int[])`** : retourne `true` si les deux tableaux sont identique, `false` sinon
- `Arrays.fill(tc,7);` → **`static void fill(int[], int)`** : affecte aux éléments du tableau correspondant au premier argument la valeur correspondant au second argument
- `Arrays.fill(tb 0,5,-1);` → **`static void fill(int[] a, int, int, int)`** : remplit partiellement le tableau correspondant au premier argument avec la valeur du dernier argument. Il s'agit des éléments dont l'indice est supérieur ou égal au second argument et dont l'indice est strictement inférieur au troisième argument
- `Arrays.sort(tb,3,tb.length);` → **`static void sort(int[], int, int)`** : tri par ordre croissant un sous ensemble des éléments du tableau correspondant au premier argument. Il s'agit des éléments dont l'indice est supérieur ou égal au second argument et dont l'indice est strictement inférieur au troisième argument
- `Arrays.binarySearch(ta,0,4,5);` → **`static int binarySearch(int[], int, int, int)`** : effectue la recherche dichotomique de la valeur correspondant au second argument dans un sous tableau du tableau correspondant au dernier argument, si la valeur est trouvée l'indice de l'élément est retourné, sinon une valeur négative est retournée (il faut préalablement que le tableau soit trié par ordre croissant). Les éléments concernés par la recherche sont les éléments dont l'indice est supérieur ou égal au second argument et dont l'indice est strictement inférieur au troisième argument
- `Arrays.sort(ta);` → **`static void sort(int[] a)`** : tri par ordre croissant les éléments du tableau correspondant au l'argument
- `Arrays.binarySearch(ta,5);` → **`static int binarySearch(int[], int)`** : effectue la recherche dichotomique de la valeur correspondant au second argument dans le tableau correspondant au premier argument, si la valeur est trouvée l'indice de l'élément est retourné, sinon une valeur négative est retournée (il faut préalablement que le tableau soi trié par ordre croissant)
- `Arrays.toString(ta);` → **`static String toString(int[])`** : converti le tableau en chaîne de caractères en utilisant une représentation de type liste.

Exemple :

```
import java.util.*;
public class Test {
 public static void main(String[] args) {
  int []ta = {2, 6, 8, 4, 1, 0, 6, 3, 11, 12, 5};
  int [] tb, tc;
  printTbInt(ta);
  tb = Arrays.copyOf(ta, 5);
  printTbInt(tb);
  tb = Arrays.copyOf(ta, 15);
  printTbInt(tb);
```

```
   System.out.println(tb.length);
   tc =Arrays.copyOfRange(ta, 3, 9);
   printTbInt(tc);
   System.out.println(Arrays.equals(ta, tb));
   Arrays.fill(tc, 7);
   printTbInt(tc);
   Arrays.fill(tb, 0, 5, -1);
   printTbInt(tb);
   System.out.println(tb.length);
   Arrays.sort(tb,3,tb.length);
   printTbInt(tb);
   Arrays.sort(ta);
   printTbInt(ta);
   int i = Arrays.binarySearch(ta, 0, 4, 5);
   System.out.println(i);
   i =  Arrays.binarySearch(ta, 5);
   System.out.println(i);
   System.out.println(Arrays.toString(ta));
 }
    static void printTbInt(int []t) {
    if(t == null) return;
    System.out.print("{");
    int i;
    for(i = 0; i < t.length -1; i++) {
     System.out.print(t[i] + ", ");
    }
    System.out.println(t[i] + "}");
    }
}
```

Résultat de l'exécution :

```
{2, 6, 8, 4, 1, 0, 6, 3, 11, 12, 5}
{2, 6, 8, 4, 1}
{2, 6, 8, 4, 1, 0, 6, 3, 11, 12, 5, 0, 0, 0, 0}
15
{4, 1, 0, 6, 3, 11}
false
{7, 7, 7, 7, 7, 7}
{-1, -1, -1, -1, -1, 0, 6, 3, 11, 12, 5, 0, 0, 0, 0}
15
{-1, -1, -1, -1, -1, 0, 0, 0, 0, 0, 3, 5, 6, 11, 12}
{0, 1, 2, 3, 4, 5, 6, 6, 8, 11, 12}
-5
5
[0, 1, 2, 3, 4, 5, 6, 6, 8, 11, 12]
```

12.Exercices de synthèse

12.1 Nombres complexes

L'objectif de cet exercice est de définir une classe permettant de représenter et de manipuler des nombres complexes. Un nombre complexe z peut se représenter en coordonnées cartésiennes comme une somme z = x + yi, où x et y sont des nombres réels quelconques et (l'unité imaginaire) est un nombre particulier tel que $i^2 = -1$.

Le réel x est appelé partie réelle de z et se note Re(z), et le réel y est sa partie imaginaire et se note Im(z).

Pour représenter les nombres complexes, nous avons défini la classe `Complexe` qui comporte deux champs de type `double`, `x` pour la partie réelle et `y` pour la partie imaginaire. Nous avons défini trois constructeurs :

- `public Complexe(double nx, double ny)` : le constructeur générique
- `public Complexe()` : le constructeur par défaut
- `public Complexe(Complexe p)` : le constructeur de recopie

Nous avons, par ailleurs, défini pour chaque opération une méthode. On peut remarquer que ces méthodes ne sont pas des méthodes statiques, nous aurions pu choisir d'adopter des méthodes statiques (ou avoir les deux versions, les signatures étant différentes). Dans ce cas, la méthode statique équivalent à `public Complexe somme(Complexe z)` aurait eu l'allure suivante :

```
public static Complexe somme(Complexe z1, Complexe z2) {
  return new Complexe(z1.x + z2.x, z1.y + z2.y);
}
```

Et être alors appelée :

```
z3 = Complexe.somme(z1, z2);
```

Pour des questions de lisibilité, les accesseurs sont appelés `getRe()` et `getIm()` plutôt que `getX()` et `getY()` (même convention pour les altérateurs).

Pour le calcul du module, la fonction `hypot` de la classe `Math` est utilisée (`Math.hypot(x, y)` ⇔ `Math.sqrt(x*x + y*y)`).

On peut enfin remarquer que la valeur `Double.NaN` est utilisée pour donner une valeur à l'inverse d'un complexe dont le module est nul ou lorsque l'argument d'un complexe ne peut être calculé.

Les méthodes de clonage, de comparaison et de conversion en chaînes de caractères sont ajoutées afin d'obtenir une classe en forme standard.

Solution possible :

```
public class Test {
 static public void main(String [] args) {
  Complexe z1, z2, z3;
  z1 = new Complexe(1.0, 2.0);
  z2 = new Complexe(3.0, 1.0);
  z3 = z1.somme(z2);
  System.out.println(z3);
  z3 = z1.difference(z2);
  System.out.println(z3);
  z3 = z1.produit(z2);
  System.out.println(z3);
```

```
   z3 = z1.division(z2);
   System.out.println(z3);
   z3 = z1.conjugue();
   System.out.println(z3);
   z2 = z1.inverse();
   System.out.println(z2);
   z3 = z1.produit(z2);
   System.out.println(z3);
   z1 = new Complexe(4.0, 3.0);
   System.out.println(z1.module());
   System.out.println(z1.argument());
  }
 }
 class Complexe {
  private double x, y;
  public  Complexe(double nx, double ny) {
     x = nx;
     y = ny;
  }
  public  Complexe() {
   this(0, 0);
  }
  public Complexe(Complexe p) {
   this(p.x, p.y);
  }
  public Complexe clone() {
   return new Complexe(this);
  }
  public void setRe(double nx) {
   x = nx;
  }
  public void setIm(double ny) {
   y = ny;
  }
  public double getRe() {
   return x;
  }
  public double getIm() {
   return y;
  }
  public Complexe somme(Complexe z) {
   return new Complexe(x + z.x, y + z.y);
  }
  public Complexe difference(Complexe z) {
   return new Complexe(x - z.x, y - z.y);
  }
  public Complexe produit(Complexe z) {
   return new Complexe(x * z.x - y * z.y, x*z.y + y* z.x);
  }
  public Complexe division(Complexe z) {
```

```
   return produit(z.inverse());
  }
  public Complexe conjugue() {
   return new Complexe(x, -y);
  }
  public double module() {
   return Math.hypot(x, y);
  }
  public double argument() {
   double mod = module();
   if(mod == 0) return Double.NaN;
   if (y >= 0) return Math.acos(x/mod);
   return 2*Math.PI - Math.acos(x/mod) ;
  }
  public Complexe inverse() {
   if(x == 0 && y == 0)
    return new Complexe(Double.NaN, Double.NaN);
   return new Complexe(x / (x*x + y*y), -y / (x*x + y*y));
  }
  public String toString() {
   return x + " + " + y + "i";
  }
  public boolean equals(Complexe z) {
   return x == z.x && y ==  z.y;
  }
  public int compareTo(Complexe z){
   return (int)(module() - z.module());
  }
 }
```

Résultat de l'exécution :

```
4.0 + 3.0i
-2.0 + 1.0i
1.0 + 7.0i
0.5 + 0.5i
1.0 + -2.0i
0.2 + -0.4i
1.0 + 0.0i
5.0
0.6435011087932843
```

12.2 Compte en banque

L'objectif de cet exercice est de définir une classe permettant de représenter et de manipuler des comptes bancaires. Pour simplifier le problème, nous supposons qu'il s'agit de comptes de type compte courant permettant d'effectuer des versements et retraits dont on ne détaille pas l'origine ou le type (espèce, carte bleue, chèque, …) . On suppose que tout compte courant est associé à un et un seul titulaire, identifié par son nom, et possède un numéro unique.

Par ailleurs, tout compte dispose d'un découvert autorisé défini une fois pour toute à son ouverture, si une tentative de débit dépasse ce découvert, alors le compte est bloqué.

Par ailleurs, après son ouverture, le compte est bloqué jusqu'à que le titulaire dépose une somme supérieure ou égale à 10€. Afin de débloquer le compte après une tentative de découvert trop importante, il faut recréditer le compte avec un solde supérieur ou égal 10€.

La classe Compte possède les champs déclarés `final` correspondant aux caractéristiques constantes demandées :

- `nomTitulaire` nom du titulaire
- `prenomTitulaire` prénom du titulaire
- `numero` numéro de compte
- `decouvert` découvert permis

Le champ `solde` correspond au solde du compte.

La constante statique `SOLDMIN` correspond au solde minimum permettant le premier déblocage du compte, tous les comptes ont la même valeur de déblocage.

Le champ booléen `verrou` correspond à l'indicateur de verrouillage du compte qui est vrai tant que l'on a pas versé le premier versement de 10€ après la création du compte, ou après une tentative de retrait donnant lieu à un découvert trop important.

Le champ `nbVersement` correspond au nombre de versements, initialement à zéro, il est utilisé pour le premier déblocage.

Enfin le champ statique `nbase` permet d'attribuer automatiquement les numéros de compte au fur et à mesure de leur création.

Les méthodes `versement` et `retrait` permettent d'ajouter ou de retirer de l'argent sur le compte, elle vérifie si le compte est verrouillé ou non. La méthode `retrait` provoque le verrouillage en cas de tentative de dépassement du découvert. La méthode `versement` déverrouille le compte lors du premier versement supérieur ou égal à 10€.

La méthode `deverrouille` permet de déverrouiller le compte, le déverrouillage ne peut s'effectuer que si le compte est soldé d'au moins 10€.

La méthode `toString()` de la classe Compte qui retourne une chaîne représentant toutes les caractéristiques de l'objet.

L'unicité des titulaires n'est pas testée, c'est au niveau du `main` que cela doit être fait (l'idéal étant de créer une classe agence qui encapsule et gère les comptes).

Comme les chaînes de caractères de type `String` sont constantes, l'utilisation des méthodes `getNomTitulaire` et `getPrenomTitulaire` ne nécessitent pas de précaution particulière pour protéger les champs `nomTitulaire` et `prenomTitulaire`.

Solution possible :

```
import java.util.*;
public class Test {
 public static void main(String[] args) {
  Compte [] gringotts = new Compte [20];
  gringotts[0] = new Compte("Rolanda", "Bibine",
  Math.ceil(100*Math.random()));
  gringotts[1] = new Compte("Cuthbert", "Binns",
  Math.ceil(100*Math.random()));
  gringotts[2] = new Compte("Pomona", "Chourave",
  Math.ceil(100*Math.random()));
  gringotts[3] = new Compte("Albus", "Dumbledore",
  Math.ceil(100*Math.random()));
```

```
   gringotts[4] = new Compte("Filius", "Flitwick",
   Math.ceil(100*Math.random()));
   gringotts[5] = new Compte("Minerva", "McGonagall",
   Math.ceil(100*Math.random()));
   gringotts[6] = new Compte("Rubeus", "Hagrid",
   Math.ceil(100*Math.random()));
   gringotts[7] = new Compte("Remus", "Lupin",
   Math.ceil(100*Math.random()));
   gringotts[8] = new Compte("Alastor", "Maugrey",
   Math.ceil(100*Math.random()));
   gringotts[9] = new Compte("Dolores", "Ombrage",
   Math.ceil(100*Math.random()));
   gringotts[10] = new Compte("Severus", "Rogue",
   Math.ceil(100*Math.random()));
   int i = 0;
   while(gringotts[i] != null) {
    gringotts[i].versement(Math.ceil(1000*Math.random()));
    gringotts[i].retrait(Math.ceil(1000*Math.random()));
    System.out.println(gringotts[i]);
    i++;
   }
   gringotts[11] = new Compte("Argus", "Rusard", 100);
   System.out.println(gringotts[11].versement(20));
   System.out.println(gringotts[11]);
   System.out.println(gringotts[11].retrait(30));
   System.out.println(gringotts[11]);
   System.out.println(gringotts[11].retrait(30));
   System.out.println(gringotts[11]);
   System.out.println(gringotts[11].retrait(80));
   System.out.println(gringotts[11]);
   System.out.println(gringotts[11].versement(20));
   System.out.println(gringotts[11]);
   System.out.println(gringotts[11].retrait(30));
   System.out.println(gringotts[11]);
   System.out.println(gringotts[11].deverrouille(120));
   System.out.println(gringotts[11]);
  }
 }
 class Compte {
  private final String nomTitulaire;
  private final String prenomTitulaire;
  private final int numero;
  private double solde;
  private final double decouvert;
  private boolean verrou;
  private int nbVersement;
  static final int SOLDMIN = 10;
  static int nbase = 100;
   public Compte(String prenom,String nom, double decou){
   numero = nbase ++;
```

```
 nomTitulaire = nom;
 prenomTitulaire = prenom;
 decouvert = decou;
 verrou = true;
}
public boolean versement(double v){
 if(nbVersement == 0) {
  if(v >= SOLDMIN) {
   solde += v;
   verrou = false;
   nbVersement ++;
   return true;
  } else
   return false;
 }
 if(verrou) return false;
 solde += v;
 nbVersement ++;
 return true;
}
public boolean retrait(double v){
 if(verrou) return false;
 double nsolde = solde - v;
 if(nsolde < -decouvert){
  verrou = true;
  return false;
 }
 solde = nsolde;
 return true;
}
public boolean isVerrou() {
 return verrou;
}
public String getNomTitulaire() {
 return nomTitulaire;
}
public String getPrenomTitulaire() {
 return prenomTitulaire;
}
public int getNumero() {
 return numero;
}
public boolean deverrouille(double v) {
 double nsolde = solde + v;
 if(nsolde > SOLDMIN) {
  verrou = false;
  solde = nsolde;
  return true;
 }
 return false;
```

```
  }
  public String toString(){
   return  numero + " : "+nomTitulaire+ " " +prenomTitulaire+ " = " +
   solde + " " + verrou+ " " + decouvert;
  }
 }
```

Résultat de l'exécution :

```
100 : Bibine Rolanda = 840.0 true 39.0
101 : Binns Cuthbert = 457.0 false 21.0
102 : Chourave Pomona = 40.0 false 84.0
103 : Dumbledore Albus = 58.0 false 96.0
104 : Flitwick Filius = 590.0 false 78.0
105 : McGonagall Minerva = 352.0 false 72.0
106 : Hagrid Rubeus = 242.0 false 7.0
107 : Lupin Remus = 592.0 true 33.0
108 : Maugrey Alastor = 530.0 true 83.0
109 : Ombrage Dolores = 60.0 false 31.0
110 : Rogue Severus = 170.0 false 54.0
true
111 : Rusard Argus = 20.0 false 100.0
true
111 : Rusard Argus = -10.0 false 100.0
true
111 : Rusard Argus = -40.0 false 100.0
false
111 : Rusard Argus = -40.0 true 100.0
false
111 : Rusard Argus = -40.0 true 100.0
false
111 : Rusard Argus = -40.0 true 100.0
true
111 : Rusard Argus = 80.0 false 100.0
```

12.3 Cercle et classe interne

L'objectif de cet exercice est de définir une classe permettant de représenter et de manipuler des cercles. Un cercle est caractérisé par son centre, un point de coordonnées x et y, et son rayon. On désire par ailleurs définir deux opérations sur les cercles : l'homothétie de rapport k par rapport au centre du cercle qui agrandit ou réduit le cercle et une translation qui déplace le cercle. On impose d'utiliser une classe interne pour définir le point. On ne définit qu'un ensemble minimum de méthodes (pas d'écriture standard).

Pour représenter les cercles, nous avons défini la classe interne `Point` qui comporte deux champs de type `double`, `x` pour l'abscisse et `y` pour l'ordonnée. La classe `Point` possède un constructeur, la méthode `translate` permettant d'effectuer la translation d'un point et la méthode d'affichage `print`. Puis, pour définir la classe `Cercle`, nous avons déclaré le champ `centre` de type `Point` et le champ `rayon`. Cette classe possède un constructeur dont l'objet est d'initialiser ses champs, deux méthodes `translate` et `homothetie` permettant de réaliser les opérations demandées et la méthode d'affichage `print`. La méthode `translate` de la classe `Cercle` appelle la méthode `translate` de la classe `Point` (translater un cercle revient à translater sont centre). Comme l'objet

référencé par `centre` ne manipule jamais l'objet de type `Cercle` qui l'encapsule, on aurait pu déclarer `Point static`. La solution retenue, par rapport à l'exemple traité avec deux classes séparées (§7 p97), permet de complètement dissimuler le champ `centre` et la définition de sa classe, mais bien évidemment, seuls les objets de type `Cercle` peuvent manipuler des objets de type `Point`.

Solution possible :

```
public class Test {
 static public void main(String [] args) {
  Cercle c1 = new Cercle(1,1,2);
  c1.print();
  c1.translate(1,2);
  c1.print();
  c1.homothetie(3);
  c1.print();
 }
}
class Cercle {
 private class Point {
  private double x, y;
  public  Point(double nx, double ny) {
      x = nx;
      y = ny;
  }
  public void print() {
      System.out.print("("+ x+", "+ y+")");
  }
  public void translate(double dx, double dy) {
   x += dx;
   y += dy;
  }
 }
 private Point centre;
 private double rayon;
 public Cercle(double nx, double ny, double nRayon) {
  rayon = nRayon;
  centre = new Point(nx, ny);
 }
 public void print() {
  centre.print();
  System.out.println(" : " + rayon );
 }
 public void translate(double dx, double dy) {
  centre.translate(dx, dy);
 }
 public void homothetie(double rap) {
  rayon *= rap;
 }
}
```

Résultat de l'exécution :

```
(1.0, 1.0) : 2.0
(2.0, 3.0) : 2.0
(2.0, 3.0) : 6.0
```

12.4 Liste chaînée

Une liste est une collection ordonnée et de taille arbitraire d'éléments de même type, sur laquelle on peut effectuer les opérations suivantes :

- Ajouter un élément à l'ensemble
- Trouver la position d'un élément
- Connaître le nombre total d'éléments de l'ensemble
- …

L'accès aux éléments d'une liste se fait de manière séquentielle : chaque élément permet l'accès au suivant (contrairement au cas du tableau, dans lequel, l'accès se fait de manière directe via un indice).

L'API Java possède des classes nommées collections qui permettent de créer et d'exploiter des structures de données comme les listes chaînées.

Nous allons définir ici une liste chaînée simple de points. Une liste chaînée simple est un cas particulier de liste, c'est un ensemble d'éléments dont chacun contient la référence de l'élément suivant. Un élément se nomme nœud ou cellule. Nous rendrons cette liste dynamique, c'est-à-dire que les nœuds seront créés au fur et à mesure, ce qui permet de gérer des ensembles de tailles variables inconnues à l'avance.

Chaque noeud possède des champs dont :

- la partie informationnelle (contenu), ici les coordonnées d'un point.
- la référence du noeud suivant.

Nous avons plusieurs possibilités pour coder une liste chaînée de points. La première consiste à ajouter à la classe `Point` un champ `suivant`, un accesseur et un altérateur associés à ce champ. L'enchaînement des points constitue la liste, à partir du premier nœud on peut accéder à tous les nœuds et ainsi parcourir la liste.

```
class Point {
  private int x, y;
  private Point suivant;
  public  Point(int nx, int ny) {
    …
  public void setSuivant(Point nsuivant) {
   suivant = nsuivant;
  }
  public Point getSuivant() {
   return suivant;
  …
}
```

Cette structure, bien que suffisante, n'est pas satisfaisante, en effet, elle nécessite de réécrire partiellement la classe `Point`. Il n'est pas toujours possible de modifier une classe existante, par exemple, nous ne pourrions pas faire une liste chaînée d'objets de type `String` en adoptant ce principe. Nous proposons donc, de définir une structure de données intermédiaire dont un champ référence un objet de type `Point` et un autre la structure elle-même.

La classe `NœudPoint` est une structure autoréférentielle, chaque noeud référence le suivant, c'est le rôle du champ `next`, et possède la référence d'un objet de type `Point` qui constitue la partie informationnelle du noeud. Ainsi, la classe `Point` ne nécessite aucune modification (nous avons défini une classe `Point` simplifiée). Par convention, La liste possède comme champ `premier`, la référence du premier noeud de la liste et un autre champ `size`, dont la valeur correspond au nombre de nœuds créés. Une liste est vide si la valeur de `premier` vaut `null`. De plus, le dernier noeud a pour valeur `null` (marque la fin de la liste).

La Figure 4 représente la structure d'une liste chaînée constituée ici de trois objets de type `Point`. L'objet (référencé par `lp`) de type `ListPoint` possède le champ `premier` qui référence le premier objet de type `NoeudPoint` de la liste, cet objet référence son successeur (champ `next`) et l'objet de type `Point` associé (champ `point`).

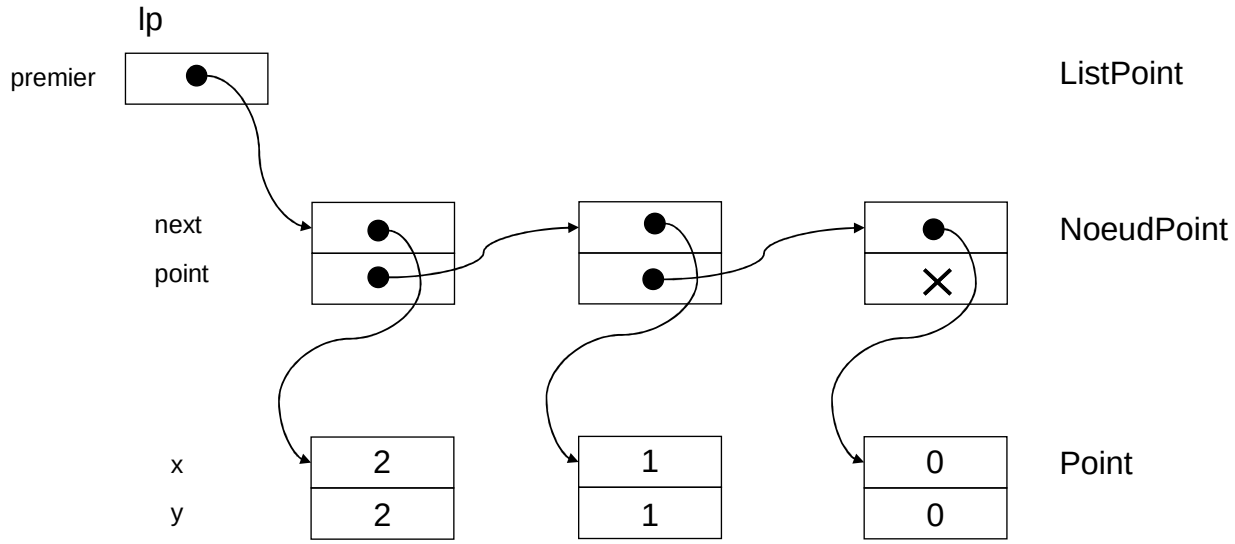


Figure 18 : Représentation d'une liste chaînée de trois points

Sur le principe, c'est la méthode `add` de la classe `ListPoint` qui ajoute un nouveau point. Cette méthode crée un nouvel objet de type `NoeudPoint` qui est ajouté en début de liste, si la liste est vide, le premier nœud créé constitue le premier et le seul nœud de la liste. La Figure 19 montre les étapes de l'ajout d'un objet de type Point à une liste existante (non vide).

- `lp.add(new Point(i = 3, i = 3));` : création d'un nouvel objet de type `Point` et appel de la méthode `add`.
- `NoeudPoint n = new NoeudPoint(p);` : création d'un nouvel objet de type `NoeudPoint`.
- `n.setNext(premier);` le nouvel objet de type `NoeudPoint` référence le premier objet de type `NoeudPoint` de la liste, `premier = n;`: le nouvel objet de type `NoeudPoint` devient le nouveau premier élément de la liste.

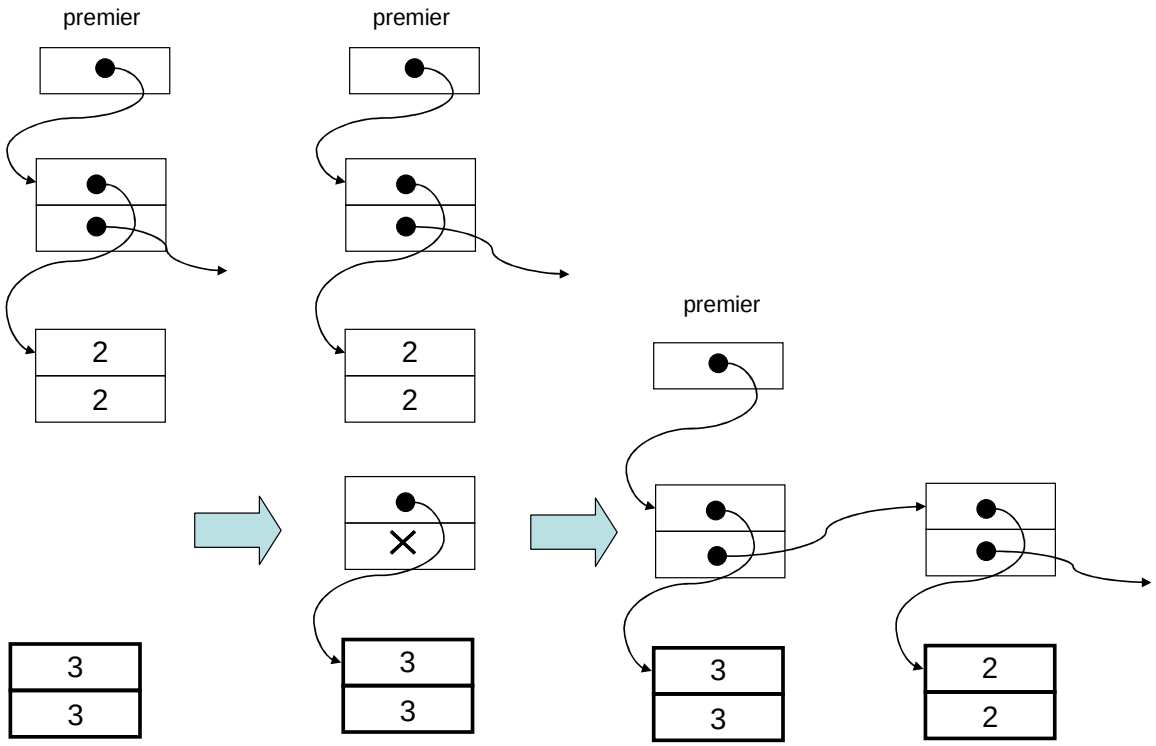


Figure 19 : Ajout d'un point à une liste non vide

Les principales méthodes (`find`, `getPoint` et `size`) de la classe `ListPoint` parcourent la liste du premier au dernier élément (ou à l'élément recherché), le passage à l'élément suivant s'effectue à l'aide de l'accesseur `getNext()` de la classe `NoeudPoint` (`n = n.getNext();`) qui retourne le successeur dans la liste de l'objet courant.

La méthode `toString()` de la classe `ListPoint` qui retourne une chaîne représentant toute la liste, utilise, pour concaténer les chaînes représentant les coordonnées des points de la liste, la classe `StringBuffer` et, en particulier, la méthode `append`, plus efficace que la concaténation de chaînes de type `String` (chaînes nécessitant de nombreuses allocations).

Solution possible :

```
public class Test {
 static public void main(String [] args) {
  ListPoint lp = new ListPoint();
  Point p;
  int i;
  for(i = 0; i < 10; i++) {
   lp.add(new Point(i, i));
  }
  System.out.println(lp);
  System.out.println(lp.size());
  i=lp.find(new Point(2, 2));
  p = lp.getPoint(i);
  if(p != null)System.out.println(p);
  i=lp.find(new Point(2, 3));
  p = lp.getPoint(i);
  if(p != null)System.out.println(p);
```

```
   System.out.println(lp);
  }
 }
 class Point {
  private int x, y;
  public  Point(int nx, int ny) {
     x = nx;
     y = ny;
  }
  public String toString() {
     return "("+ x+", "+ y+")";
  }
  public boolean equals(Point p) {
   return x == p.x && y == p.y;
  }
 }
 class NoeudPoint {
  private NoeudPoint next;
  private Point point;
  public NoeudPoint next() {
   return next;
  }
  public NoeudPoint(Point p){
   point = p;
   next = null;
  }
  public Point getPoint() {
   return point;
  }
  public void setPoint(Point p) {
   point = p;
  }
  public NoeudPoint getNext() {
   return next;
  }
  public void setNext(NoeudPoint n) {
   next = n;
  }
 }
 class ListPoint {
  private NoeudPoint premier;
  int size;
  public ListPoint(){
   premier = null;
   size = 0;
  }
  public void add(Point p){
   NoeudPoint n = new NoeudPoint(p);
   if(premier == null) {
    premier = n;
```

```
   } else {
    n.setNext(premier);
    premier = n;
   }
   size ++;
  }
  public int size() {
   return size;
  }
  public String toString() {
   NoeudPoint n = premier;
   StringBuffer sb = new StringBuffer("[");
   while(n != null) {
    sb.append(n.getPoint());
    sb.append("; ");
    n = n.getNext();
   }
   sb.delete(sb.length()-2, sb.length());
   sb.append("]");
   return sb.toString();
  }
  public int find(Point p) {
   NoeudPoint n = premier;
   int index = 0;
   while(n != null) {
    if (n.getPoint().equals(p)) return index;
    n = n.getNext();
    index ++;
   }
   return -1;
  }
  public Point getPoint(int pos) {
   if(pos < 0 || pos >= size()) return null;
   int index = 0;
   NoeudPoint n = premier;
   while(index != pos) {
    n = n.getNext();
    index ++;
   }
   return n.getPoint();
  }
 }
```

Résultat de l’exécution :

```
[(9, 9); (8, 8); (7, 7); (6, 6); (5, 5); (4, 4); (3, 3); (2, 2); (1,
1); (0, 0)]
10
(2, 2)
```

12.5 Cartes

L'objectif de cet exercice est de définir des classes permettant de représenter et d'aider à la manipulation de cartes à jouer, ici le jeu choisi est la belote. On se limite à compter le nombre de points que vaut une carte donnée en connaissant l'atout. Les différentes cartes se distinguent par leur valeur as, sept à dix, ou leur figure roi, dame, valet et par leur couleur. Les quatre couleurs sont trèfle, carreau, cœur et pique.

Nous avons défini deux énumérations :

- `enum TCarte` associe un type de carte avec son nom : champ `nom`, dont la valeur peut être lue avec la méthode `toString`, sa valeur à l'atout : `valeurAtout` et sa valeur normale : `valeurNormale`. Les valeurs de ces deux champs pouvent être lues avec les accesseurs correspondant `getValeurAtout` et `getValeurNormale`.
- `enum Couleur` associe une couleur avec son nom : champ `nom`, dont la valeur peut être lue avec la méthode `toString`.

La classe `Carte` possède un champ `tcarte` de type `TCarte` et un champ `couleur` de type `Couleur` dont les valeurs caractérisent les objets de type `Carte`. Les valeurs de ces deux champs pouvant être lues avec les accesseurs correspondant `getTCarte` et `getCouleur`. La méthode `getValeur` retourne la valeur effective de l'objet courant de type `Carte` en fonction de la valeur de l'argument qui désigne la couleur de l'atout. La méthode `toString` retourne une chaîne représentant la carte.

La classe `Jeu` quant à elle permet de créer un jeu de 32 cartes, de retourner la $i^{ème}$ carte (méthode `getCarte`) et d'afficher toutes les cartes (méthode `print`).

Par ailleurs, il est possible d'ajouter d'autres méthodes à la classe `Jeu`, comme la méthode `battre` dont on donne une implémentation possible qui consiste à parcourir le tableau `tCarte` en échangeant l'élément courant avec un élément pris au hasard :

```
public void battre() {
  Carte c;
  for(int i = 0; i < tCarte.length; i++) {
   c = tCarte[i];
   int j = (int)(tCarte.length*Math.random());
   tCarte[i] = tCarte[j];
   tCarte[j] = c;
  }
 }
```

Solution possible :

```
public class Test {
 public static void main(String[] args) {
  Jeu j = new Jeu();
  Couleur catout = Couleur.CARREAU;
  Carte c1 = j.getCarte(5);
  System.out.println(c1);
  System.out.println(c1.getValeur(catout));
  c1 = j.getCarte(11);
  System.out.println(c1);
  System.out.println(c1.getValeur(catout));
  }
```

```
}
enum TCarte {
 AS("as", 11, 11), ROI("roi", 4,
4),DAME("dame",3,3),VALET("valet",20,2), DIX("10",10,10),
NEUF("9",14,0),HUIT ("8",0, 0),SEPT("7",0, 0);
 private int valeurAtout, valeurNormale;
 private String nom;
 private TCarte(String nnom, int vAtout, int vNormal) {
  nom = nnom;
  valeurAtout = vAtout;
  valeurNormale = vNormal;
 }
 public int getValeurAtout() {
  return valeurAtout;
 }
 public int getValeurNormale() {
  return valeurNormale;
 }
 public String toString() {
  return nom;
 }
}
enum Couleur {
 TREFLE("trèfle"), CARREAU("carreau"), COEUR("coeur"), PIQUE("pique");
 private String nom;
 private Couleur(String nnom) {
  nom = nnom;
 }
 public String toString() {
  return nom;
 }
}
class Carte {
 private TCarte tcarte;
 private Couleur couleur;
 public Carte(TCarte ntcarte, Couleur ncouleur){
  tcarte = ntcarte;
  couleur = ncouleur;
 }
 public TCarte getTCarte() {
  return tcarte;
 }
 public Couleur getCouleur() {
  return couleur;
 }
 public int getValeur(Couleur atout) {
  if(atout == couleur) return tcarte.getValeurAtout();
  return tcarte.getValeurNormale();
 }
 public String toString() {
```

```
   return tcarte.toString() + " de " + couleur.toString();
  }
 }
 class Jeu {
  private Carte []tCarte;
  public Jeu() {
   tCarte = new Carte[32];
   TCarte []ttc = TCarte.values();
   for(int i = 0; i<ttc.length ; i++){
    tCarte[i] = new Carte(ttc[i], Couleur.TREFLE);
    tCarte[i+8] = new Carte(ttc[i], Couleur.CARREAU);
    tCarte[i+16] = new Carte(ttc[i], Couleur.COEUR);
    tCarte[i+24] = new Carte(ttc[i], Couleur.PIQUE);
   }
  }
  public Carte getCarte(int i) {
   if(i < 0 || i > 31) return null;
   return tCarte[i];
  }
  public void print() {
   for(int i = 0; i < tCarte.length ; i++)
    System.out.println(tCarte[i]);
  }
 }
```

Résultat de l’exécution :

```
9 de trèfle
0
valet de carreau
20
```

C'est par la pensée obscure ou claire, par ce qui a été compris et surtout par ce qui a été voulu, dans l'unité et l'innocence de l'acte, que les êtres se transmettent leur héritage.

Gaston Bachelard

Héritage et polymorphisme

1. Introduction

1.1 Définitions

L'**héritage** est un des fondements de la programmation orientée objet et constitue sans doute l'un des apports les plus intéressants. Cette technique de réutilisation permet de créer une classe à partir d'une autre, en héritant des propriétés de cette dernière. La **classe dérivée** possèdera l'ensemble des propriétés de la **classe de base** et des propriétés additives spécifiques. La nouvelle structure de données ainsi construite possède les membres de la classe de base et ceux de la classe dérivée : la classe dérivée est donc plus spécialisée que la classe de base. De plus, une classe dérivée constitue un sous type de sa classe de base, un objet du type de la classe dérivée est aussi un objet du type de sa classe de base, le polymorphisme le plus avancé repose sur cette notion.

De manière simplifiée, l'héritage est une relation entre classes, comparable à la relation d'inclusion d'un sous-ensemble dans un ensemble, par opposition, à l'instanciation qui est une relation entre objet et classe, comparable à la relation d'appartenance d'un élément à un ensemble.

Lorsqu'une classe ne peut hériter directement que d'une seule classe, l'héritage est dit simple, sinon il est qualifié de multiple. Dans le cas de héritage simple, l'ensemble des classes qui héritent directement d'une classe mère s'appelle une **hiérarchie de classes**, cette hiérarchie peut être représentée par un arbre dont la classe mère constitue la racine.

1.2 Exemple introductif

Considérons l'exemple suivant, la classe `PointCouleur` regroupe les données et les méthodes permettant de représenter un point de couleur et son comportement. On envisage ici des points dont les coordonnées sont entières et dont la couleur est aussi codée par un entier. De plus, supposons que l'on dispose d'une classe `Point` qui permet de représenter des points de coordonnées entières. Le tableau suivant donne une représentation simplifiée de l'interface (partie publique) des deux classes.

Classe Point	Classe PointCouleur

class Point {
private int x, y;
public Point(int, int)
public void translate(int, int)
public boolean equals(Point)
public int getX()
public int getY()
public void setX(int)
public void setY(int)
public String toString()
}
	class PointCouleur {
	private int x, y, c;
	public PointCouleur(int, int, int)
	public void translate(int, int)
	public boolean equals(PointCouleur)
	public int getX()
	public int getY()
	public int getC()
	public void setX(int)
	public void setY(int)
	public void setC(int)
	public String toString()
	}

Tableau 15 : classes Point et PointCouleur

On remarque que la classe `PointCouleur` possède des membres qui lui sont spécifiques : `c`, `getC` et `setC`, elle possède des membre qui ne concernent que sa « partie `Point` » : `translate`, `getX`, `getY`, `setX` et `setY`, et enfin, elle possède aussi des membres qui concernent sa partie spécifique et sa « partie `Point` » : le constructeur `PointCouleur`, `equals`, `toString` (qui manipule les coordonnées et la couleur).

Il va donc falloir dupliquer une partie du code de la classe `Point` dans la classe `PointCouleur` et maintenir le code. En particulier, les modifications du comportement de la classe `Point` devant être mises à jour dans la classe `PointCouleur`. On duplique donc une partie du code ce qui peut, par ailleurs, engendrer des oublis et des erreurs.

2. Bases sur l'héritage

2.1 Définition d'une classe dérivée et droits d'accès

Une classe Java se définit de manière simplifiée :

```
[public] class ClasseBase {
[public | protected | private] définition des champs
    [public | protected | private] définition des méthodes
}
```

Une classe Java héritant d'une autre classe se définit de manière simplifiée :

```
[public] class ClasseDerivee extends ClasseBase {
[public | protected | private] définition des champs
    [public | protected | private] définition des méthodes
}
```

Nous venons de spécifier que la classe `ClasseDerivee` hérite de la classe `ClasseBase` en utilisant le mot clé `extends`. Désormais la classe `ClasseDerivee` possède en plus de ses membres tous les membres de sa classe de base `ClasseBase`. Bien qu'elle possède les membres de sa classe de base, cela n'implique pas qu'elle peut les utiliser. En effet, le principe de dissimulation est toujours respecté, ainsi les membres privés de la classe de base ne sont pas accessibles à la classe dérivée (Si la classe

dérivée tente d'accéder à `membre` privé dans sa classe de base cela conduit à l'erreur de compilation : « `membre has private access in ClasseBase` »).

Dans certains langages de programmation orientés objets, C++ en particulier, une classe peut dériver simultanément de plusieurs classes de base. A l'opposé, dans le cas de l'héritage simple une classe dérivée hérite d'une unique classe de base. C'est ce second choix qui a été fait pour le langage Java (l'utilisation d'interfaces permet de contourner en partie cette difficulté). Les énumérations ne peuvent pas hériter ou être héritées.

Droits d'accès des membres de la classe de base : `public`, rien `private` ou `protected`, ces droits sont classés du plus restrictif au moins restrictif

- `private` : les membres privés ne sont accessibles que par les méthodes (fonctions membres) de la classe. Même les méthodes d'une classe dérivée ne peuvent accéder aux membres privés de sa classe de base.
- `rien` : tous ces membres sont accessibles à partir de la classe, à partir d'une autre classe, en particulier une classe dérivée, du même fichier ou d'un fichier du même répertoire (c'est le modificateur par défaut), on le qualifie d'accès « friendly ».
- `protected` : tous ces membres sont accessibles à partir de la classe, à partir d'une autre classe du même fichier ou d'un fichier du même répertoire et des classes dérivées[21].
- `public` : les membres publics sont accessibles par tous, à partir de la classe, à partir d'une autre classe, en particulier les classes dérivées, du même fichier ou à partir d'une classe d'un autre fichier.

Les classes ne peuvent être que de type publique ou de friendly. Le mode d'accès qui s'applique effectivement aux membres est le mode le plus restrictif entre celui de la classe et celui qui qualifie le membre. Par exemple, un membre privé d'une classe publique sera privé, et un membre protégé d'une classe friendly sera friendly.

Les constructeurs de la classe de base ne sont pas accessibles de manière explicite dans la classe dérivée, mais eux aussi sont hérités.

2.2 Suite de l'exemple introductif

Nous allons reprendre l'exemple introductif du 1.2, la classe `PointCouleur` regroupe les données et les méthodes permettant de représenter un point de couleur qui va hériter de la classe `Point`. Afin de spécifier que la classe `Pointcouleur` hérite de la classe `Point`, il suffit d'utiliser le mot clé `extends` : `class PointCouleur extends Point`. On peut remarquer que les objets de type `PointCouleur` peuvent invoquer les méthodes de la classe `Point`, par exemple dans `main`, l'objet de type `PointCouleur` référencé par `pc1` appelle la méthode `translate` (définie dans la classe `Point` et pas définie dans `PointCouleur`). De même, au sein de la classe `PointCouleur` les méthodes de la classe `Point` peuvent être directement appelées, de la même manière que si elles sont définies dans la classe dérivée `PointCouleur`, par exemple les accesseurs `getX` et `getY` et les altérateurs `setX` et `setY`. La Figure 20 représente la variable `pc1` référençant un objet de type `PointCouleur`, l'objet référencé est constitué d'une partie regroupant les membres

[21] En C++, l'accès protégé est plus restrictif, seule la classe de base et ses classes dérivées peuvent accéder aux membres protégés de la classe de base

de la classe `Point` et d'une autre partie regroupant les membres spécifiques à la classe `PointCouleur`.

Cependant, dans l'état actuel, on peut remarquer quelques détails :

- Si l'initialisation d'un `Point` est complexe, l'utilisation d'un altérateur n'est peut être pas suffisante. L'initialisation de la « partie `Point` » dans le constructeur de `PointCouleur` nécessite que le concepteur de la classe `PointCouleur` ait une connaissance du comportement de la classe `Point` qu'il n'a pas forcément
- La méthode `equals` de `PointCouleur` refait le travail effectué dans celle de Point (de même avec `toString`), comme pour le cas du constructeur, cela implique qu'au niveau de `PointCouleur` on connaît des détails sur le comportement des points

Exemple :

```
public class Test {
 static public void main(String [] args) {
  PointCouleur pc1 = new PointCouleur(1, 2, 3);
  System.out.println(pc1);
  pc1.translate(3,3);
  System.out.println(pc1);
  System.out.println(pc1.getX());
  PointCouleur pc2 = new PointCouleur(4, 5, 3);
  System.out.println(pc1.equals(pc2));
  System.out.println(pc2);
  pc2.setX(2);
  System.out.println(pc1.equals(pc2));
 }
}
class Point {
 private int x, y;
 public  Point(int nx, int ny) {
    x = nx;
    y = ny;
 }
 public  Point(){
  this(0, 0);
 }
 public void translate(int dx, int dy) {
    x += dx;
    y += dy;
 }
 public boolean equals(Point p) {
  if(p.x == x && p.y == y) return true;
  return false;
 }
 public int getX() {
  return x;
 }
 public int getY() {
```

```
   return y;
  }
  public void setX(int nx) {
   x = nx;
  }
  public void setY(int ny) {
   y = ny;
  }
  public String toString() {
   return "(" + x + ", " + y + ")";
  }
}
class PointCouleur extends Point{
  private int  c;
  public  PointCouleur(int nx, int ny, int nc) {
     setX(nx);
     setY(ny);
     c = nc;
  }
  public boolean equals(PointCouleur p) {
   if(p.getX() == getX() && p.getY() == getY() && p.c == c) return true;
   return false;
  }
  public int getC() {
   return c;
  }
  public void setC(int nc) {
   c = nc;
  }
  public String toString() {
   return "(" + getX() + ", " + getY() + ")" + ":" +c;
  }
}
```

Résultat de l'exécution :

```
(1, 2):3
(4, 5):3
4
true
(4, 5):3
False
```

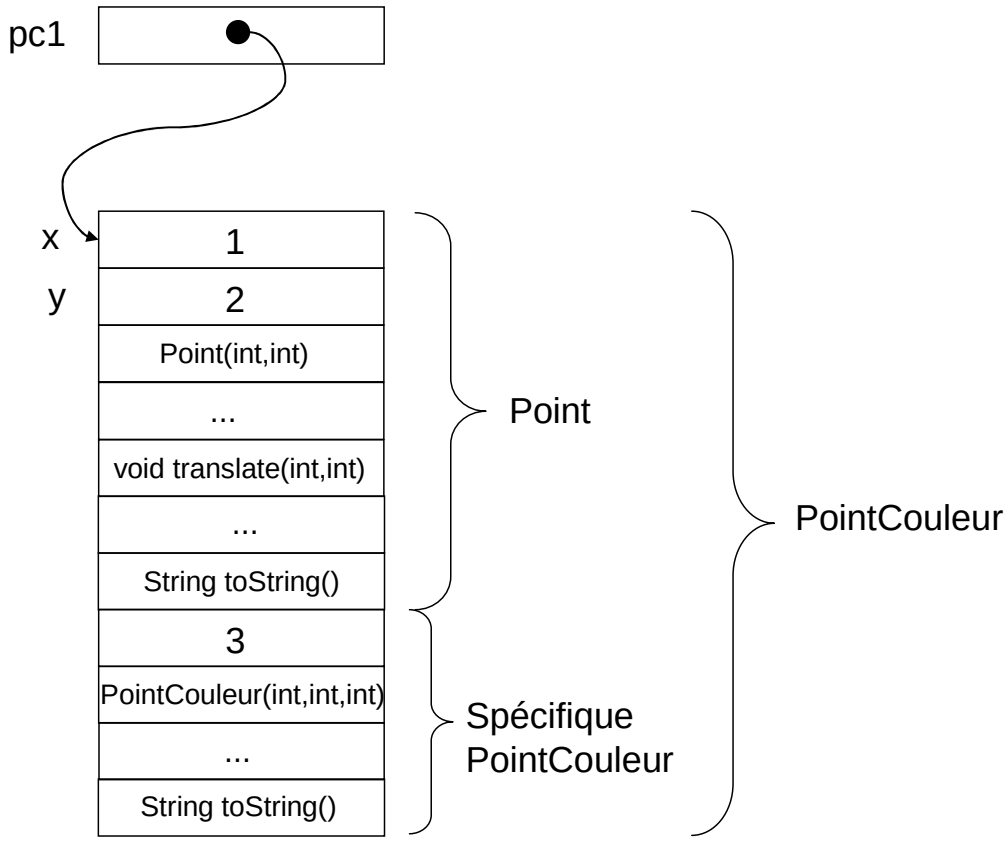


Figure 20 : Représentation d'un objet d'une classe dérivée

3. Constructeurs

3.1 Le mot clé super

Les constructeurs d'une classe sont des méthodes particulières qui permettent de créer et d'initialiser les objets appartenant à cette classe. Le constructeur est appelé lors de la création d'un nouvel objet par `new`. Dans le cas de l'héritage, la partie héritée doit être aussi initialisée, le constructeur de la classe de base ne peut pas être appelé directement et l'utilisation d'altérateur n'est pas non plus une solution satisfaisante (du reste, il n'y en pas toujours dans le cas de classes immuables). Java permet tout de même d'invoquer les constructeurs de la classe de base via le mot clé `super`. Comme dans le cas de `this`, on peut considérer que `super` constitue la référence de la partie classe de base de l'objet courant, `this` étant la référence de l'objet courant de la classe dérivée. L'appel du constructeur de la classe de base via `super(…)` doit être la première instruction du bloc du constructeur de la classe dérivée.

L'exemple suivant illustre l'utilisation du mot clé `super` dans le constructeur général de `PointCouleur`. Ainsi le constructeur de `Point` sera appelé avant de mettre à jour le champ `c` de `PointCouleur`.

Exemple :

```
public class Test {
 static public void main(String [] args) {
  PointCouleur pc1 = new PointCouleur(1, 2, 3);
  System.out.println(pc1);
 }
```

```
}
class Point {
 private int x, y;
 public  Point(int nx, int ny) {
    x = nx;
    y = ny;
 }
 public  Point(){
  this(0, 0);
 }
 public int getX() {
  return x;
 }
 public int getY() {
  return y;
 }
}
class PointCouleur extends Point{
 private int  c;
 public  PointCouleur(int nx, int ny, int nc) {
  super(nx, ny);
  c = nc;
 }
 public  PointCouleur() {
  this(0, 0, 0);
 }
 public String toString() {
  return "(" + getX() + ", " + getY() + ")" + ":" +c;
 }
}
```

Résultat de l'exécution :

```
(1, 2):3
```

Comme le mot `this` doit être la première instruction du bloc du constructeur de la classe dérivée, cela signifie qu'on ne peut pas avoir à la fois de `this` et de `super` dans le constructeur d'une classe dérivée (on provoque dans ce cas une des deux erreurs de compilation : « `call to super must be first statement in constructor` » ou « `call to this must be first statement in constructor` »). Il faut faire un choix entre les deux. De manière générale, il faut privilégier le constructeur général de la classe dérivée qui appelle le constructeur général de sa classe de base via `super`, tous les autres constructeurs de la classe dérivée appelant le constructeur général de celle-ci via `this`.

3.2 Cycle de vie des objets et ordre d'appel des constructeurs

Lorsqu'une classe dérive d'une autre classe, la partie relative à la classe de base des objets de la classe dérivée est construite (allouée et initialisée) avant la partie spécialisée. Comme pour le cas des classes simples, le processus commence par les initialisations des champs (après l'allocation de mémoire). En résumé :

1. Initialisation des champs de la classe de base
2. Exécution du constructeur de la classe de base

3. Initialisation des champs de la classe de dérivée
4. Exécution du constructeur de la classe de dérivée (à partir de l'instruction suivant le super)

L'exemple suivant qui reprend le précédent mais en ajoutant des affichages dans les constructeurs illustre la façon dont les objets d'une classe dérivée sont effectivement créés (la Figure 21 schématise l'enchaînement des appels des constructeurs à partir du `pc1 = new PointCouleur(1, 2, 3)`).

Exemple :

```
public class Test {
 static public void main(String [] args) {
  PointCouleur pc1 = new PointCouleur(1, 2, 3);
  System.out.println(pc1);
  PointCouleur pc2 = new PointCouleur();
  System.out.println(pc2);
 }
}
class Point {
 private int x, y;
 public  Point(int nx, int ny) {
  System.out.println("Point(int nx, int ny)");
    x = nx;
    y = ny;
 }
 public  Point(){
  this(0, 0);
  System.out.println("public  Point()");
 }
 public int getX() {
  return x;
 }
 public int getY() {
  return y;
 }
}
class PointCouleur extends Point {
 private int  c;
 public PointCouleur(int nx, int ny, int nc) {
  super(nx, ny);
  System.out.println("PointCouleur(int nx, int ny, int nc)");
  c = nc;
 }
 public  PointCouleur() {
  this(0, 0, 0);
  System.out.println("PointCouleur()");
 }
 public String toString() {
  return "(" + getX() + ", " + getY() + ")" + ":" +c;
 }
```

```
}
```

Résultat de l'exécution :

```
Point(int dx, int dy)
PointCouleur(int nx, int ny, int nc)
(1, 2):3
Point(int dx, int dy)
PointCouleur(int nx, int ny, int nc)
PointCouleur()
(0, 0):0
```

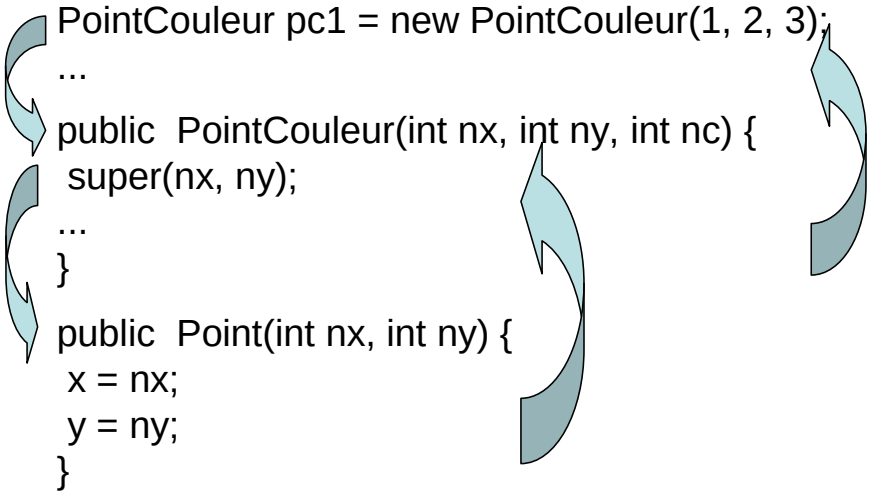


Figure 21 : Appel de constructeur et retour avec super

3.3 Constructeur par défaut

Dans le cas d'une classe qui dérive d'une classe de base et en l'absence de `super` ou de `this` dans un des constructeurs de la classe dérivée, le constructeur par défaut de la classe de base est automatiquement appelé. Dans ce cas, on peut considérer que le compilateur synthétise alors l'instruction `super()` en tant que première instruction. En l'absence de tout constructeur, le compilateur Java en synthétise un de type constructeur par défaut (sans paramètre). Ce constructeur ne fait rien, mais, dans le cas d'une classe dérivée, il appelle le constructeur par défaut de sa classe de base.

Dans l'exemple suivant, lorsque l'objet de type `PointCouleur` référencé par la variable `pc2` est crée, le constructeur `PointCouleur()` appelle d'abord le constructeur par défaut de `Point`, `Point()` bien qu'il n'y ait pas de `super()` dans `PointCouleur()`. Maintenant, si la classe `Point` possède comme unique constructeur `Point(int nx, int ny)`, comme le constructeur `PointCouleur()` tente d'appeler le constructeur par défaut de la classe de base `Point()`, on provoque alors l'erreur de compilation « `cannot find symbol constructor Point()` ». C'est la raison pour laquelle, afin de faciliter l'héritage, il vaut mieux que toute classe possède un constructeur par défaut.

Exemple :

```
public class Test {
 static public void main(String [] args) {
  PointCouleur pc2 = new PointCouleur();
  System.out.println(pc2);
 }
}
```

```
class Point {
 private int x, y;
 public  Point(int nx, int ny) {
  System.out.println("Point(int nx, int ny)");
    x = nx;
    y = ny;
 }
 public  Point(){
  this(0, 0);
  System.out.println("public  Point()");
 }
 public int getX() {
  return x;
 }
 public int getY() {
  return y;
 }
}
class PointCouleur extends Point{
 private int  c;
 public  PointCouleur(int nx, int ny, int nc) {
  super(nx, ny);
  System.out.println("PointCouleur(int nx, int ny, int nc)");
    c = nc;
 }
 public  PointCouleur() {
  System.out.println("PointCouleur()");
 }
 public String toString() {
  return "(" + getX() + ", " + getY() + ")" + ":" +c;
 }
}
```

Résultat de l'exécution :

```
Point(int nx, int ny)
public  Point()
PointCouleur()
(0, 0):0
```

4. Redéfinition et surdéfinition

4.1 Définition et principes

Il est possible de définir dans une classe dérivée des méthodes de même signature (de même nom et dont les paramètres sont de même type) que celles de sa classe de base, en général, à des fins de spécialisation. Il ne s'agit pas d'une surdéfinition (*overloading* en anglais), mais d'une **redéfinition** (*overriding* en anglais). On dit aussi qu'il y a **outrepassement** de la méthode de la classe de base par celle de la classe dérivée. Bien évidemment, il faut que la méthode outrepassée ne soit pas privée, sinon elle n'est pas accessible. Il n'est pas non plus possible de redéfinir un constructeur, dans la mesure où le nom du constructeur est identique à celui de sa classe. Lorsqu'une méthode de la classe dérivée surcharge une méthode de la classe de base, il s'agit d'un ajout de

méthode (les deux méthodes coexistent), dans le cas d'une méthode qui outrepasse une méthode de la classe de base, il s'agit d'un remplacement.

En cas d'outrepassement, la méthode redéfinie dans la classe dérivée masque la méthode correspondante dans sa classe de base, les objets de la classe dérivée appelleront la méthode redéfinie. Dans des cas plus compliqués, où redéfinition et surcharge se mêlent, lorsqu'un objet d'une classe dérivée appelle une méthode, le choix de la « bonne méthode » à appeler s'effectue schématiquement de la manière suivante :

1. Le compilateur recherche d'abord une correspondance exacte dans la classe (dérivée) de l'objet
2. Le compilateur recherche une correspondance exacte dans la classe de base de l'objet, puis il continue à remonter la hiérarchie de classe
3. Si aucune méthode n'a encore été sélectionnée, le choix s'effectue par le biais de la surcharge, en partant de la classe de base et en remontant la hiérarchie de classe
4. Sinon, c'est le message d'erreur (« `cannot find symbol method …` »).

Donc, il y a priorité de la recherche par remontée dans la hiérarchie de classes plutôt que par recherche des types compatibles.

L'exemple suivant illustre la manière dont sont sélectionnées les méthodes surdéfinies et redéfinies par le compilateur Java. Pour faciliter la compréhension de l'exemple, un affichage correspondant à l'entête de la méthode est ajouté au début de chaque méthode. Il s'agit de la méthode `coincide` dont deux versions sont définies dans la classe de base `Point` (il s'agit d'une surcharge) :

- `coincide(int nx, int ny)` surcharge `coincide(Point p)` et réciproquement

Trois versions de `coincide` définies dans la classe `PointCouleur` se surchargent :

- `coincide(Point p)`, `coincide(PointCouleur p)` et `coincide(long nx, long ny)`

Deux versions de `coincide` de `PointCouleur` surchargent les `coincide` de `Point`

- `coincide(PointCouleur p)` et `coincide(long nx, long ny)`

Une version de `coincide` redéfinie (outrepasse) la version de `coincide` de même signature :

- `coincide(Point p)`

Etudions le comportement de ce programme, appel par appel, de la méthode coincide :

- `pc.coincide(pca)` : `pca` est de type `PointCouleur`, il y a une méthode `coincide` dont le type du paramètre correspond exactement dans `PointCouleur` : `coincide(PointCouleur p)` et qui est appelée (on est dans le cas 1).
- `pc.coincide(2l, 2l)` : les deux arguments sont de type `long`, il y a une méthode `coincide` dont les types des paramètres correspondent exactement dans `PointCouleur` : `coincide(long nx, long ny)` et qui est appelée (on est encore dans le cas 1).
- `pc.coincide(p)`: `p` est de type `Point`, il y a une méthode `coincide` dont le type du paramètre correspond exactement dans `PointCouleur` :

`coincide(PointCouleur p)` et qui est appelée (on est dans le cas 1), bien qu'il en existe une dans `Point` qui soit candidate.

- `pc.coincide(2, 2)` : les deux arguments sont de type `int`, comme il n'y a pas de méthode qui corresponde dans `PointCouleur`, on remonte la hiérarchie de classe (→`Point`), il y a une méthode `coincide` dont les types des paramètres correspondent exactement dans `Point` : `coincide(int nx, int ny)` est appelée (on est encore dans le cas 2), bien que la méthode `PointCouleur coincide(long nx, long ny)` soit candidate.

Concernant l'appel `pc.coincide(2, 2)`, si il n'y avait pas eu la méthode `coincide(int nx, int ny)` dans `Point`, alors la méthode `coincide(long nx, long ny)` de Point aurait été appelée (on aurait été dans le cas 3). Si il n'y avait pas eu la méthode `coincide(long nx, long ny)` dans `PointCouleur`, alors lors de l'appel `pc.coincide(2l, 2l)` aurait échoué et provoqué l'erreur de compilation « `cannot find symbol method coincide(long,long)` » (on aurait été dans le cas 3).

Exemple :

```
public class Test {
 static public void main(String [] args) {
  PointCouleur pc = new PointCouleur(1,2,3);
  PointCouleur pca = new PointCouleur(1,2,3);
  Point p = new Point(1,2);
  System.out.println(pc.coincide(pca));
  System.out.println(pc.coincide(2l, 2l));
  System.out.println(pc.coincide(p));
  System.out.println(pc.coincide(2,2));
 }
}
class Point {
 private int x, y;
 public  Point(int nx, int ny) {
    x = nx;
    y = ny;
 }
 public boolean coincide(Point p) {
  System.out.println("Point : coincide(Point p)");
  return p.x == x && p.y == y;
 }
 public boolean coincide(int nx, int ny) {
  System.out.println("Point : coincide(int nx, int ny)");
  return nx == x && ny == y;
 }
 public int getX() {
  return x;
 }
 public int getY() {
  return y;
 }
 public String toString() {
  return "(" + x + ", " + x + ")";
```

```
  }
 }
 class PointCouleur extends Point{
  private int  c;
  public  PointCouleur(int nx, int ny, int nc) {
   super(nx, ny);
     c = nc;
  }
  public String toString() {
   return "(" + getX() + ", " + getY() + ")" + ":" +c;
  }
  public boolean coincide(Point p) {
   System.out.println("PointCouleur : coincide(Point p)");
   return p.getX() == getX() && p.getY() == getY();
  }
  public boolean coincide(PointCouleur p) {
   System.out.println("PointCouleur : coincide(PointCouleur p)");
   return p.getX() == getX() && p.getY() == getY() && p.c == c;
  }
  public boolean coincide(long nx, long ny) {
   System.out.println("PointCouleur : coincide(long nx, long ny)");
   return getX() == nx && getY() == ny ;
  }
 }
```

Résultat de l'exécution :

```
false
PointCouleur : coincide(Point p)
true
Point : coincide(int nx, int ny)
false
Point : coincide(int nx, int ny)
false
```

En proposant d'utiliser un même nom de méthode pour plusieurs types d'objets différents, on obtient une forme de polymorphisme plus avancée que dans le cas de la surcharge, c'est la combinaison du type de l'objet et de la signature de la méthode qui permet de choisir la bonne méthode à exécuter (nous verrons dans la suite encore d'autres possibilités).

4.2 Contraintes

Pour qu'une méthode puisse outrepasser celle de sa classe mère, il faut qu'elle vérifie quelques contraintes[22]. Le droit d'accès de la méthode redéfinie doit être identique ou moins restrictif que celui de la méthode outrepassée. Si on restreint l'accès des méthodes redéfinies, cela signifie que l'on privilégie la méthode outrepassée dans la classe de base, ce qui est contradictoire. Le tableau suivant spécifie les règles, la colonne indique le droit d'accès de la méthode outrepassée dans la classe de base et la ligne du

[22] Il faut aussi qu'elle traite au moins les exceptions de sa classe de base, ce que nous ne traitons pas ici.

haut indique le droit d'accès de la méthode redéfinie dans la classe dérivée. Par exemple une méthode protégée ne pourra pas être redéfinie par une méthode privée, mais pourra l'être par une méthode publique.

↗	**privée**	**friendly**	**protégée**	**publique**
privée	oui	oui	oui	oui
friendly	non	oui	oui	oui
protégée	non	non	oui	oui
publique	non	non	non	oui

Tableau 16 : Droits d'accès et redéfinition

Dans l'exemple suivant, la méthode publique `coincide` de la classe `Point` ne peut pas être outrepassée par la méthode protégée `coincide` de la classe `PointCouleur` (`protected` étant plus restrictif que `public`). Cela provoque l'erreur de compilation « `coincide(Point) in PointCouleur cannot override coincide(Point) in Point; attempting to assign weaker access privileges; was public` ».

Exemple :

```
class Point {
 private int x, y;
 …
 public boolean coincide(Point p) {
  System.out.println("Point : coincide(Point p)");
  return p.x == x && p.y == y;
 }
 …
}
class PointCouleur extends Point{
 private int  c;
 …
 protected boolean coincide(Point p) {
  System.out.println("PointCouleur : coincide(Point p)");
  return p.getX() == getX() && p.getY() == getY();
 }
}
```

Dans le cadre de la redéfinition d'une méthode, il faut que la signature de la méthode de la classe dérivée soit exactement la même que celle de la classe mère. Le type retour d'une méthode que l'on redéfinit peut être différent de celui de la classe mère, mais il doit être obligatoirement une classe dérivée de la classe constituant le type de retour de la méthode outrepassée. Cette règle[23] se nomme la **covariance** des types de retour. Elle assure, d'une part, la cohérence sémantique lors de la redéfinition (normalement, la redéfinition d'une méthode constitue un raffinement du comportement d'une méthode) et permet d'éviter des conversions de type inutiles. Ce mécanisme fonctionne aussi sur des tableaux d'objets, la covariance portant sur le type de base. Par contre, il ne fonctionne pas avec les types primitifs ou des tableaux de type primitif.

[23] Cette possibilité a été introduite dans le JDK 5.0, avant il fallait coïncidence exacte des types de retour.

L'exemple qui suit est caractéristique, il s'agit de la méthode `clone` qui retourne une copie de l'objet courant. Les méthodes `clone` n'ayant pas de paramètre, la méthode `clone` de `PointCouleur` outrepasse celle de `Point`, bien évidement, chacune doit retourner un objet du type de sa classe. Le second cas `clone(int n)` illustre ce même mécanisme, mais avec des tableaux d'objets.

Exemple :

```
public class Test {
 static public void main(String [] args) {
  PointCouleur pca = new PointCouleur(1,2,3);
  PointCouleur pcb = pca.clone();
  System.out.println(pcb + " <-> " + pca);
  PointCouleur []tpc = pca.clone(2);
  System.out.println(tpc[0]);
 }
}
class Point {
 private int x, y;
 public  Point(int nx, int ny) {
    x = nx;
    y = ny;
 }
 public Point clone() {
  return new Point(x, y);
 }
 public int getX() {
  return x;
 }
 public int getY() {
  return y;
 }
 public String toString() {
  return "(" + x + ", " + x + ")";
 }
 public Point[] clone(int n) {
  Point []t = new Point [n];
  for(int i = 0; i < n; i++) {
   t[i] = clone();
  }
  return t;
 }
}
class PointCouleur extends Point{
 private int  c;
 public  PointCouleur(int nx, int ny, int nc) {
  super(nx, ny);
    c = nc;
 }
 public String toString() {
  return "(" + getX() + ", " + getY() + ")" + ":" +c;
```

```
  }
  public PointCouleur clone() {
   return new PointCouleur(getX(), getY(), c);
  }
  public PointCouleur[] clone(int n) {
   PointCouleur []t = new PointCouleur [n];
   for(int i = 0; i < n; i++) {
    t[i] = clone();
   }
   return t;
  }
 }
```

Résultat de l'exécution :

```
(1, 2):3 <-> (1, 2):3
(1, 2):3
```

4.3 Mot clé super

Le mot clé `super` désigne la référence de la « partie classe de base » de l'objet courant. Ce mot clé n'a de sens que dans la définition d'une classe dérivée. Dans la définition d'une classe dérivée, lorsque l'on manipule les membres (champs ou méthodes) de la classe de base, ceux-ci sont implicitement référencés par rapport à `super` (désignation relative). Ce mot clé joue un rôle similaire à celui de `this`. Par contre, il ne peut être utilisé qu'une seule fois, on ne peut accéder aux membres de la classe de base de la classe de base, `super.super.XXX` n'a pas de sens. Bien évidemment, le mot clé `super` ne permet de désigner que des membres accessibles (les membres privés de la classe de base ne le sont pas).

Pour invoquer la méthode `getX` de la classe de base écrire, on a `super.getX()` ⇔ `getX()`. Les deux écritures suivantes de la méthode `toString` de `PointCouleur` sont équivalentes,

```
  public String toString() {
   return "(" + getX() + ", " + getY() + ")" + ":" +c;
  }
  public String toString() {
   return "(" + super.getX() + ", " + super.getY() + ")" + ":" +c;
  }
```

La référence `super` a de nombreuses utilités. Nous l'avons déjà utilisée pour invoquer le constructeur de la classe de base dans une classe dérivée. Mais on peut utiliser « `super` » comme technique de résolution de portée afin de différencier des membres homonymiques entre ceux de la classe de base et ceux de la classe dérivée. Dans la méthode `toString`, il aurait été plus simple d'utiliser la méthode `toString` de `Point`, plutôt que de passer par les accesseurs `getX` et `getY` et de redéfinir (ici dupliquer) un traitement qui concerne la classe de base `Point`. Or un appel direct de `toString` de `PointCouleur` provoque l'appel du `toString` de `PointCouleur` et pas celui de `Point`. En préfixant par « `super.` », c'est bien celui de `Point` qui est appelé.

Exemple :

```
public class Test {
 static public void main(String [] args) {
  PointCouleur pca = new PointCouleur(1,2,3);
```

```
   System.out.println(pca);
  }
}
class Point {
 private int x, y;
 public  Point(int nx, int ny) {
    x = nx;
    y = ny;
 }
 public Point clone() {
  return new Point(x, y);
 }
 public String toString() {
  return "(" + x + ", " + x + ")";
 }
}
class PointCouleur extends Point{
 private int  c;
 public  PointCouleur(int nx, int ny, int nc) {
  super(nx, ny);
  c = nc;
 }
 public String toString() {
  return super.toString() + ":" +c;
 }
}
```

Résultat de l'exécution :

```
(1, 1):3
```

4.4 Masquage des champs

Comme pour les méthodes, il est possible de redéfinir les champs d'une classe de base dans sa classe dérivée, dans ce cas, il n'y a aucune contrainte sur les droits d'accès. On peut alors considérer que le mécanisme qui s'applique est du **masquage**. En cas d'homonymie des champs d'une classe de base et de sa classe dérivée, les champs homonymiques désignent ceux de la classe dérivée (dans la classe dérivée). Par contre, les champs homonymiques de la classe de base existent toujours, et sont accessibles en utilisant le mot clé `super` (sauf s'ils sont privés[24]). De plus, il faut éviter de donner le même nom à des champs qui ne représentent pas la même chose.

L'exemple suivant qui est abstrait montre le mécanisme de masquage et les possibilités d'utilisation de `super` avec des champs. Dans la méthode `methode` :

- `x` désigne le champ `x` de `ClasseD`
- `y` désigne le champ `y` de `ClasseD`
- `z` désigne le champ `z` de `ClasseB`
- `super.y` désigne le champ `y` de `ClasseB`
- `super.z` désigne le champ `z` de `ClasseB`

[24] Généralement, comme les champs sont privés, ils sont inaccessibles.

Si nous avions tenté d'écrire l'expression `super.x` dans la méthode `methode`, nous aurions provoqué l'erreur de compilation « `x has private access in ClasseB` », le champ `x` étant privé dans `ClasseB`.

Exemple :

```
public class Test {
 static public void main(String [] args) {
  ClasseD ocd = new ClasseD();
  ocd.methode();
 }
}
class ClasseB {
 private int x = 1;
 public int y = 2;
 public int z = 3;
}
class ClasseD extends ClasseB{
 private int x = 4;
 private int y = 5;
 public void methode() {
  System.out.println(x);
  System.out.println(y);
  System.out.println(z);
  System.out.println(super.y);
  System.out.println(super.z);
 }
}
```

Résultat de l'exécution :

```
4
5
3
2
3
```

5. Membres statiques

5.1 Particularités

Les règles de redéfinition et de masquage des membres statiques sont les mêmes que celles des membres non statiques. Une méthode statique (`static`) peut outrepasser celle de sa classe de base, si elle respecte la règle sur le droit d'accès et la règle sur la covariance des types de retour. Par ailleurs, un champ statique de la classe dérivée peut masquer un champ statique de sa classe de base.

Du reste, un champ statique de la classe dérivée peut masquer[25] un champ non statique de sa classe de base, la réciproque est vraie. Mais, une méthode statique de la classe dérivée ne peut pas masquer une méthode non statique de sa classe de base (provoque une erreur de compilation du type « `methode() in ClasseD cannot`

[25] Le terme redéfinition n'aurait ici aucun sens

`override methode() in ClasseB; overriding method is static` »). Une méthode non statique de la classe dérivée ne peut pas, non plus, masquer une méthode statique de sa classe de base (provoque une erreur de compilation du type « `methode() in ClasseD cannot override methode() in ClasseB; overridden method is static` »).

Dans l'exemple qui suit, la classe `PointCouleur` hérite de la classe `Point`. Dans chacune de ces classes est défini un champ statique `compteur` dont la valeur correspond au nombre d'instances de la classe d'appartenance créés. Dans chacune des classes, la méthode statique `getCompteur` retourne la valeur du champ statique `compteur`. Le champ statique `compteur` de `PointCouleur` masque celui de Point et la méthode `getCompteur` de `PointCouleur` redéfinit celle de `Point`. Par exemple, l'appel `PointCouleur.getCompteur()` correspond à l'appel de la méthode `getCompteur` de la classe `PointCouleur`.

Exemple :

```
public class Test {
 static public void main(String [] args) {
  int i;
  Point []tp = new Point[5];
  PointCouleur []tpc = new PointCouleur[10];
  System.out.println(Point.getCompteur());
  System.out.println(PointCouleur.getCompteur());
  for(i = 0; i < tpc.length; i++) {
   tpc[i] = new PointCouleur(i, i+1, i+2);
  }
  System.out.println(Point.getCompteur());
  System.out.println(PointCouleur.getCompteur());
  for(i = 0; i < tp.length; i++) {
   tp[i] = new Point(2*i, 2*i);
  }
  System.out.println(Point.getCompteur());
  System.out.println(PointCouleur.getCompteur());
 }
}
class Point {
 private int x, y;
 private static int compteur = 0;
 public  Point(int nx, int ny) {
    x = nx;
    y = ny;
    compteur ++;
 }
 public Point clone() {
  return new Point(x, y);
 }
 public String toString() {
  return "(" + x + ", " + x + ")";
 }
 public static int getCompteur() {
  return compteur;
```

```
 }
}
class PointCouleur extends Point{
 private int  c;
 private static int compteur = 0;
 public  PointCouleur(int nx, int ny, int nc) {
  super(nx, ny);
    c = nc;
    compteur ++;
 }
 public String toString() {
  return super.toString() + ":" +c;
 }
 public static int getCompteur() {
  return compteur;
 }
}
```

Résultat de l'exécution :

```
0
0
10
10
15
10
```

5.2 Ordre de création

Lors de la création d'un nouvel objet d'une classe dérivée, tout ce qui constitue la partie statique est d'abord exécuté, puis ce qui concerne l'instance. Pour chacune de ces parties, c'est d'abord ce qui concerne la classe de base qui l'est, puis ce qui concerne la partie spécifique de la classe dérivée. Si il y a plusieurs niveaux d'héritage, le processus commence par la classe de plus haut niveau (la classe de base de la classe de base …).

A la création du premier objet ou à l'utilisation du premier membre statique d'une classe dérivée d'une classe de base (une seule fois)

1. Initialisation des champs statiques (explicite ou par défaut) de la classe de base
2. Exécution des blocs d'initialisation statiques dans leur ordre de la classe de base
3. Initialisation des champs statiques (explicite ou par défaut) de la classe dérivée
4. Exécution des blocs d'initialisation statiques dans leur ordre de la classe dérivée

A chaque création d'un nouvel objet de la classe dérivée

5. Allocation dynamique de l'objet
6. Initialisation des champs non statiques (explicite ou par défaut) de la classe de base
7. Exécution des blocs d'initialisation non statiques dans leur ordre d'apparition de la classe de base
8. Exécution du ou des constructeurs de la classe de base

9. Initialisation des champs non statiques (explicite ou par défaut) de la classe dérivée
10. Exécution des blocs d'initialisation non statiques dans leur ordre d'apparition de la classe dérivée
11. Exécution du ou des constructeurs de la classe dérivée

Le programme suivant illustre ce processus :

`cd1 = new ClasseDerivee();` qui va créer la première instance de `ClasseDerivee` référencée par `cd1` :

Partie statique

Classe de base

- le champ statique `vclasseBase` de `classeBase` est initialisé (`vclasseBase←1`)
- le bloc d'initialisation statique de `classeBase` est exécuté (`vclasseBase←2`)

Classe dérivée

- le champ statique `vclasseDerivee` de `ClasseDerivee` est initialisé (`vclasseDerivee ←31`)
- le bloc d'initialisation statique de `ClasseDerivee` est exécuté (`vclasseDerivee ←32`)

Partie non statique

Classe de base

- le champ `vinstanceBase` de `classeBase` est initialisé (`vinstanceBase ←10`)
- le bloc d'initialisation de `classeBase` est exécuté (`vinstanceBase ←11`)
- le constructeur `ClasseBase()` est exécuté (`vclasseBase←3` et `vinstanceBase ←12`)

Classe dérivée

- le champ `vinstanceDerivee` de `classeDerivee` est initialisé (`vinstanceDerivee ←20`)
- le bloc d'initialisation de `classeDerivee` est exécuté (`vinstanceDerivee ←21`)
- le constructeur `ClasseDerivee()` est exécuté (`vclasseDerivee←33` et `vinstanceDerivee ←22`)

Lors de la création `cd2 = new ClasseDerivee();` qui va créer la seconde instance de `ClasseDerivee` référencée par `cd2`, seule la seconde partie du processus est exécutée (la partie statique ne l'est pas). Lors de la création de l'objet référencé par `cb` qui est une instance `ClasseBase` seule la seconde partie du processus concernant `ClasseBase` est exécutée.

Exemple :

```
public class Test {
 static public void main(String [] args) {
  ClasseDerivee cd1, cd2;
  System.out.println("------------------------------");
  cd1 = new ClasseDerivee();
  System.out.println("------------------------------");
```

```
   cd1 = new ClasseDerivee();
   System.out.println("------------------------------");
   ClasseBase cb;
   cb = new ClasseBase();
   System.out.println("------------------------------");
  }
 }
 class ClasseBase {
  static  int vclasseBase = 1;
  int vinstanceBase = 10;
  static {
   afficheVclasse();
   vclasseBase = 2;
  }
  {
   System.out.println("vinstanceBase = " + vinstanceBase);
   vinstanceBase = 11;
  }
  public ClasseBase(){
   afficheVclasse();
   System.out.println("vinstanceBase = " + vinstanceBase);
   vinstanceBase ++;
   vclasseBase ++;
   afficheVclasse();
   System.out.println("vinstanceBase = " + vinstanceBase);
  }
  public static void afficheVclasse() {
   System.out.println("vclasseBase = " + vclasseBase);
  }
 }
 class ClasseDerivee extends ClasseBase {
  static private int vclasseDerivee = 31;
  private int vinstanceDerivee = 20;
  static {
   afficheVclasse();
   vclasseDerivee = 32;
  }
  {
   System.out.println("vinstanceDerivee = " + vinstanceDerivee);
   vinstanceDerivee = 21;
  }
  public ClasseDerivee(){
   super();
   afficheVclasse();
   System.out.println("vinstanceDerivee = " + vinstanceDerivee);
   vinstanceDerivee ++;
   vclasseDerivee ++;
   afficheVclasse();
   System.out.println("vinstanceDerivee = " + vinstanceDerivee);
  }
```

```
 public static void afficheVclasse() {
  System.out.println("vclasseDerivee = " + vclasseDerivee);
 }
}
```

Résultat de l'exécution :

```
------------------------------
vclasseBase = 1
vclasseDerivee = 31
vinstanceBase = 10
vclasseBase = 2
vinstanceBase = 11
vclasseBase = 3
vinstanceBase = 12
vinstanceDerivee = 20
vclasseDerivee = 32
vinstanceDerivee = 21
vclasseDerivee = 33
vinstanceDerivee = 22
------------------------------
vinstanceBase = 10
vclasseBase = 3
vinstanceBase = 11
vclasseBase = 4
vinstanceBase = 12
vinstanceDerivee = 20
vclasseDerivee = 33
vinstanceDerivee = 21
vclasseDerivee = 34
vinstanceDerivee = 22
------------------------------
vinstanceBase = 10
vclasseBase = 4
vinstanceBase = 11
vclasseBase = 5
vinstanceBase = 12
------------------------------
```

6. Classe Objet

6.1 Héritage implicite et super classe

La classe `Object` fait partie du package `java.lang` qui est automatiquement importé dans tout programme Java. C'est la « **super classe** » de toutes les classes Java. Toute classe Java hérite implicitement directement ou indirectement de cette classe. Lorsque l'on définit une classe de base qui n'hérite d'aucune classe, celle-ci hérite de la classe `Object`. Les deux déclarations suivantes sont parfaitement équivalentes.

héritage implicite	**héritage** explicite

`[public] class ClasseBase {`
`  …`
`}`
	`[public] class ClasseBase extends Object {`
	`  …`

héritage implicite	**héritage** explicite
	}

Tableau 17 : héritage implicite et explicite de la classe Object

Ainsi toutes les classes qui héritent d'une classe de base vont aussi hériter de la classe `Objet`. La classe Object constitue la racine de la hiérarchie des classes : toute nouvelle classe en constitue donc une classe dérivée (par transitivité).

La classe `Object` ne possède aucun champ, mais regroupe quelques méthodes non statiques, nous allons en décrire quelques-unes :

- son constructeur `public Object()` est unique et n'a pas de paramètre
- le comparateur `public boolean equals(Object)` qui retourne `true` si l'objet dont la référence est passée a la même référence que l'objet courant, `false` sinon. Une classe doit outrepasser cette méthode, en général, en comparant les champs.
- La méthode `public String toString()` qui retourne une chaîne de caractères résultant de la concaténation du nom de la classe, suivi du séparateur @, puis la référence de l'objet.
- La méthode de clonage `protected Object clone()` retourne une copie de l'objet courant. Normalement la référence renvoyée doit être distincte de celle de l'objet courant. Comme cette méthode est protégée et qu'elle fait partie du `java.lang`, elle ne peut pas être appelée dans un autre package, elle ne peut être que redéfinie. Son but est d'être outrepassée, sinon elle est inutilisable.

On peut remarquer que ces méthodes sont systématiquement redéfinies lorsque l'on adopte une écriture standard de classe (§6 p 92). Il existe d'autres méthodes utilisées dans le cadre du « multi threading » et de l'introspection que nous découvrirons ultérieurement.

Dans l'exemple qui suit, deux versions de la classe `Point` sont définies, la première la classe `Point` dans laquelle les méthodes précédentes sont outrepassées, la seconde `PointFaible` dans laquelle elle ne le sont pas, les résultats sont moins « satisfaisants ». En particulier, la comparaison de deux points de mêmes coordonnées `paf.equals(pbf)` donne un résultat faux, en effet, la méthode `equals` de `Object` utilisée ne fait que comparer les adresses. L'écriture `paf.clone()` provoquerait l'erreur de compilation : « `clone() has protected access in java.lang.Object` ».

Il est à noter que le comparateur `public boolean equals(Object)`n'est pas strictement redéfini, mais plutôt remplacé par `public boolean equals(Point p)`. Nous espérons que l'utilisateur de la classe comparera toujours des objets de type Point. Nous verrons dans la partie relative au polymorphisme comment aller encore plus loin (voir §7.6 p184).

Exemple :

```
public class Test {
 static public void main(String [] args) {
  Point pa = new Point(1, 1);
  Point pb = new Point(1, 1);
  System.out.println(pa);
  System.out.println(pb);
  System.out.println(pa.equals(pb));
```

```
   System.out.println(pa.equals(pa));
   System.out.println(pa.clone());
   PointFaible paf = new PointFaible(1, 1);
   PointFaible pbf = new PointFaible(1, 1);
   System.out.println(paf);
   System.out.println(pbf);
   System.out.println(paf.equals(pbf));
   System.out.println(paf.equals(paf));
  }
}
class Point {
 private int x, y;
 public  Point(int nx, int ny) {
    x = nx;
    y = ny;
 }
 public boolean equals(Point p) {
  if(p.x == x && p.y == y) return true;
  return false;
 }
 public String toString() {
  return "(" + x + ", " + y + ")";
 }
 public Point clone() {
  return new Point(x, y);
 }
}
class PointFaible {
 private int x, y;
 public  PointFaible(int nx, int ny) {
    x = nx;
    y = ny;
 }
}
```

Résultat de l'exécution :

```
(1, 1)
(1, 1)
true
true
(1, 1)
PointFaible@addbf1
PointFaible@42e816
false
true
```

6.2 Copie profonde

Lors d'un clonage, si on n'y prend pas gare, si un des champs recopiés est une référence sur un objet, seule la référence est recopiée et désignera le même objet. Donc, pour que les références clonées désignent des objets différents, il faut aussi cloner ces

objets, dans la méthode `clone`. Le problème est identique pour les constructeurs de recopie.

Dans l'exemple suivant, la classe `Boite` possède comme unique champ `p` la référence d'un objet de type `Point`. La méthode `cloneFaible` retourne un « mauvais clone » qui possède comme valeur du champ `p` la même que celle de l'objet de type `Boite` d'origine. Ainsi toute modification de l'objet `Point` référencée par l'un modifiera l'autre. La méthode `clone` de `Boite` effectue une copie en profondeur en utilisant la méthode `clone` de `Point`, ainsi, l'objet d'origine et son clone référencent des objets de type `Point` distincts (leur contenu pouvant être éventuellement identique). Le problème aurait pu aussi être contourné en modifiant le constructeur de `Point` :

```
public  Boite(Point np) {
     p = np.clone();
 }
```

Le constructeur de recopie de `Boite` pourrait avoir l'allure suivante :

```
public Boite(Boite b){
  new Boite(p.clone());
 }
```

Exemple :

```
public class Test {
 static public void main(String [] args) {
  Point p = new Point(1, 1);
  Boite b = new Boite(p);
  System.out.println(p);
  System.out.println(b);
  Boite ba = b.clone();
  Boite bb = b.cloneFaible();
  b.plus();
  System.out.println(b);
  System.out.println(ba);
  System.out.println(bb);
 }
}
class Point {
 private int x, y;
 public  Point(int nx, int ny) {
    x = nx;
    y = ny;
 }
 public boolean equals(Point p) {
  if(p.x == x && p.y == y) return true;
  return false;
 }
 public String toString() {
  return "(" + x + ", " + y + ")";
 }
 public Point clone() {
  return new Point(x, y);
 }
```

```
  public void plus() {
   x ++;
   y ++;
  }
}
class Boite {
 private Point p;
 public  Boite(Point np) {
    p = np;
 }
 public String toString() {
  return "Box : " + p;
 }
 public Boite clone() {
  return new Boite(p.clone());
 }
 public Boite cloneFaible() {
  return new Boite(p);
 }
 public void plus() {
  p.plus();
 }
}
```

Résultat de l'exécution :

```
(1, 1)
Box : (1, 1)
Box : (2, 2)
Box : (1, 1)
Box : (2, 2)
```

7. Polymorphisme

7.1 Compatibilité

7.1.1 Cas général

Une classe dérivée constitue un sous type de sa classe de base, un objet du type de la classe dérivée est aussi un objet du type de sa classe de base. Dès lors la référence d'un objet de la classe de base peut être utilisée en lieu et place de la référence d'un objet de la classe dérivée. Ceci n'est pas contradictoire, en effet, par définition de l'héritage, un objet de la classe de dérivée est aussi un objet de la classe de base. Ainsi, un objet peut posséder plus d'un type.

Si `classeDerivee` est une sous classe de `classeBase`, une variable de type référence d'un objet de type `classeBase` peut référencer un objet de type `classeDerivee`, on dira que cette objet est surclassé. De même si on affecte à une variable `vbase` de type référence d'un objet de type `classeBase` une variable `vderivee` de type référence sur un objet de type `classeDerivee`, `vderivee` est surclassée.

Cette opération de **surclassement** (*upcasting*) peut se faire sans conversion, il s'agit de surclassement implicite, ici la variable `cd` est surclassée, lors de l'affectation `cb = cd` :

```
class ClasseBase {
```

```
  …
}
class ClasseDerivee extends ClasseBase {
  …
}
…
ClasseDerivee cd = new ClasseDerivee(…);
ClasseBase cb = cd;
```

En résumé, toute instance d'une classe dérivée est implicitement compatible avec une classe hiérarchiquement supérieure :

- Toute instance de classe est compatible avec la classe Objet
- Cette caractéristique peut être appliquée aux tableaux d'objets compatibles (hors types primitifs)
- Le surclassement implicite et explicite (avec une conversion) est permis

L'inverse est possible et nécessite l'utilisation d'une conversion de type, mais à l'exécution la variable affectée doit être de type compatible, c'est-à-dire dire que (la référence) de l'objet à affecter doit correspondre au type (classe) de la variable à affecter ou à un sous type de la variable (classe dérivée directement ou indirectement). Le **sousclassement** (*downcasting*) explicite est permis, mais il nécessite une conversion. Le sousclassement implicite est interdit. Ici la variable `cb` est sousclassée, lors de l'affectation `cd = cb` :

```
ClasseBase cb = new ClasseBase (…);
ClasseDerivee cd = cb;
```

Ceci est interdit en Java et provoque l'erreur de compilation « `incompatible types` ». Par contre, l'écriture suivante est permise, il s'agit de sousclassement explicite :

```
ClasseBase cb = new ClasseBase(…);
ClasseDerivee cd = (ClasseDerivee)cb;
```

Mais bien évidemment, cette écriture bien qu'acceptée par le compilateur n'a toujours pas de sens, en effet, `cd` référence un objet du type de la classe de base (`ClasseBase`), cela n'a pas de sens. L'erreur se produira à l'exécution : « `… java.lang.ClassCastException: ClasseBase cannot be cast to ClasseDerivee at …` ».

Bien évidemment, Java nous laisse cette possibilité si `cb` référence non pas un objet du type de la classe `ClasseBase`, mais à un objet du type de la classe `ClasseDerivee` (ou une classe dérivée de `ClasseDerivee`), dans ce cas, la compatibilité est assurée.

```
ClasseBase cb = new ClasseDerivee();
ClasseDerivee cd = (ClasseDerivee)cb;
```

Java laisse cette possibilité, car le type effectif de l'objet référencé ne peut être connu qu'à l'exécution. Dans le cas précédent, `cb` aurait aussi bien pu référencer un objet de type `ClasseBase` qu'un objet de type `ClasseDerivee`. Comme il y a risque d'incompatibilité, Java impose tout de même une conversion (c'est au programmeur d'en prendre la responsabilité). Java détermine le type effectif d'un objet au moment de l'exécution.

En résumé, une référence sur un type d'objet de classe X ne peut référencer qu'un objet de classe X ou d'une classe dérivée de la classe X (directement ou indirectement).

Le programme suivant illustre les possibilités en matière de compatibilité de type, de surclassement et sousclassement. Les classes `PointCouleur` et `PointNoirEtBlanc` héritent de la classe `Point`, donc indirectement de la classe `Object`, on en déduit que :

- une variable de type référence (d'un objet) de la classe `Objet` peut référencer un objet de type `Object`, `Point`, `PointCouleur` ou `PointNoirEtBlanc`
- une variable de type référence de la classe `Point` peut référencer un objet de type `Point`, `PointCouleur` ou `PointNoirEtBlanc`
- une variable de type référence de la classe `PointCouleur` peut uniquement référencer un objet de la classe `PointCouleur`
- une variable de type référence de la classe `PointNoirEtBlanc` peut uniquement référencer un objet de la classe `PointNoirEtBlanc`

Dans l'exemple suivant, considérons l'affectation `PointNoirEtBlanc pnb = (PointNoirEtBlanc)o`, il s'agit d'un sousclassement explicite car la variable `o` est de type référence (d'un objet de type) `Object`, la conversion est nécessaire. Comme variable `o` référence un objet de type `PointNoirEtBlanc`, alors l'affectation est licite et ne provoquera pas d'erreur d'exécution. L'affectation `p=pnb` constitue un surclassement implicite d'une variable référence de type `PointNoirEtBlanc` à une référence de type `Point` qui ne pose pas de problème. L'affectation `o = (Object)p` constitue un surclassement explicite d'une référence de type `Point` à une référence de type `Object`, ce surclassement ne pose pas non plus de problème, du reste, la conversion est optionnelle.

La Figure 22 représente les comptabilités entre les références et les objets concernant uniquement les trois classes `Point`, `PointNoirEtBlanc` et `PointCouleur`.

La référence d'un `PointCouleur` et d'un `PointNoirEtBlanc` sont incompatibles (aucune relation hiérarchique), ce qui provoque l'erreur de compilation : « `inconvertible types` ».

```
PointCouleur pc = new PointCouleur(1,2,3);
PointNoirEtBlanc pnb = (PointNoirEtBlanc)pc;
```

Dans ce cas, lors de l'affectation, une tentative de sousclassement implicite est effectué, ce qui provoque l'erreur de compilation : « `inconvertible types` ». Comme `p` référence un objet de type `PointCouleur`, il n'y aurait pas eu de problème en effectuant une conversion.

```
Point p = new PointCouleur(3,4,5);
pc = p;
```

Dans ce dernier cas, lors de l'affectation, une tentative de sousclassement explicite est effectuée, celle-ci ne pose pas de problème de compilation. Mais cela provoque une erreur d'exécution : « `java.lang.ClassCastException: Point cannot be cast to PointCouleur at` ». En effet, `pa` référence un `Point`, or la référence d'un `Point` ne peut pas être affectée à la référence d'un `PointCouleur`.

```
Point pa = new Point(6, 7);
pc = (PointCouleur) pa;
```

Exemple :

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

```
   Object o = new Object();
   o = new Point(2,1);
   o = new PointCouleur(5,4,3);
   o = new PointNoirEtBlanc(7,6,false);
   Point p;
   p = new Point(1,2);
   p = new PointCouleur(3,4,5);
   p = new PointNoirEtBlanc(6,7,true);
   PointCouleur pc = new PointCouleur(8,9,10);
   PointNoirEtBlanc pnb = (PointNoirEtBlanc)o;
   p = pnb;
   o = (Object)p;
 }
}
class Point {
 private int x, y;
 public  Point(int nx, int ny) {
    x = nx;
    y = ny;
 }
}
class PointCouleur extends Point{
 private  int c;
 public  PointCouleur(int nx, int ny, int nc) {
  super(nx, ny);
  c = nc;
 }
}
class PointNoirEtBlanc extends Point{
 private  boolean c;
 public  PointNoirEtBlanc(int nx, int ny, boolean nc) {
  super(nx, ny);
  c = nc;
 }
}
```

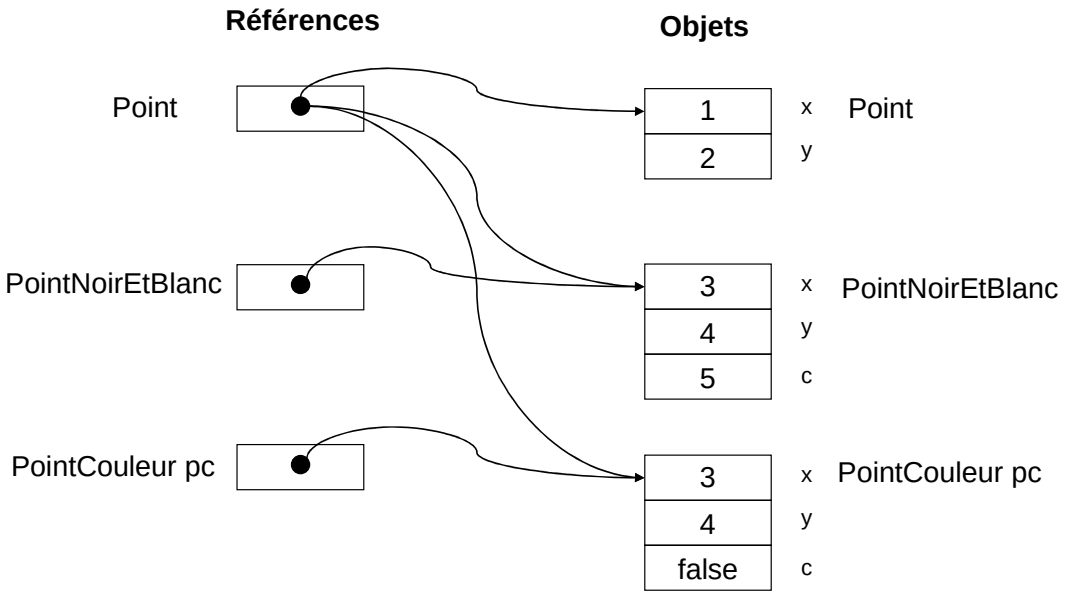


Figure 22 : Compatibilité référence objet

Dans le cas des tableaux, cette compatibilité s'exerce de façon similaire. Une variable de type référence d'un tableau d'une classe donnée peut référencer un tableau dont les éléments sont des références d'objets d'une classe dérivée de cette classe donnée. Chaque élément du tableau peut à son tour référencer un objet de type classe dérivée de cette classe dérivée (on se trouve exactement dans le cas de figure précédent).

7.1.2 Cas des tableaux

Dans l'exemple suivant `to` est une variable de type référence d'un tableau d'objets de type `Object`. Ce tableau référence un tableau d'objets de type `Point`, comme `Point` est dérivé `d'Object`, la compatibilité est assurée. Ensuite, `to[0]`, de type référence d'un objet de type `Point`, référence un objet de type `PointCouleur`, la compatibilité est encore assurée, `PointCouleur` étant une classe dérivée de `Point`.

Par contre, si on avait écrit :

```
to[1] = new String("toto");
```

Il n'y aurait pas eu d'erreur de compilation `String` étant une sous classe de `Object`, mais une erreur d'exécution de type « `Exception in thread "main" java.lang.ArrayStoreException: java.lang.String at` ». En effet, comme il s'agit de typage dynamique, le compilateur n'a pas les moyens de vérifier qu'il y a compatibilité entre `to[1]` et le type de ce qui est vraiment référencé (en effet, `to` aurait très bien pu référencer une tableau de chaînes de caractères, en écrivant, `to = new String[10];` et dans ce cas, il n'y aurait pas de problème) ; dans ce second cas, c'est un objet dynamique, dont le type sera défini à l'exécution, qui référence un autre objet dynamique.

Exemple :

```
public class Test {
 static public void main(String [] args) {
  Object []to;
  to = new Point[10];
  to[0] = new PointCouleur(1,2,3);
 }
```

```
}
class Point {
 private int x, y;
 public  Point(int nx, int ny) {
    x = nx;
    y = ny;
 }
}
class PointCouleur extends Point{
 private  int c;
 public  PointCouleur(int nx, int ny, int nc) {
  super(nx, ny);
  c = nc;
 }
}
```

7.2 Polymorphisme universel d'inclusion

Le mot **polymorphisme** est formé à partir du grec ancien et signifie « plusieurs formes ». En proposant d'utiliser un même nom de méthode pour plusieurs types d'objets différents, le polymorphisme permet une programmation beaucoup plus générique.

Nous avons vu que l'appel d'une « méthode polymorphe » permet de distinguer la bonne méthode par rapport à un autre :

- Soit à l'aide des types des arguments, ce qui est sous jacent au mécanisme de surcharge
- Soit à l'aide du type d'objet auquel cette méthode est appliqué et des types des arguments, ce qui est sous jacent aux mécanismes d'outrepassement, si les types des paramètres correspondent dans la classe de base et ses classes dérivées, et sinon, de surcharge

Mais, il est possible d'aller plus loin, l'appel d'une méthode spécifique ne dépend pas du type associé à la référence (définie à la déclaration) à laquelle la méthode est associée, mais de la classe effective de l'objet référencé. Le choix de la bonne méthode n'est pas fait lors de la compilation, mais lors de l'exécution. Ce mécanisme est qualifié de liaison dynamique ou liaison retardée (dynamic binding ou late binding).

La condition nécessaire pour qu'un langage implémente la liaison dynamique c'est qu'il existe un mécanisme pour déterminer le type de l'objet lors de l'exécution afin d'appeler ainsi la méthode appropriée. Le compilateur ne connaît toujours pas le type de l'objet, mais le mécanisme d'appel de méthode trouve et effectue l'appel vers la bonne méthode. L'association entre la liaison tardive et la détermination dynamique du type va permettre de réaliser un type de polymorphisme universel[26], le polymorphisme d'inclusion (aussi appelé polymorphisme d'héritage car l'inclusion prend la forme d'un héritage). Java permet donc d'appliquer ce mécanisme entre des classes faisant partie d'une même hiérarchie, en se basant sur la compatibilité des classes et l'outrepassement des méthodes.

[26] L'autre type de polymorphisme universel est le polymorphisme paramétrique basé sur les types génériques en java (et les templates en C++)

Dans l'exemple qui suit, les classes `PointCouleur` et `PointNoirEtBlanc` héritent de la classe `Point`, chacune des classes dérivée possèdent une méthode `toString` qui outrepasse celle de la classe de base `Point`. Lors des trois premiers appels, il y a coïncidence entre le type de la référence et le type de l'objet référencé (bien qu'inutile ici, la liaison dynamique est tout de même utilisée), il n'y a aucune ambiguïté. Par contre, dans les deux derniers cas, le type de la variable est distinct de celui de l'objet référencé (qui est surclassé), dans ce cas, on tire donc partie du mécanisme de liaison dynamique, lors du dernier appel, c'est bien la méthode `toString` de `PointNoirEtBlanc` qui est appelée et lors de l'avant dernier appel c'est celle de `PointCouleur`.

Exemple :

```
public class Test {
 static public void main(String [] args) {
  Point p = new Point(1,2);
  PointCouleur pc = new PointCouleur(3,4,5);
  PointNoirEtBlanc pnb = new PointNoirEtBlanc(6,7,true);
  System.out.println(p.toString());
  System.out.println(pc.toString());
  System.out.println(pnb.toString());
  p = pc;
  System.out.println(p.toString());
  p = pnb;
  System.out.println(p.toString()); }
}
class Point {
 private int x, y;
 public  Point(int nx, int ny) {
    x = nx;
    y = ny;
 }
 public String toString() {
  return "Point->(" + x + ", " + y + ")";
 }
}
class PointCouleur extends Point{
 private  int c;
 public  PointCouleur(int nx, int ny, int nc) {
  super(nx, ny);
  c = nc;
 }
 public String toString() {
  return super.toString() + " Couleur : " +c;
 }
}
class PointNoirEtBlanc extends Point{
 private  boolean c;
 public  PointNoirEtBlanc(int nx, int ny, boolean nc) {
  super(nx, ny);
  c = nc;
 }
```

```
  public String toString() {
   return super.toString() + " NoirEtBlanc : " +c;
  }
 }
```

Résultat de l'exécution :

```
Point->(1, 2)
Point->(3, 4) Couleur : 5
Point->(6, 7) NoirEtBlanc : true
Point->(3, 4) Couleur : 5
Point->(6, 7) NoirEtBlanc : true
```

7.3 Contrôle statique

Lorsque l'on utilise la référence d'un objet d'une classe de base pour lui affecter la référence d'un objet d'une de ses classes dérivées et que l'on appelle une méthode, il faut obligatoirement que cette méthode soit aussi définie dans la classe de base. Ce mécanisme permet au compilateur d'effectuer un contrôle statique des types d'arguments. Ainsi, il s'assure qu'il existe une méthode qui pourra toujours être appelée et éviter une erreur en cours d'exécution. Si, la classe dérivée n'outrepasse pas la méthode, c'est la méthode de la classe de base qui sera appelée. Ceci correspond au principe de substitution : une classe dérivée doit pouvoir être utilisée dans toutes les situations où ses classes de base peuvent l'être, sans qu'il puisse être possible de percevoir une différence.

Dans l'exemple suivant, bien que la variable `p` référence un objet de type `PointCouleur`, le programme ne se compile pas car la classe `Point` ne possède pas de méthode `transparent`. Ceci provoque l'erreur de compilation : « `cannot find symbol method transparent()` ».

```
public class Test {
 static public void main(String [] args) {
  Point p = new PointCouleur(3,4,5);
  System.out.println(p.transparent());
 }
}
class Point {
 private int x, y;
 public  Point(int nx, int ny) {
    x = nx;
    y = ny;
 }
}
class PointCouleur extends Point{
 private  int c;
 public  PointCouleur(int nx, int ny, int nc) {
  super(nx, ny);
  c = nc;
 }
 public boolean transparent() {
  return c == 0;
 }
}
```

7.4 Surcharge et hiérarchie de classes

Nous avions vu, avant même d'aborder les aspects objets du langage Java, que lors d'un appel de méthode, lorsque les types de paramètre ne correspondent pas exactement à ceux des arguments, si une méthode compatible peut convenir (tous les paramètres sont hiérarchiquement supérieurs ou égaux à ceux des arguments dans la hiérarchie des types primitifs), elle est choisie (s'il y en à plusieurs la meilleure est choisie), c'est la base du mécanisme de surcharge.

Le langage Java généralise cela aux hiérarchies de types sous jacentes à l'héritage. Il correspond à la possibilité d'invoquer une méthode définie pour un paramètre de type (classe) X avec un paramètre de type (classe) Y défini comme un sous-type (classe dérivée) du type X. Lorsque plusieurs méthodes sont candidates, il choisit la meilleure (type de paramètres les plus proches en remontant la hiérarchie de classes). C'est du polymorphisme d'inclusion (d'h'ériatge).

Dans l'exemple qui suit, la classes `PointCouleur` hérite de la classe Point qui elle-même hérite implicitement de la classe `Object`. Donc dans la hiérarchie de classes (types), on a `Object` > `Point` > `PointCouleur`.

Dans main, plusieurs méthodes, n'ayant qu'un unique paramètre, sont définies :

- `montre` qui est surchargée dans les trois classes `Object`, `Point` et `PointCouleur`
- `montrebis` qui est surchargée dans les deux classes `Object` et `Point`
- `montrebisbis` qui n'existe que dans le type `Object`

Lors des appels de `montre`, il y a coïncidence exacte entre le type de la référence passé et le type du paramètre.

Lors de l'appel de `montreBis` avec une référence de `PointCouleur`, comme il n'y a pas de méthode correspondant exactement, le compilateur cherche la meilleure version compatible, c'est donc `montrebis(Point)` qui est appelée, meilleure candidate que `montrebis(Object)`.

Lors de l'appel de `montrebisbis` avec une référence de `PointCouleur` ou de `Point`, comme il n'y a pas de méthode correspondant exactement, le compilateur cherche la meilleure version compatible, il n'y en a qu'une, c'est donc `montrebisbis(Object)` qui est appelée,

Exemple :

```
public class Test {
 static public void main(String [] args) {
  Object o = new Object();
  Point p = new Point(1,1);
  PointCouleur pc = new PointCouleur(1,2,3);
  montre(o);
  montre(p);
  montre(pc);
  montrebis(p);
  montrebis(pc);
  montrebisbis(o);
  montrebisbis(p);
  montrebisbis(pc);
 }
```

```
 static void montre(Object a) {
  System.out.println("Object : " + a);
 }
 static void montre(Point a) {
  System.out.println("Point : " + a);
 }
 static void montre(PointCouleur a) {
  System.out.println("PointCouleur : " + a);
 }
 static void montrebis(Point a) {
  System.out.println("Point : " + a);
 }
 static void montrebisbis(Object a) {
  System.out.println("Object : " + a);
 }
}
class Point {
 private int x, y;
 public  Point(int nx, int ny) {
    x = nx;
    y = ny;
 }
 public String toString() {
  return "(" + x + ", " + y + ")";
 }
}
class PointCouleur extends Point{
 private  int c;
 public  PointCouleur(int nx, int ny, int nc) {
  super(nx, ny);
  c = nc;
 }
 public String toString() {
  return super.toString() + ":" +c;
 }
}
```

Résultat de l'exécution :

```
Object : java.lang.Object@addbf1
Point : (1, 1)
PointCouleur : (1, 2):0
Point : (1, 1)
Point : (1, 2):0
Object : java.lang.Object@addbf1
Object : (1, 1)
Object : (1, 2):0
```

7.5 Limites du polymorphisme en Java

7.5.1 Types de paramètres

Dans le cas du polymorphisme d'inclusion de Java, le type effectif de l'objet auquel la méthode est appliquée et le type des arguments qui lui sont passés sont exploités. Dans le cas où les arguments sont des objets, c'est le type de la variable qui est utilisé et non pas le type de l'objet référencé. Le type de l'objet auquel la méthode est appliquée est déterminé à l'exécution et celui des arguments à la compilation. On peut regretter que les concepteurs de Java n'ait fait le choix inverse.

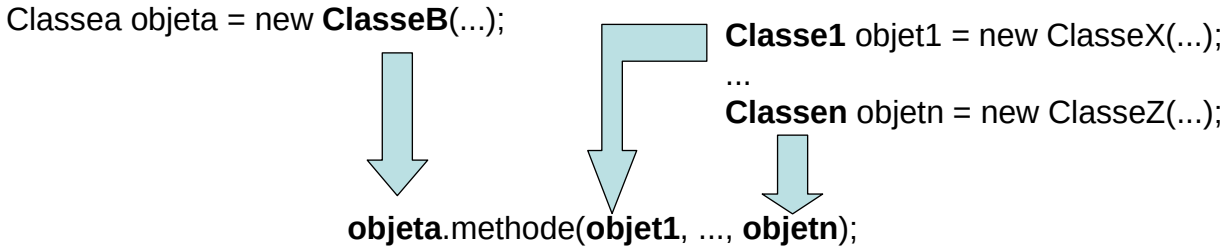


Figure 23 : Types des objets utilisés (méthode non statique)

Dans l'exemple qui suit, la classe `PointCouleur` hérite de la classe `Point`. Chacune de ces deux classes possède une version de la méthode `coincide` avec comme type de paramètre la référence d'un `Point` et la référence d'un `PointCouleur`. Des affichages sont ajoutés afin d'améliorer la compréhension. Etudions les différents appels (c'est ici le second qui est le plus pertinent) :

- `pb.coincide(pa);` `pb` référence un `PointCouleur`, le type de l'argument est (une référence de) `Point`, c'est donc la méthode `coincide(Point)` de `PointCouleur` qui est appelée
- `pa.coincide(pb);` `pa` référence un `Point`, le type de l'argument est (une référence de) `Point`, bien que l'objet référencé par l'argument soit de type `PointCouleur`, c'est donc la méthode `coincide(Point)` de `Point` qui est appelée
- `pa.coincide(pb);` `pa` référence un `Point`, le type de l'argument est (une référence de) `Point` qui est explicitement convertie en `PointCouleur`, c'est donc la méthode `coincide(PointCouleur)` de `Point` qui est appelée
- `pa.coincide(new Point(1,1));` pa référence un `Point`, le type de l'argument est (une référence de) `Point` créé dynamiquement, c'est donc la méthode `coincide(Point)` de `Point` qui est appelée

Exemple :

```
public class Test {
 static public void main(String [] args) {
  Point pa = new Point(1, 1);
  Point pb = new PointCouleur(1, 1, 2);
  pb.coincide(pa);
  pa.coincide(pb);
  pa.coincide((PointCouleur)pb);
  pa.coincide(new Point(1,1));
 }
}
```

```
class Point {
 private int x, y;
 public  Point(int nx, int ny) {
    x = nx;
    y = ny;
 }
 public String toString() {
  return "(" + x + ", " + y + ")";
 }
 public  boolean coincide(Point p) {
  System.out.println("Point -> coincide(Point p)");
  if (x == p.x && y == p.y) return true;
  return false;
 }
 public  boolean coincide(PointCouleur p) {
  System.out.println("Point -> coincide(PointCouleur p)");
  if (x == p.getX() && y == p.getY()) return true;
  return false;
 }
 public int getX() {
  return x;
 }
 public int getY() {
  return y;
 }
}
class PointCouleur extends Point{
 private  int c;
 public  PointCouleur(int nx, int ny, int nc) {
  super(nx, ny);
  c = nc;
 }
 public String toString() {
  return super.toString() + ":" +c;
 }
 public  boolean coincide(Point p) {
  System.out.println("PointCouleur -> coincide(Point p)");
  if (p.getX() == getX() && p.getY() == getY()) return true;
  return false;
 }
 public  boolean coincide(PointCouleur p) {
  System.out.println("PointCouleur -> coincide(PointCouleur p)");
  if (p.getX() == getX() && p.getY() == getY()) return true;
  return false;
 }
}
```

Résultat de l'exécution :

```
PointCouleur -> coincide(Point p)
Point -> coincide(Point p)
Point -> coincide(PointCouleur p)
```

```
Point -> coincide(Point p)
```

7.5.2 Méthode statique

Les méthodes statiques (`static`) appartiennent à la classe dans laquelle ils sont définis. L'appel d'une méthode est déterminé par le nom de la classe à laquelle elle est associée ou par la classe de la variable à laquelle elle est associée. Ce n'est pas la classe de l'objet référencé qui est utilisée, comme dans le cas de l'outrepassement de méthodes non statiques. C'est grâce à la déclaration de la référence que le compilateur détermine bon appel, à l'inverse du polymorphisme d'inclusion. Dans ce cas, il ne s'agit pas vraiment d'une limite du polymorphisme.

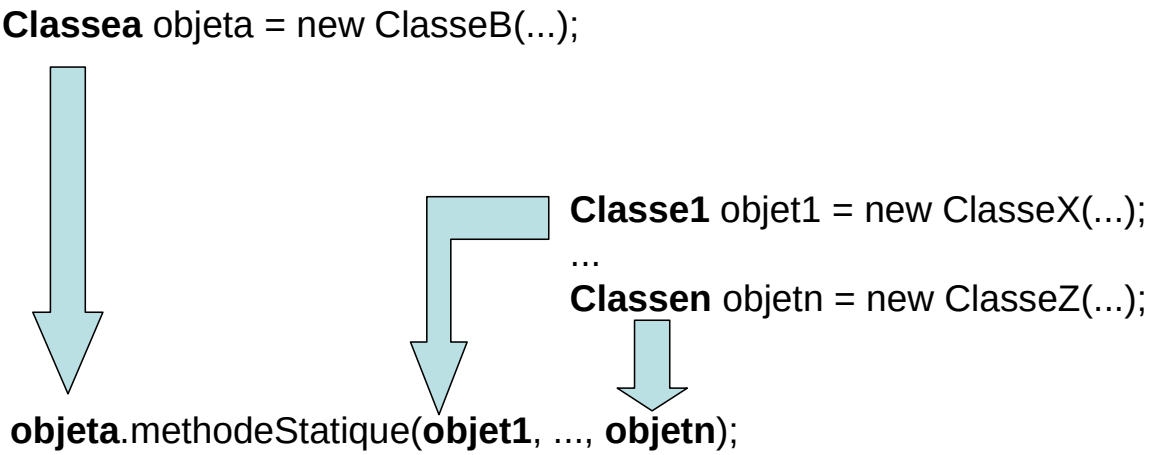


Figure 24 : Types des objets utilisés (méthode statique)

Dans l'exemple qui suit, la classe `PointCouleur` hérite de la classe `Point`. Chacune de ces deux classes possèdent à la fois un champ statique compteur dont la valeur correspond au nombre d'instances créées et une méthode statique qui retourne la valeur de ce champ. Lorsque l'on appelle la méthode `printCompteur` avec la variable `p` de type (référence de) `Point` qui référence un objet de type `PointCouleur` (`p.printCompteur();`), c'est bien la méthode `printCompteur` de la classe `Point` qui est appelée, comme dans le cas d'une association directe avec le nom de la classe `Point` (`Point.printCompteur();`).

Exemple :

```
public class Test {
 static public void main(String [] args) {
  int i;
  Point []tp = new Point[5];
  Point []tpb = new Point[5];
  PointCouleur []tpc = new PointCouleur[10];
  Point.printCompteur();
  PointCouleur.printCompteur();
  for(i = 0; i < tpc.length; i++) {
   tpc[i] = new PointCouleur(i, i+1, i+2);
  }
  for(i = 0; i < tp.length; i++) {
   tp[i] = new PointCouleur(2*i, 2*i, 2*i);
  }
  for(i = 0; i < tpb.length; i++) {
```

```
    tpb[i] = new Point(3*i, 3*i+1);
   }
   Point.printCompteur();
   PointCouleur.printCompteur();
   Point p = new Point(0,0);
   p.printCompteur();
   PointCouleur pc = new PointCouleur(1,1,1);
   pc.printCompteur();
  }
}
class Point {
 private int x, y;
 private static int compteur = 0;
 public  Point(int nx, int ny) {
    x = nx;
    y = ny;
    compteur ++;
 }
 public static void printCompteur() {
   System.out.println("Point : "+ compteur);
 }
}
class PointCouleur extends Point{
 private int  c;
 private static int compteur = 0;
 public  PointCouleur(int nx, int ny, int nc) {
  super(nx, ny);
    c = nc;
    compteur ++;
 }
 public static void printCompteur() {
   System.out.println("PointCouleur : "+ compteur);
 }
}
```

Résultat de l'exécution :

```
Point : 0
PointCouleur : 0
Point : 20
PointCouleur : 15
Point : 21
PointCouleur : 16
```

7.6 Opérateur instanceof

L'opérateur `instanceof` est un opérateur de comparaison dont le membre de gauche est un objet et celui de droite une classe[27] qui retourne `true` si l'objet est une instance de la classe ou de l'une de ses classes dérivées, `false` sinon. La comparaison

[27] Ou une interface

s'effectue à l'exécution, cependant le compilateur vérifie que les types sont compatibles (le membre de gauche doit être faire partie de la classe de droite ou en dériver), s'ils ne le sont pas, cela produit l'erreur de compilation « `inconvertible types` ».

Il est important d'utiliser `instanceof` avant une conversion descendant lorsque vous n'avez pas d'autres informations vous indiquant le type de l'objet. Ce type de précaution permet d'éviter des erreurs d'exécution liées aux conversions dynamiques.

Dans l'exemple qui suit, les classes `PointCouleur` et `PointNoirEtBlanc` héritent de la classe `Point`. La classe Point et chacune des classes dérivées possèdent une méthode `equals` qui outrepasse celle de la classe `Object` (classe dont `Point` hérite implicitement).

Dans les versions précédentes de la méthode `equals` que nous avions définies dans ces mêmes classes, il s'agissait de surcharge, chaque version de `equals` ayant comme type d'argument le même que celui de la classe dans laquelle elle était définie. Désormais, il est possible de comparer n'importe quel objet de cette hiérarchie de la classe avec n'importe quel type d'objet. Analysons le code de la méthode `equals` de `Point` :

`if(!(o instanceof Point))` permet de vérifier que l'objet reçu est bien de type `Point`, si l'objet n'est pas de la bonne classe, `false` est retourné.

`Point p = (Point) o;` la référence de l'objet reçu est convertie en `Point`, si le test précédent n'avait pas été effectué, l'appel de la méthode avec la référence d'un objet de type XXX incompatible avec `Point` aurait provoqué une erreur d'exécution de type « `java.lang.ClassCastException: XXX cannot be cast to Point` ».

`if (p.x == x && p.y == y)`, comme dans les versions précédentes, l'égalité des coordonnées est testée.

On peut remarquer que ce type d'écriture offre beaucoup de souplesse.

Exemple :

```
public class Test {
 static public void main(String [] args) {
  Point p = new Point(1,1);
  PointCouleur pc = new PointCouleur(1,2, 9);
  PointCouleur pc2 = new PointCouleur(1,2, 9);
  PointNoirEtBlanc pnb = new PointNoirEtBlanc(1,3, true);
  System.out.println(pc instanceof Object);
  System.out.println(pc instanceof PointCouleur);
  System.out.println(pc instanceof Point);
  System.out.println(p instanceof Point);
  System.out.println(p instanceof PointCouleur);
 //System.out.println(pnb instanceof PointCouleur);
  System.out.println(pc.equals(pc2));
  System.out.println(pc.equals(p));
  System.out.println(p.equals(pc));
  System.out.println(pnb.equals(pc));
  System.out.println(((Point)pnb).equals(pc));
 }
}
class Point {
 private int x, y;
```

```
 public  Point(int nx, int ny) {
    x = nx;
    y = ny;
 }
 public boolean equals(Object o) {
  if(!(o instanceof Point)) return false;
  Point p = (Point) o;
  if(p.x == x && p.y == y) return true;
  return false;
 }
}
class PointCouleur extends Point{
 private int  c;
 public  PointCouleur(int nx, int ny, int nc) {
  super(nx, ny);
    c = nc;
 }
 public boolean equals(Object o) {
  if(!(o instanceof PointCouleur)) return false;
  PointCouleur p = (PointCouleur) o;
  if(super.equals((Point)p) && p.c == c) return true;
  return false;
 }
}
class PointNoirEtBlanc extends Point{
 private  boolean c;
 public  PointNoirEtBlanc(int nx, int ny, boolean nc) {
  super(nx, ny);
  c = nc;
 }
 public boolean equals(Object o) {
  if(!(o instanceof PointNoirEtBlanc)) return false;
  PointNoirEtBlanc p = (PointNoirEtBlanc) o;
  if(super.equals((Point)p) && p.c == c) return true;
  return false;
 }
}
```

Résultat de l'exécution :

```
true
true
true
true
false
true
false
false
false
false
```

7.7 Classes et méthodes final

Le mot clé Java `final` spécifie le caractère immuable de quelque chose. Précédemment, nous avons utilisé ce mot clé associé à des variables dont la valeur, une fois initialisée, ne peut plus être modifiée ; ces variables sont qualifiées de constantes.

Les méthodes peuvent aussi être qualifiées de `final` dans une classe de base. Il faut faire précéder sa définition par le mot-clé `final`. Dans ce cas, la méthode ne peut pas être redéfinie dans les classes dérivées, directement ou indirectement. Ainsi, on s'assure que le comportement de cette méthode est préservé durant l'héritage et ne peut pas être redéfini, mais la méthode peut être surchargée. Pour les méthodes `final`, il n'y à pas choisir la bonne méthode à l'exécution, il n'y a donc pas de ligature dynamique, l'exécution du code est généralement plus efficace. Une méthode privée est implicitement `final`.

Dans l'exemple qui suit, la classe `PointCouleur` hérite de la classe `Point`. La méthode `testCoincidence` de la classe `Point` est déclarée `final`. Elle ne peut pas être pas être redéfinie dans la classe `PointCouleur`, ce qui provoquerait l'erreur de compilation : « `testCoincidence(int,int) in PointCouleur cannot override testCoincidence(int,int) in Point; overridden method is final` ». Cependant, elle peut être surchargée dans `PointCouleur`. Comme elle est publique, elle peut être appelée dans `PointCouleur` ou n'importe où ailleurs.

Exemple :

```
public class Test {
 static public void main(String [] args) {
  Point p = new Point(1,1);
  PointCouleur pc = new PointCouleur(1,1, 9);
  System.out.println(p.testCoincidence(2,2));
  System.out.println(pc.testCoincidence(1,1,9));
 }
}
class Point {
 private int x, y;

 public  Point(int nx, int ny) {
    x = nx;
    y = ny;
 }
 final public boolean testCoincidence(int nx, int ny) {
  return x == nx && y == ny;
 }
}
class PointCouleur extends Point{
 private int  c;
 public  PointCouleur(int nx, int ny, int nc) {
  super(nx, ny);
    c = nc;
 }
 public boolean testCoincidence(int nx, int ny, int nc) {
  return testCoincidence(nx, ny) && c == nc;
```

```
 }
}
```

Résultat de l'exécution :

```
false
true
```

Quand on qualifie une classe de `final`, en faisant précéder sa définition par le mot-clé `final`, on spécifie qu'aucune classe ne peut en hériter. Cela est généralement motivé par des questions de sécurité ou d'efficacité du code (pas de ligature dynamique). Dans tous les cas, il faut vérifier la pertinence de ce type de choix qui limite les possibilités de réutilisation. Dans l'extrait de code qui suit, la tentative d'héritage conduit à l'erreur de compilation : « `cannot inherit from final ClasseXXX` ».

```
final class ClasseXXX {
 …
}
class ClasseYYY extends ClasseXXX {
…
}
```

8. Classes abstraites et interfaces

8.1 Classes abstraites

En programmation orientée objet, une classe dérivée est plus spécialisée que sa classe de base. A l'inverse, les classes dont elles dérivent qui sont plus générales et abstraites. A tel point qu'en poussant ce mécanisme de généralisation, on peut imaginer des classes incomplètement définies qui ne possèdent pas d'instance, ces classes peuvent néanmoins être pertinentes pour partager les caractéristiques de leurs sous classes, sous classes qui peuvent posséder des instances. Une telle classe est qualifiée de classe abstraite (`abstract`). Par exemple, une classe Pièce qui possède des données et méthodes relatives à sa composition, mais dont les caractéristiques précises géométriques (volumes, surfaces, …) ne sont connues que pour ses classes dérivées dont la forme est précisée, la classe Voiture qui possède des données et des méthodes relatives au nombre de passagers, au poids, … mais dont les performances (vitesse, accélération, consommation, …) ne sont connues que pour les classes dérivées dont la motorisation est précisée … Ces classes incomplètement définies sont dites abstraites. Par contre, toute instance de pièce possède bien un volume et une surface, mais le calcul ne pourra se faire que dans une sous classe suffisamment bien définie comme une Pièce cylindrique ou parallélépipédique, … Ainsi la classe Pièce ne possède pas directement d'instance, mais elle en possède au travers de ses sous classes.

En Java, les classes abstraites sont des classes dont certaines méthodes non statiques ne sont pas définies. Ces méthodes non définies sont précédées du mot clé `abstract` et ne possèdent pas de corps (sinon cela provoque l'erreur de compilation : « `abstract methods cannot have a body` »). Ainsi, une méthode abstraite ne possède pas d'implémentation, c'est une simple déclaration. Bien que ce ne soit pas une obligation, les méthodes abstraites ont vocation à supporter du polymorphisme d'inclusion.

Une classe abstraite ne peut pas posséder d'instance (cela provoque l'erreur de compilation : « `XXX is abstract; cannot be instantiated` »). Dès qu'une classe

possède une méthode abstraite, elle devient abstraite et sa définition doit être précédée du mot clé `abstract` (l'inverse est possible, mais d'un intérêt restreint). Ce sont les classes dérivées qui doivent fournir l'implémentation en définissant cette méthode. Si toutes les méthodes ne sont pas redéfinies dans la classe dérivée d'une classe de base abstraite, la classe dérivée est aussi abstraite (elle doit être déclarée `abstract`).

```
abstract class classeXXX {
 …
 abstract int methodeZZZ(type1 nom1, ..., typek nomk);
 …
}
```

Une méthode ne peut pas être à la fois abstraite et privée, une méthode ne peut pas avoir vocation à être à la fois redéfinie et dissimulée (cela provoque l'erreur de compilation : « `illegal combination of modifiers: abstract and private` »).

Un constructeur ne peut pas être abstrait, il appartient qu'à sa classe et ne peut pas être redéfini (cela provoque l'erreur de compilation : « `modifier abstract not allowed here` »).

Dans l'exemple qui suit, les classes `PointCouleur` et `PointNoirEtBlanc` héritent de la classe `Point`. Cette classe est abstraite car elle possède la méthode abstraite `isAllume` qui n'a pas de sens pour un `Point` qui n'a pas de couleur. Par contre, cette méthode a un sens pour les points de couleur et les points noirs et blancs, la méthode `isAllume` est donc définie dans les classes `PointCouleur` et `PointNoirEtBlanc`.

La tentative `Point p = new Point(1,1);` aurait provoqué l'erreur de compilation : « `Point is abstract; cannot be instantiated` », mais une variable de type référence d'un `Point` peut référencer des objets de type `PointCouleur` et `PointNoirEtBlanc`, car ce sont des classes compatibles, on retrouve bien les caractéristiques du polymorphisme d'inclusion car c'est bien la bonne version de la méthode `isAllume` qui est appelée.

Exemple :

```
public class Test {
 static public void main(String [] args) {
  Point pc = new PointCouleur(1,2,9);
  Point pnb = new PointNoirEtBlanc(1,2,true);
  System.out.println(pc.isAllume());
  System.out.println(pnb.isAllume());
 }
}
abstract class Point {
 private int x, y;
 public  Point(int nx, int ny) {
    x = nx;
    y = ny;
 }
 abstract public boolean isAllume();
}
class PointCouleur extends Point{
 private int  c;
```

```
  public  PointCouleur(int nx, int ny, int nc) {
   super(nx, ny);
     c = nc;
  }
  public boolean isAllume() {
   return !(c == 0);
  }
 }
 class PointNoirEtBlanc extends Point{
  private boolean c;
  public PointNoirEtBlanc(int nx, int ny, boolean nc) {
   super(nx, ny);
   c = nc;
  }
  public boolean isAllume() {
   return c;
  }
 }
```

Résultat de l'exécution :

```
true
true
```

8.2 Interface

8.2.1 Principes et définitions

Le concept d'interface est une généralisation du concept de classe abstraite, c'est une classe qui ne possède aucun membre à l'exception de constantes statiques et de méthodes abstraites non statiques. Les constantes statiques doivent être obligatoirement initialisées à la déclaration (pas de bloc d'initialisation statique).

Une interface au sens de Java correspond donc à une classe totalement abstraite dans laquelle :

- toutes les méthodes sont implicitement[28] et obligatoirement[29] `public` et `abstract`
- il n'y a pas de constructeur
- tous les champs sont implicitement et obligatoirement `public`, `static` et `final` qui doivent donc être initialisés à la déclaration (pas de bloc d'initialisation statique).

On peut aussi considérer une interface comme un ensemble de services qui doivent être réalisés par les objets d'une classe afin d'être manipulés par un objet (généralement, d'un autre type). Ces services sont rendus par des méthodes identifiées qui devront être implémentées. C'est plutôt de cette façon qu'elles sont vues dans Java, les interfaces sont très utilisées dans les API, en particulier, dans la gestion des évènements et dans les collections.

Une interface se définit comme une classe via le mot-clé interface :

[28] Ils peuvent l'être explicitement

[29] Ceci ne peut pas être modifié

```
[public] interface InterfaceXXX {
  membres
}
```

Une classe implémente une ou plusieurs interfaces :

```
[public] class MaClasse implements InterfaceXXX, InterfaceYYY {
  membres
  définition des méthodes de InterfaceXXX et InterfaceYYY
}
```

Si un champ de deux interfaces implémentées par une classe ont le même nom, il y a ambiguïté, si ce champs est manipulé dans une classe implémentant les deux interfaces (double définition des constantes statiques), il y a une erreur de compilation[30] : « `reference to xxx is ambiguous, both variable xxx in YYY and variable xxx in ZZZ match` ». Si cette constante doit être manipulée, il faut la préfixer par le nom de son interface. Pour les méthodes, qui ne sont que déclarées, le problème ne se pose pas. Dans l'exemple qui suit, le champ `xxx` est suffixé par le nom de son interface `InterfaceZZZ` pour lever l'ambiguïté :

```
interface InterfaceYYY {
    int xxx=0;
    int methode();
}
interface InterfaceZZZ {
    int xxx=1;
    int methode();
}
class AAA implements InterfaceYYY, InterfaceZZZ {
  AAA() {
    int a = InterfaceZZZ.xxx;
  }
  public int methode() {
    return 1;
  }
}
```

Pour qu'une classe implémentant une interface ne soit pas abstraite, elle doit fournir une implémentation à toutes les méthodes spécifiées par l'interface, sinon elle devient une classe abstraite et doit être déclarée `abstract`. Comme les méthodes des interfaces sont publiques, il faut qu'elles soient déclarées publiques aussi lors de leur définition (sinon cela provoque l'erreur de compilation : « `xxx in YYY cannot implement xxx in ZZZ; attempting to assign weaker access privileges; was public` »).

Une interface peut servir de type, les classes l'implémentant servant à en instancier les objets doivent cependant être complètement définies.

[30] Ambiguïté classique dans le cas de l'héritage multiple

Une interface peut hériter d'une ou plusieurs interfaces. Dans ce cas, bien évidemment, les membres de la nouvelle interface doivent respecter les contraintes des interfaces (méthodes abstraites, …)

```
interface InterfaceYYY {
    membres
}
interface InterfaceZZZ {
    membres
}
interface InterfaceXXX extends InterfaceYYY, InterfaceZZZ{
    membres
}
```

Dans le cas de l'héritage, si plusieurs champs des interfaces de base ont le même nom, il y a ambiguïté et erreur de compilation[31] si un des champs dupliqués est manipulé dans l'interface dérivée ou dans une classe implémentant l'interface dérivée : « `reference to xxx is ambiguous, both variable xxx in YYY and variable xxx in ZZZ match` ». Le problème est résolu de la même façon que dans le cas d'implémentations multiples.

8.2.2 Exemple

Dans l'exemple suivant, la classe `EditGrandeur` constituant un éditeur de propriété de grandeurs physiques est défini. Pour manipuler des grandeurs physiques, l'éditeur a besoin de :

- Connaître le nom de la grandeur : `String name()`
- Connaître l'unité de la grandeur : `String unite()`
- Connaître la valeur de la grandeur : `double getValeur()`
- Modifier la valeur de la grandeur : `void setValeur(double nv)`

Ces quatre méthodes définissent la spécification de l'interface `Grandeur`. Toutes les grandeurs qui devront être manipulées par un objet de type `EditGrandeur` devront implémenter l'interface `Grandeur`. C'est le cas des classes `Masse` et `Vitesse`. Ce cas est relativement simple, généralement, dans le cas d'une classe préexistante, on passe par une classe interne qui implémente ce type d'interface.

Exemple :

```
public class Test {
 static public void main(String [] args) {
  Vitesse c = new Vitesse(300000000);
  Masse mTerre = new Masse(5.9736e24);
  EditGrandeur eg = new EditGrandeur();
  eg.print(mTerre);
  eg.print(c);
  eg.change(c, 299792458);
  eg.print(c);
```

[31] Ambiguïté classique dans le cas de l'héritage multiple

```
 }
}
interface Grandeur {
 String name();
 String unite();
 double getValeur();
 void setValeur(double nv);
}
class Masse implements Grandeur {
 private double valeur;
 private String unite = "kg";
 public Masse(double nv) {
  valeur = nv;
 }
 public Masse(){
  this(0.0);
 }
 public String name(){
  return "Masse";
 }
 public String unite(){
  return unite;
 }
 public double getValeur(){
  return valeur;
 }
 public  void setValeur(double nv){
  valeur = nv;
 }
}
class Vitesse implements Grandeur {
 private double valeur;
 private String unite = "m/s";
 public Vitesse(double nv) {
  valeur = nv;
 }
 public Vitesse(){
  this(0.0);
 }
 public String name(){
  return "Vitesse";
 }
 public String unite(){
  return unite;
 }
 public double getValeur(){
  return valeur;
 }
 public  void setValeur(double nv){
  valeur = nv;
```

```
  }
 }
 class EditGrandeur {
  public void  print(Grandeur g) {
   System.out.println(toString(g));
  }
  public String toString(Grandeur g) {
   return g.name() + " = " + g.getValeur() + " " + g.unite();
  }
  public void change(Grandeur g, double nv) {
   g.setValeur(nv);
  }
 }
```

Résultat de l'exécution :

```
Masse = 5.9736E24 kg
Vitesse = 3.0E8 m/s
Vitesse = 2.99792458E8 m/s
```

8.2.3 Interface Comparable et Arrays

Nous avons vu que Java dispose d'une classe `Arrays` (du package `java.util`) qui possède quelques fonctions permettant d'effectuer le tri et la recherche sur des tableaux d'éléments de types de base, mais cette classe permet aussi de traiter des objets, mais ces objets doivent être comparables. La classe des objets du tableau doit implémenter l'interface `Comparable`. Nous allons simplement illustrer l'utilisation des deux méthodes :

- **`static void sort(short[] a)`** : trie par ordre croissant les éléments du tableau `a`
- **`static int binarySearch(Object[] a, Object key)`** : effectue la recherche dichotomique de la valeur de `key` dans le tableau `a` si la valeur est trouvée l'indice de l'élément est retourné, sinon une valeur négative est retournée (il faut préalablement que le tableau soit trié par ordre croissant de préférence par `sort`)

L'interface `Comparable` nécessite l'implémentation de la méthode `compareTo`, elle pourrait être définie de la manière suivante :

```
interface Comparable {
   int compareTo(Object o);
}
```

La méthode `compareTo` retourne une valeur positive si l'objet courant est plus grand que l'objet passé en argument, 0 s'ils sont égaux, une valeur négative sinon.

Dans l'exemple développé ici, l'interface `Comparable` est implémentée par la classe `Point`. On considère qu'un point est plus grand qu'un autre s'il est plus éloigné de l'origine (deux objets équidistants sont égaux). Si la méthode reçoit un objet qui n'est pas de type `Point`, `-1` est retourné (Normalement, une exception devrait être lancée). Pour utiliser l'interface `Comparable`, il faut obligatoirement importer `java.util`.

Exemple :

```
 import java.util.*;
 public class Test {
  static public void main(String [] args) {
```

```
   Point []tp = new Point [20];
   for(int i = 0; i < tp.length; i++) {
    tp[i] = new Point((int)(Math.random()*100),
(int)(Math.random()*100));
   }
   Point p = tp[10].clone();
   System.out.println(Arrays.toString(tp));
   Arrays.sort(tp);
   System.out.println(Arrays.toString(tp));
   int index = Arrays.binarySearch(tp, p);
   if(index >= 0)
    System.out.println(tp[index] + " <-> " + p);
 }
}
class Point implements Comparable {
 private int x, y;
 public  Point(int nx, int ny) {
    x = nx;
    y = ny;
 }
 public  Point() {
    this(0,0);
 }
 public Point clone() {
  return new Point(x,y);
 }
 public int compareTo (Object o) {
  if(!(o instanceof Point)) return -1;
  Point p = (Point) o;
  if(x == p.x && y == p.y) return 0;
  return ((int)(Math.hypot(x, y) - Math.hypot(p.x, p.y))) ;
 }
 public String toString() {
  return "(" + x + ", " + y + ")";
 }
}
```

Résultat de l'exécution :

```
 [(46, 40), (29, 58), (54, 76), (21, 26), (68, 8), (95, 99), (71, 76),
(18, 27), (63, 72), (79, 29), (85, 49), (18, 64), (53, 37), (94, 10),
(29, 7), (57, 29), (41, 2), (45, 19), (52, 96), (28, 61)]
[(29, 7), (21, 26), (18, 27), (41, 2), (45, 19), (46, 40), (29, 58),
(53, 37), (57, 29), (18, 64), (28, 61), (68, 8), (79, 29), (54, 76),
(94, 10), (63, 72), (85, 49), (71, 76), (52, 96), (95, 99)]
(85, 49) <-> (85, 49)
```

9. Classes internes anonymes

Rappelons qu'une classe interne est une classe qui est définie à l'intérieur d'une autre classe (voir paragraphe §8 p101). Ces classes permettent de créer des objets (eux mêmes pouvant être anonymes) à partir de la dérivation d'une classe existante ou de l'implémentation d'une interface. La classe ainsi définie ne l'est qu'une fois pour créer un

objet, cette classe éphémère n'a pas de nom. Ce type de classe a été créé pour faciliter la gestion des évènements (un objet écouteur) dans le cadre de la conception des interfaces graphiques. A l'exception de ce cas particulier, les classes anonymes réduisant la lisibilité d'un programme, il faut se restreindre à la création d'un objet d'une classe dont on veut outrepasser quelques méthodes.

Une classe interne anonyme peut se définir de manière simplifiée à partir d'une classe de base, il faut appeler un des constructeurs de la classe de base :

```
ClasseBase objet = new ClasseBase(arg1, …, argn) {
Définition des méthodes outrepassées
}
```

Une classe interne anonyme se définit de manière simplifiée à partir d'une interface :

```
Interface objet = new Interface() {
Implémentation des méthodes
}
```

L'objet créé a pour type celui de la classe de base ou de l'interface. Dans les deux cas, de nouveaux membres peuvent être définis (ni statique, ni constructeur), mais uniquement à titre interne. Cela constitue un « mode d'héritage » très restreint.

Dans l'exemple qui suit, une classe interne anonyme est définie à partir de la classe `Point`. Elle outrepasse la méthode `print` de sa classe de base. Elle permet de créer un objet de type `Point` avec une méthode `print` différente de celle définie dans la classe de base.

Exemple :

```
public class Test {
 static public void main(String [] a) {
  Point p = new Point(1,1);
  p.print();
  p = new Point(1,1){
   public void print() {
   System.out.println("(" + getX() + ", " +getY() + ")");
   }
  };
  p.print();
 }
}
class Point {
 private int x, y;
 public Point(int nx, int ny) {
  x = nx;
  y = ny;
 }
 public int getX() {
  return x;
 }
 public int getY() {
  return y;
 }
```

```
  public void print() {
   System.out.println(x + ":" +y);
  }
 }
```

Résultat de l'exécution :

```
1:1
(1, 1)
```

10.Exercices de synthèses

10.1 Hiérarchie de classes des cétacés

Les cétacés constituent un ordre de mammifères placentaires, considéré comme monophylétique. Les cétacés ont tous en commun une adaptation spécifique au milieu marin. Cet ordre se subdivise en deux sous ordres les Mysticètes, les cétacés à fanons (on se limite ici aux baleines à bosses et aux baleines bleues), et les Odontocètes, les cétacés à dents (on se limite ici aux grands dauphins et aux orques). On s'intéressent ici à leur régime alimentaire :

- Tous les cétacés se nourrissent de sardines.
- Tous les mysticètes de nourrissent de krill et de plancton, la baleine bleue mangent aussi maquereaux et la baleine à bosse des harengs.
- Tous odontocètes de nourrissent d'anchois, de maquereaux, de mulets et de seiches, le grand dauphin mange aussi des crustacés, et l'orque des manchots et des phoques.

Tout cétacé est caractérisé par son poids, sa taille et son prénom. On demande de définir une méthode permettant de savoir si on peut donner un type de nourriture à un cétacé : Peut on donner à l'orque Willy du plancton ou au dauphin Flipper des anchois ?

Nous avons décomposé les cétacés en 7 classes :

La classe `Cetace`, racine de la hiérarchie de classes, qui possède les caractéristiques de tous les cétacés :

- `prenom`, `taille` et `poids`, le prénom étant sensé ne pas être modifié est déclaré `final`.

les méthodes associées :

- le constructeur qui initialise les trois champs précédents et la méthode `toString` qui retourne une chaîne de caractères représentant l'objet courant de type `Cetace`

Les classes `Mysticete` et `Odontocete` dérivent de la classe `Cetace`.

Les classes `BaleineBleue` et `BaleineABosse` dérivent de la classe `Mysticete`.

Les classes `GrandDauphin` et `Orque` dérivent de la classe `Odontocete`.

Chacune des classes possède son propre constructeur qui ne fait qu'appeler le constructeur de sa classe de base et gère à son niveau la nourriture que le cétacé qu'elle représente peut consommer :

- Le tableau `tRegime` correspond à la liste des aliments comestibles pour la classe
- La méthode `isNourriture` qui est redéfinie dans les classes dérivées teste que la chaîne passée correspond à un aliment du tableau `tRegime` ou bien qu'il est compatible avec ceux de la classe de base.

Dans le `main`, les classes dérivées sont bien compatibles avec leur classe de base, ce qui permet par exemple d'écrire : `Cetace bb = new BaleineBleue("Moby Dick", 34, 190000`. On affecte à la variable `bb` qui est de type référence d'un objet de type `Cetace` la référence d'un objet de type `BaleineBleue`. On remarquera que le polymorphisme d'inclusion fonctionne, par exemple, lors de l'appel `or.isNourriture("phoque")`, c'est bien la méthode `isNourriture` de la classe `Orque` qui est appelée et non celle de la classe `Cetace` (sinon `false` serait retourné).

Solution possible :

```
public class Test {
 static public void main(String [] args) {
  Cetace bb = new BaleineBleue("Moby Dick", 34, 190000);
  Cetace gd = new GrandDauphin("Flipper", 3.5, 500);
  Cetace or = new Orque("Willy", 7, 6000);
  System.out.println(or);
  System.out.println(bb);
  System.out.println(gd);
  System.out.println(gd.isNourriture("sardine"));
  System.out.println(bb.isNourriture("krill"));
  System.out.println(or.isNourriture("phoque"));
  System.out.println(gd.isNourriture("phoque"));
 }
}
class Cetace {
 private double poids;
 private double taille;
 final private String prenom;
 private final String []tRegime = {"sardine"};
 public Cetace(String nprenom, double ntaille, double npoids) {
  prenom = nprenom;
  taille = ntaille;
  poids = npoids;
 }
 public boolean isNourriture(String s) {
  for (int i = 0; i < tRegime.length; i++) {
   if(s.equalsIgnoreCase(tRegime[i])) return true;
  }
  return false;
 }
 public String toString(){
  return prenom + " mesure " + taille + "m et pèse " + poids +"kg";
 }
}
class Mysticete extends Cetace{
 private final String []tRegime = {"plancton", "krill"};
 public Mysticete(String nprenom, double ntaille, double npoids) {
  super(nprenom, ntaille, npoids);
 }
 public boolean isNourriture(String s) {
```

```
   if(super.isNourriture(s)) return true;
   for (int i = 0; i < tRegime.length; i++) {
    if(s.equalsIgnoreCase(tRegime[i])) return true;
   }
   return false;
  }
 }
 class Odontocete extends Cetace{
  private final String []tRegime = {"anchois","maquereau",
 "mulet","seiches"};
  public Odontocete(String nprenom, double ntaille, double npoids) {
   super(nprenom, ntaille, npoids);
  }
  public boolean isNourriture(String s) {
   if(super.isNourriture(s)) return true;
   for (int i = 0; i < tRegime.length; i++) {
    if(s.equalsIgnoreCase(tRegime[i])) return true;
   }
   return false;
  }
 }
 class BaleineBleue extends Mysticete {
  private final String []tRegime = {"maquereau"};
  public BaleineBleue(String nprenom, double ntaille, double npoids) {
   super(nprenom, ntaille, npoids);
  }
  public boolean isNourriture(String s) {
   if(super.isNourriture(s)) return true;
   for (int i = 0; i < tRegime.length; i++) {
    if(s.equalsIgnoreCase(tRegime[i])) return true;
   }
   return false;
  }
 }
 class BaleineABosse extends Mysticete {
  private final String []tRegime = {"hareng"};
  public BaleineABosse(String nprenom, double ntaille, double npoids) {
   super(nprenom, ntaille, npoids);
  }
  public boolean isNourriture(String s) {
   if(super.isNourriture(s)) return true;
   for (int i = 0; i < tRegime.length; i++) {
    if(s.equalsIgnoreCase(tRegime[i])) return true;
   }
   return false;
  }
 }
 class Orque extends Odontocete {
  private final String []tRegime = {"manchot","phoque"};
  public Orque(String nprenom, double ntaille, double npoids) {
```

```
   super(nprenom, ntaille, npoids);
  }
  public boolean isNourriture(String s) {
   if(super.isNourriture(s)) return true;
   for (int i = 0; i < tRegime.length; i++) {
    if(s.equalsIgnoreCase(tRegime[i])) return true;
   }
   return false;
  }
}
class GrandDauphin  extends Odontocete {
 private final String []tRegime = {"crustacé"};
 public GrandDauphin(String nprenom, double ntaille, double npoids) {
  super(nprenom, ntaille, npoids);
 }
 public boolean isNourriture(String s) {
  if(super.isNourriture(s)) return true;
  for (int i = 0; i < tRegime.length; i++) {
   if(s.equalsIgnoreCase(tRegime[i])) return true;
  }
  return false;
 }
}
```

Résultat de l'exécution :

```
Willy mesure 7.0m et pèse 6000.0kg
Moby Dick mesure 34.0m et pèse 190000.0kg
Flipper mesure 3.5m et pèse 500.0kg
true
true
true
false
```

10.2 Classes pour représenter des salariés

L'objectif de cet exercice est de définir plusieurs classes permettant de représenter les salariés et en particulier de calculer leur salaire. Les salaires sont payés en écus martiens.

Un salarié est caractérisé par son nom, son ancienneté et son salaire. Il existe deux types de salariés : les cadres et les employés. Le salaire mensuel des cadres est fixe quel que soit le nombre d'heures effectués :

- Salaire net = salaire annuel brut * 0.8 / 12

Celui des employés dépend du nombre d'heures effectués, il est égal au salaire de base, auquel il faut ajouter les heures supplémentaires sur lesquelles aucune charge n'est payée, le taux horaire est fixe pour tous les employés et vaut 10 écus martiens de l'heure :

- Salaire net = salaire annuel brut * 0.8 / 12 + heures supplémentaire * taux horaire

La solution proposée consiste à définir une classe de base `Salarie` de laquelle dérivent les deux classes `Employe` et `Cadre`. Cette classe regroupe les caractéristiques des salariés, mais celles des employés aussi, au travers de la gestion des heures

supplémentaires. La méthode `calculSalaire` de la classe Salarie est redéfinie dans la classe `Employe` afin de prendre en compte le calcul des heures supplémentaire, le calcul de base étant fait dans la classe `Salarie`, il n'est pas redéfini dans la classe `Cadre`. La méthode `setHeureSup` est redéfinie aussi dans la classe `Cadre` (ce n'est pas une obligation, mais une simple précaution qui force à `0` le champ `heureSup`). Le défaut de cette solution est que la gestion des heures supplémentaires est remontée au niveau de la classe `Salarie` alors qu'elle ne concerne que les employés.

Solution possible :

```
public class Test {
 static public void main(String [] args) {
  Salarie []ts = new Salarie [5];
  ts[0] = new Employe("Martin", 5, 1500);
  ts[0].setHeureSup(5.5);
  ts[1] = new Employe("Durant", 7, 1600);
  ts[1].setHeureSup(2.0);
  ts[2] = new Employe("Dupond", 7, 1600);
  ts[2].setHeureSup(2.0);
  ts[3] = new Cadre("Clinton", 10, 3200);
  ts[4] = new Cadre("Obama", 5, 3300);
  for(Salarie s : ts)
   System.out.println(s + " : " + s.calculSalaire());
 }
}
class Salarie {
 private String nom;
 private int anciennete;
 private double salaireBrut;
 private double heureSup;
 private static final double tauxCharge = 0.8;
 public Salarie(String nnom, int nanciennete, double nsalaireBrut){
  nom = nnom;
  anciennete = nanciennete;
  salaireBrut  = nsalaireBrut;
 }
 public String toString() {
  return nom;
 }
 public double getTauxCharge() {
  return tauxCharge;
 }
 public double getSalaireBrut() {
  return salaireBrut;
 }
 public double calculSalaire() {
  return getSalaireBrut() * getTauxCharge() / 12;
 }
 public void setHeureSup(double nheureSup){
  heureSup=nheureSup;
 }
```

```
  public double getHeureSup(){
   return heureSup;
  }
 }
 class Cadre extends Salarie {
  public Cadre(String nnom, int nanciennete, double nsalaireBrut){
   super(nnom, nanciennete, nsalaireBrut);
  }
  public void setHeureSup(double nheureSup){
   super.setHeureSup(0);
  }
 }
 class Employe extends Salarie {
  private static  final double tauxHoraire = 10;

  public double calculSalaire() {
   return super.calculSalaire() + 10*getHeureSup();
  }
  public Employe(String nnom, int nanciennete, double nsalaireBrut){
   super(nnom, nanciennete, nsalaireBrut);
  }
 }
```

Résultat de l'exécution :

```
Martin : 155.0
Durant : 126.66666666666667
Dupond : 126.66666666666667
Clinton : 213.33333333333334
Obama : 220.0
```

Nous aurions pu opter pour une autre solution en définissant la méthode `setHeureSup` dans les classe `Employe` et `Cadre`, en la rendant abstraite dans la classe Salarie, en effet, cette méthode n'a pas de sens pour la classe `Cadre`. De plus, le champ `heureSup` passe dans la classe `Employe`, mais pas dans la classe `Cadre`. Cette seconde solution semble mieux structurée en terme de répartition, mais nécessite de définir dans la classe `Cadre` la méthode `setHeureSup` qui ne fait rien.

Solution possible :

```
abstract class Salarie {
 private String nom;
 private int anciennete;
 private double salaireBrut;
 private static  final double tauxCharge = 0.8;
 public Salarie(String nnom, int nanciennete, double nsalaireBrut){
  nom = nnom;
  anciennete = nanciennete;
  salaireBrut = nsalaireBrut;
 }
 public String toString() {
  return nom;
 }
```

```
  public double getTauxCharge() {
   return tauxCharge;
  }
  public double getSalaireBrut() {
   return salaireBrut;
  }
  public double calculSalaire() {
   return getSalaireBrut() * getTauxCharge() / 12;
  }
  public abstract void setHeureSup(double nheureSup);
 }
 class Cadre extends Salarie {
  public Cadre(String nnom, int nanciennete, double nsalaireBrut){
   super(nnom, nanciennete, nsalaireBrut);
  }
  public void setHeureSup(double nheureSup){
   ;
  }
 }
 class Employe extends Salarie {
  private static  final double tauxHoraire = 10;
  private double  heureSup;
  public double calculSalaire() {
   return super.calculSalaire() + 10*heureSup;
  }
  public Employe(String nnom, int nanciennete, double nsalaireBrut){
   super(nnom, nanciennete, nsalaireBrut);
  }
  public void setHeureSup(double nheureSup){
   heureSup=nheureSup;
  }
 }
```

10.3 Stock

On demande de définir une classe Stock. Un stock doit permettre de stocker des objets stockables. Un « objet » stockable doit avoir deux propriétés : un volume et une température. Un stock ne peut stocker que 100 « objets » dont la température de stockage ne peut être supérieure à la température du stock et dont le volume ne peut excéder le volume résiduel, donc un stock possède un volume de stockage initial (on suppose que le stockage se fait sans perte de volume, c'est-à-dire que le volume après stockage est égal à l'ancien moins le volume de l'objet stocké).

Par ailleurs, on suppose que l'on dispose de produits et d'outils qui peuvent être stockés, les articles ont un prix unitaire, pas les outils qui ne sont pas vendus, mais les deux possèdent une référence et un poids.

Nous avons défini l'interface `Stockable` disposant des méthodes `temperature` et `volume` correspondant aux services demandés pour qu'un objet soit stockable, à laquelle une méthode d'affichage `print` est ajoutée.

La classe abstraite `Article` regroupe les caractéristiques communes aux deux classes dérivées `Produit` et `Outil`, elle est abstraite car elle n'implémente pas

complètement l'interface `Stockable`, la méthode d'affichage `print` est définie dans les deux classes dérivées.

La classe `Stock` ne manipule que des objets `Stockable` dont elle utilise exclusivement les méthodes spécifiées par l'interface `Stockable`. Un « objet » ne peut être ajouté (méthode `add`) en stock que si sa capacité est suffisante, si le nombre d' « objets » et le volume résiduel sont suffisants et si la température convient.

Solution possible :

```
public class Test {
 static public void main(String [] args) {
  Stock s = new Stock(20, 10.0, 100.0);
  for(int i = 0; i < 15; i++) {
   Stockable p = null;
   if ((i&1) == 1)p = new Produit("P"+i*101, 8 + (Math.random() * 10),
                    (Math.random() * 10),Math.random(), Math.random());
   else p =new Outil("O"+i*202, 8 + (Math.random() * 10),
                 (Math.random() * 10),Math.random());
   s.add(p);
  }
  s.print();
 }
}
interface Stockable {
 double temperature();
 double volume();
 void print();
}
abstract class Article implements Stockable {
 private double temperature;
 private double volume;
 private double poids;
 private String reference;
 public Article(String ref, double temp, double vol, double pds){
  reference = ref;
  temperature = temp;
  poids = pds;
  volume = vol;
 }
 public double temperature(){
  return temperature;
 }
 public double volume(){
  return volume;
 }
 public String getReference() {
  return reference;
 }
 public double getPoids(){
  return poids;
 }
```

```
 public String toString() {
  return reference + " : " + temperature + " : " + volume + " : " +
poids;
 }
}
class Produit extends Article {
 private double prix;
 public Produit(String ref, double temp, double vol, double pds,
                double nprix){
  super(ref, temp, vol, pds);
  prix = nprix;
 }
 public String toString() {
  return super.toString() + " : " +prix;
 }
 public void print() {
  System.out.println("Article --> " + toString());
 }
}
class Outil extends Article {
 private double prix;
 public Outil(String ref, double temp, double vol, double pds){
  super( ref, temp, vol, pds);
 }
 public String toString() {
  return super.toString();
 }
 public void print() {
  System.out.println("Outil --> " + toString());
 }
}
class Stock {
 private Stockable []ts;
 private double temperature;
 private double volume;
 private double volumeRes;
 private int index;
 public Stock(int dim, double temp, double vol){
  temperature = temp;
  volume = vol;
  ts = new Stockable [dim];
  index = 0;
  volumeRes = volume;
 }
 public boolean add(Stockable s) {
  if(index >= ts.length)return false;
  if(s.volume() > volumeRes)return false;
  if(temperature > s.temperature())return false;
  ts[index] = s;
  volumeRes -= s.volume();
```

```
   index ++;
   return true;
  }
  public void print() {
   System.out.println("Stock : " + temperature + " " + volumeRes + " " +
(ts.length - index));
   for(int i = 0; i < index; i++)
    ts[i].print();
  }
}
```

Résultat de l'exécution :

```
Stock : 10.0 43.77538133801044 8
Outil --> O0 : 13.501388367196878 : 2.8945273955383732 :
0.8523991328570665
Outil --> O404 : 16.281700264064987 : 6.247756525763373 :
0.17969518482273406
Article --> P303 : 14.141781871765712 : 4.029887834697153 :
0.36424829100811174 : 0.5144545556876078
Outil --> O808 : 16.351846589111105 : 1.3931370893080985 :
0.44885222979983797
Article --> P505 : 11.229856673170717 : 2.3714461792675356 :
0.8442388477368004 : 0.8006025294830829
Outil --> O1212 : 10.05335595027704 : 5.506950745164119 :
0.4263651997483483
Article --> P707 : 15.069452289823179 : 8.44082608213318 :
0.3953607661188414 : 0.7327005932653776
Outil --> O1616 : 16.468256020433206 : 6.636967326160381 :
0.16325827632881018
Article --> P909 : 16.07893526143809 : 8.361130978099535 :
0.5755796182660277 : 0.2374799387585681
Article --> P1111 : 16.659885044643513 : 4.504677757454197 :
0.7874632737590705 : 0.6342691188676408
Outil --> O2424 : 11.95643170456816 : 2.1478226627294106 :
0.1422134370973901
Outil --> O2828 : 10.32918482588913 : 3.6894880856741974 :
0.7830425757071448
```

Annexes

1. Bibliographie

1.1 Quelques ouvrages généraux

- **Programmer en Java**. Claude Delannoy. Editions Eyrolles.
- **Les cahiers du programmeur Java 1.4 et 5.0**. Emmanuel Puybaret. Editions Eyrolles.
- **Au coeur de Java 2, Volume 1 & 2.** Cay S Hortsmann & Gary Cornell. Editions Campus Press.

1.2 Quelques sites Internet de référence

- http://pagesperso-orange.fr/emmanuel.remy/index.htm : Un cours en français très complet avec de nombreux exemples (le C++ est aussi abordé).
- http://www.eteks.com/coursjava/ : Un manuel en français complet sur Java (versions antérieures à Java 1.5), l'auteur a, par ailleurs, écrit des ouvrages sur Java.
- http://www.jmdoudoux.fr/accueil_java.htm : Un manuel en français de référence très détaillé sur Java accompagné d'un manuel sur l'environnement Eclipse.
- http://java.sun.com/docs/books/tutorial/ : Le tutorial en anglais de référence de SUN qui aborde tous les aspects de Java
- http://java.developpez.com/ : Site de référence des développeurs francophones, c'est une mine d'informations : cours et tutoriaux sur le Java, critiques de livres, présentation d'outils de développement, forum , …
- http://www.ukonline.be/programmation/java/tutoriel/principal.php : Un tutorial en français sur l'essentiel de Java
- http://rmdiscala.developpez.com/cours/ : cours très général qui aborde de nombreux aspects de l'informatique et qui dispose d'une partie importante consacrée à Java.
- http://penserenjava.free.fr/ : Traduction en français d'un livre de référence sur Java

`abstract`, 201
accesseur, 93
allocation *Voir* new
altérateur, 93
API, 118
argument, 38, 65
`Arrays`, 139, 207
associativité, 22
autoboxing, 134
blocs d'initialisation, 80
blocs d'initialisation statique, 87
boucle, 30
`break`, 34
champ, 61
champ statique, 84
`Character`, 131
classe, 59
classe abstraite *Voir* abstract
classe de base, 158
classe dérivée, 158
classe externe *Voir* classe interne
classe imbriquée *Voir* classe interne
classe interne, 101, 208
clonage, 93
`clone`, 183
commentaire, 13
`Comparable`, 207
comparateur, 93
constantes littérales, 14
constructeur, 70, 163
`continue`, 34
conversion, 20
création *Voir* new
déclaration, 15
`default`, 34
`do while`, 30
`Double`, 130
downcasting Voir sousclassement
droit d'accès, 61, 160
droits d'accès, 69
`enum`, 110
énumérations *Voir* enum
expressions, 18
`extends`, 159
`final`, 15, 38, 90, 200
fonction *Voir* méthode
for, 31
for each, 32
friendly, 61, 160
héritage, 158
identificateur, 15
`if`, 32
immuable, 97
`import`, 115
initialisation, 17
`instanceof`, 197
instruction, 30
`Integer`, 128
interface, 203
`java.lang`, 118
`java.util`, 139, 207
`length`, 27
liaison dynamique, 189
`main`, 49
masquage, 68, 174
`Math`, 136
méthode, 37, 62
méthode statique, 85
mutable, 97
`name`, 110
`new`, 25
`null`, 24
`Object`, 180
objet, 59
opérateurs *Voir* expressions
`ordinal`, 110
outrepassement *Voir* redéfinition
overloading Voir surcharge
overriding Voir redéfinition
`package`, 115
paquetage *Voir* package
paramètre, 38
polymorphisme, 44, 189
portée, 68, 76, 173
priorité, 22
`private`, 61, 160
promotion *Voir* conversion
`protected`, 160
`public`, 61, 160
récursive *Voir* récursivité
récursivité, 42
redéfinition, 167
référence, 23
`return`, 38
sélection, 32
signature, 44
sousclassement, 185
`static`, 90, 116, 175, 196
`String`, 119
`StringBuffer`, 122
structures de contrôle *Voir* instruction
`super`, 173
surcharge, 12, 44, 45, 69, 123, 167, 168, 170, 189, 192, 198
surclassement, 184
`switch case`, 34
tableau, 23, 28, 81
`this`, 75
`toString`, 125
types primitifs, 14
upcasting, 184, *Voir* surclassement
`uper`, 163
`values`, 110
variable locale, 16
`while`, 30
wrapper, 126