REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

Ministère de l'enseignement supérieur et de la recherche scientifique

Université d'Oum El-Bouaghi

Faculté des Sciences Exactes et des Sciences de la Nature et de la Vie

Département de Mathématiques et Informatique

Cours sur la Programmation Orientée Objet en Java

Conforme à l'offre de formation pédagogique du socle commun proposé par le Ministère de l'Enseignement Supérieur et de la Recherche Scientifique pour l'année universitaire 2015-2016

Domaine : **Math et Informatique**

Filière : **Informatique**

Niveau : **2ème année licence**

Proposé par **Dr. NINI Brahim**

Edition 1.0

Table des matières

Avant-propos i

Description de la matière telle que décrite dans l'offre de formation ii

Chapitre 1 *Introduction à la Programmation Orientée Objet (POO)* 1

1. Notions de base 2

1.1. Programmation procédurale 2

1.2. Programmation par objets 2

1.2.1. Objet 2

1.2.2. Classes 3

2. Historique 4

2.1. POO 4

2.2. Java 5

Les classes 6

Chapitre 2 *Les classes* 7

1. Bases du langage Java 7

1.1. Types primitifs 7

1.2. Variables 7

1.3. Tableaux 7

1.4. Structures 8

2. Les classes 8

2.1. Définition de classe 8

2.2. Programme java 9

2.3. Instanciation 10

2.3.1. Variables d'instance 10

2.3.2. Méthodes d'instances et leur appel 11

2.3.3. Objet Java 12

2.4. Membres de classe 12

2.4.1. Variables de classe 12

2.4.2. Méthodes de classe 13

2.5. Contrôle d'accès 14

2.6. Méthodes particulières 14

3. Constructeurs et destructeurs 15

3.1. Constructeurs 15

3.2. Destructeurs 16
3.3. Le ramasse miettes 16
4. Encapsulation 17
5. Classes imbriquées 18
6. Paquetages (packages) 18
6.1. Paquetages standards 18
6.2. Création de paquetages 19
6.3. Quelques classes particulières 20
6.3.1. Classe Object 20
6.3.2. Classe String 20
6.3.3. Classes “Wrapper” 20
6.3.4. Classes “Math” 20
6.3.5. Classes “System” 21
Chapitre 3 *Héritage et Polymorphisme* 23
1. Généralités 23
2. Héritage 23
2.1. Principe 23
2.2. Constructeur de la classe mère 24
2.3. Membres protégés 24
2.4. Référence 24
2.5. Avantages de l’héritage 26
3. Polymorphisme 26
3.1. Définition 26
3.2. Surcharge 26
3.3. Redéfinition de méthode 26
3.4. Sélection de la méthode à exécuter 28
4. Classes abstraites 29
Interface et implémentation 31
Chapitre 4 *Interface et implémentation* 32
1. Introduction 32
2. Concepts 32
3. Importance des interfaces 33
4. Exemples d’interfaces 34
Exemple 1 34
Exemple 2 34

Interface graphique et Applet 36
Chapitre 5 *Interface graphique et Applet* 37
1. Composants, gestionnaire d'affichage 37
1.1. API AWT 37
1.2. Utilisation de AWT 37
Fenêtres 37
Contenant et environnement graphique 38
2. Mise en page 40
2.1. Dimensionnement des composants 40
2.2. Positionnement des composants 40
2.3. Utilisation des Layout 41
3. Gestion des événements et écouteur 42
4. Applet 44
4.1. Intégration d'applet dans une page HTML 44
4.2. Utilisation générale d'une applet 45
4.3. Quelques méthodes de l'applet 45
Exercices 46
Références 57

Avant-propos

Cet ouvrage est un cours qui s'adresse aux étudiants de la deuxième année licence de la filière informatique. Il est conforme à l'offre de formation pédagogique du socle commun proposé par le Ministère de l'Enseignement Supérieur et de la Recherche Scientifique à partir de l'année universitaire 2014-2015.

Cet ouvrage constitue un cours qui a pour objectif l'introduction des concepts de base du langage de programmation Java comme stipulé dans la description détaillée de la matière du programme proposé. Il est enseigné au troisième semestre de la formation des étudiants de la filière. Il s'adresse particulièrement aux étudiants qui ont déjà quelques notions de programmation dans un langage procédural quelconque mais des connaissances du langage C sont préférables.

Cet ouvrage ne se veut pas être une référence complète du langage Java. Au contraire, il ne revoit pas, par exemple, les structures de base du langage parce qu'on peut les considérer comme étant « communes » à beaucoup de langages. C'est le cas par exemple des structures de test et de boucle qui ne sont présentées que brièvement. Certaines particularités du langage ne sont pas aussi présentées malgré leurs importances pour la simple raison qu'elles ne font pas parties du programme officiel. De plus, du fait que cet ouvrage a une tendance d'être beaucoup plus un cours pédagogique qu'un livre, certains passages ne sont pas très détaillés dans un but de les expliquer en cours. Cependant, il reste amplement suffisant pour les étudiants pour qu'il soit pour eux une référence de base satisfaisante.

A sa fin, l'ouvrage contient un ensemble d'exercices développés personnellement ou pris de certaines références qui couvrent la majorité des parties importantes du langage. Ils ne sont cependant présentés que dans le but de donner aux étudiants un moyen pour se familiariser pédagogiquement avec les concepts du langage Java et avec ses différentes structures. Aussi, uniquement certains exercices sont présentés avec leurs solutions. C'est dans une prochaine édition de cet ouvrage que je compte rajouter un ensemble d'exercices avec solutions plus complets. J'ai également organisé les exercices dans un ordre croissant de difficulté en suivant l'ordre des concepts du langage présentés.

Enfin, cet ouvrage est un document librement téléchargeable sur ma page web (http://relacs2.univ-oeb.dz/nini.html). Toute remarque est donc la bienvenue. Je tâcherai de corriger les erreurs que l'on me signalera dans la mesure du possible, et d'apporter les modifications nécessaires.

Description de la matière telle que décrite dans l'offre de formation

Intitulé de la Matière **: Programmation Orientée Objet (POO)**

Semestre **: 3**

Objectifs de l'enseignement **:** Ce cours a pour objectif l'introduction des concepts de base du langage Java. Il traite spécialement les thèmes tels que: Technologie orientée objet, encapsulation, héritage, polymorphisme, translation dynamique. Le cours développe les notions de base du langage en particulier: les classes, les objets, les constructeurs, finalizer, les méthodes et les variables d'instances, les sous classes, les interfaces et l'héritage multiple, les packages et la notion de visibilité en java, les méthodes et les variables de classe, et les classes abstraites.

L'étudiant est censé avoir acquis pendant le module les compétences suivantes:

1- L'essence de la programmation objet en java
2- Lire et comprendre des programmes en java
3- Ecrire la solution d'un problème en java
4- Ecrire des applications sophistiquées (utilisation de structures de données avancées)

Connaissances préalables recommandées

Connaissance du langage C souhaitée

Contenu de la matière **:**

1. Introduction à la Programmation Orienté Objet
    a. Notions de base
    b. Historique
    c. Utilisation des TAD
2. Les classes
    a. Déclaration des classes
    b. Les constructeurs et destructeurs
    c. Les méthodes d'accès
    d. Encapsulation
3. Héritage et polymorphisme
    a. Généralités
    b. Surcharge et redéfinition
    c. Héritage : Références
    d. Polymorphisme
    e. Les classes abstraites
4. Interface et implémentation
    a. Principe
    b. Application
5. Interface graphique et Applet
    a. Composants, gestionnaire d'affichage

b. Mise en page
c. Gestion des événements et écouteur
d. Applet

Mode d'évaluation **:** Continu et Examen

Références :

1. Le site officiel de Sun Microsystems : *fr.**sun**.com/*
2. Le livre Penser Java : *bruce-eckel.developpez.com/**livres/java**/traduction/tij2/*
3. Conception objet en java avec bluej de david barnes. pearson education France
4. Java outside in de Bill Campbell. Cambridge University press

Chapitre 1

Introduction à la Programmation Orientée Objet (POO)

Chapitre 1

Introduction à la Programmation Orientée Objet (POO)

1. Notions de base

1.1. Programmation procédurale

En programmation procédurale (Pascal, C, …), un programme est une suite d'instructions exécutées par une machine.

- Il s'agit de répondre à la question: Que veut-on faire? L'accent est mis sur les actions.
- Il y a séparation entre données et fonctions : problème lorsqu'on change les structures de données.
- Les procédures s'appellent entre elles et peuvent modifier les mêmes données : problème lorsqu'on veut modifier une procédure.

D'où une autre vision de la programmation.

1.2. Programmation par objets

1.2.1. Objet

Un programme devient un ensemble d'objets en interaction. Pour modéliser un logiciel de trafic routier par exemple, on peut faire intervenir les objets suivants :

- les feux tricolores
- les carrefours
- les véhicules
- les agents de la circulation.

Objet = données + opérations sur ces données (méthodes).

Un objet est composé de 2 parties :

- partie **interface**: opérations qu'on peut faire dessus (partie publique)
- partie **interne**: données de l'objet (partie privée)

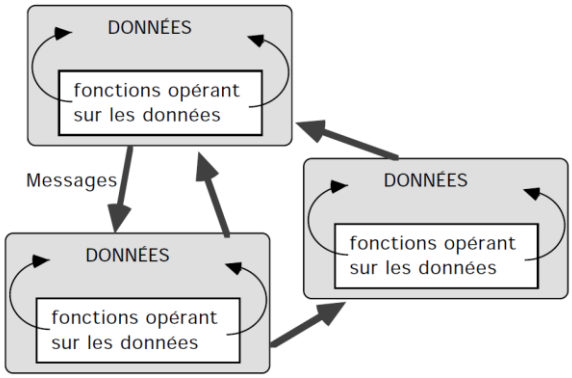
EXECUTION : les entités collaborent pour résoudre le problème final en **s'envoyant des messages**.

Les utilisateurs (i.e. les éléments extérieurs) de l'objet ne voient que la partie interface. Envoyer un message à un objet, c'est lui demander d'exécuter une de ses méthodes.

EXEMPLE: la Toyota Corolla immatriculée 2015 du recteur de l'université est un objet

Objet: CS_du_chef
genre : Toyota
immatriculation : 2015

NbPlaces : 5
propriétaire: Recteur
arrêter()
Démarrer()
fin objet



REMARQUE : le regroupement de code et de données au sein d'un objet est dit ***encapsulation***. C'est un masquage d'information au monde extérieur.

1.2.2. Classes

C'est un **modèle** décrivant le contenu et le comportement des futurs objets de la classe. Le contenu = les données, le comportement = les méthodes.

EXEMPLE DE CLASSE

Classe: Véhicule
genre
immatriculation
NbPlaces
propriétaire
arrêter()
marcher()
fin objet

Un exemple particulier d'une classe s'appelle une **instance** de la classe ou un objet de cette classe.

Programmation procédurale	VARIABLE	TYPE
Programmation orientée objets	OBJET	CLASSE

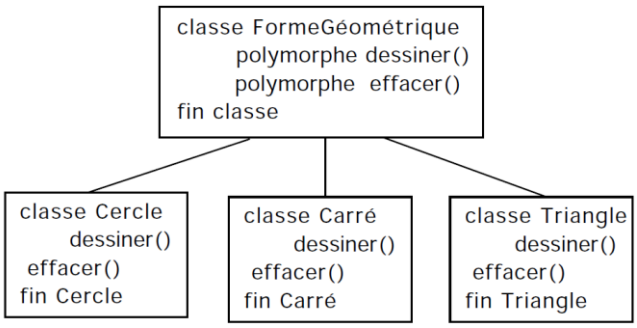
HERITAGE: construire une classe à partir d'une (d') autre(s). C'est une dérivation pour :

- reprendre intégralement ce qui a déjà été fait et pouvoir l'enrichir.
- regrouper en un seul endroit ce qui est commun à plusieurs : les modifications des éléments communs ne se font qu'à un seul endroit.

EXEMPLE: automobile et camion hérite (ou dérive) de la classe Véhicule. Dans la classe véhicule on regroupe les caractéristiques communes aux camions et aux automobiles

POLYMORPHISME: un même nom de méthode avec plusieurs implantations.

EXEMPLE:



A l'aide du polymorphisme, la détermination de la bonne fonction effacer() et dessiner() est faite automatiquement au moment de l'exécution.

2. Historique

2.1. POO

Début en Norvège à la fin des années 60: développement de Simula (Simple Universal Language) = Algol + concepts de POO.

Années 1970: Smalltalk = prototype des langages objets, développé chez Xerox.

Au cours des années 80: intérêt pour les interfaces graphiques et apparition de nouveaux langages

- Eiffel : fortement typé, entièrement orienté objet
- C++, extension du C: Ajout de spécificités permettant la mise en œuvre des concepts de la POO
- Object Pascal (Delphi)

Années 1990: Les années 1990 voient l'âge d'or de l'extension de la programmation par objet dans les différents secteurs du développement logiciel. Depuis, la programmation par objet n'a cessé d'évoluer aussi bien dans son aspect théorique que pratique:

- l'analyse objet (AOO ou OOA en anglais) ;
- la conception objet (COO ou OOD en anglais) ;
- les bases de données objet (SGBDOO) ;

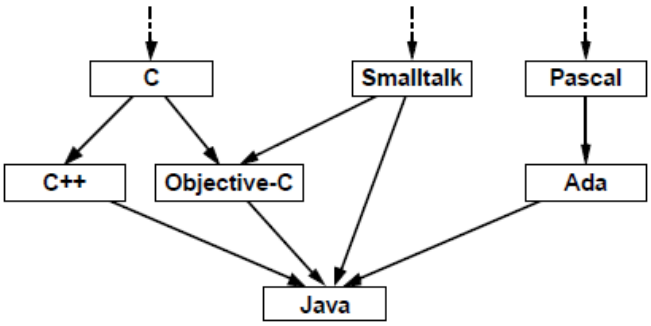
Aujourd'hui, la programmation par objet est vue davantage comme un paradigme, le paradigme objet, que comme une simple technique de programmation. C'est pourquoi, lorsque l'on parle de nos jours de programmation par objets, on désigne avant tout la partie codage d'un modèle à objets obtenu par AOO et COO. La POO est une programmation par composants ayant pour objectifs :

- Faciliter la réutilisation du code
- Réutiliser des fragments de code
    - développés dans un cadre différent
    - Fait appel aux notions d'encapsulation, d'abstraction
- Faciliter l'évolution du code
- Améliorer la conception et la maintenance des grands systèmes

2.2. Java

Le langage Java trouve ses origines dans les années 1990. Quelques ingénieurs de SUN Microsystems ont commencé à parler d'un projet d'environnement indépendant du hardware pouvant facilement permettre la programmation d'appareil aussi variés que les téléviseurs, les magnétoscopes,... James Grosling développa en 1992 un premier langage permettant de programmer dans cet environnement : Oak, ce fut un échec.

Bill Joy (co-fondateur de SUN Microsystems) et devant la montée en puissance d'Internet, il lui a semblé intéressant de proposer un tel langage (et un tel environnement). Dès lors tous s'accélère. Oak est renommé (en 1995) Java et est soumis à la communauté Internet grandissante. Une machine virtuelle, un compilateur ainsi que de nombreuses spécifications sont données gratuitement. Aujourd'hui, après de nombreuses améliorations (parfois modifications) Java n'est plus uniquement une solution liée à Internet : de plus en plus de sociétés (ou de particuliers) utilise ce langage pour leurs développements (de toutes sortes).



CARACTÉRISTIQUES

- Le code Java est compilé (Byte Code) pour une machine dite virtuelle JVM (Java Virtual Machine) pour une raison de portabilité (c'est-à-dire qui n'a pas d'existence physique, mais son concept peut être reproduit sur une machine réelle). Lors de l'exécution le code est transformé en un code machine compréhensible par le microprocesseur utilisé.
- La gestion de la mémoire n'est plus à la charge du programmeur. Des techniques de désallocation automatisée de la mémoire.
- Java est aussi multithreadé et distribué.
- Java possède une API (Application Programming Interface) très développée. C'est par elle que le programmeur peut accéder à toutes les ressources de la machine, et celles du réseau Internet (les packages (ensembles de classes) sont fournis en standard avec Java).

Chapitre 2

Les classes

Chapitre 2

Les Classes

1. Bases du langage Java

Distribué gratuitement avec des versions pour de nombreux OS avec une documentation en ligne: www.oracle.com/technetwork/java/index.html

Le compilateur de java ne génère pas du code exécutable mais du **byte code** qui est exécuté par une machine virtuelle (JVM) et qui est indépendant de la machine. Cela permet d'avoir du code portable et mobile mais moins efficace que le code exécutable. Chaque environnement (matériel + système) possède sa propre JVM.

1.1. Types primitifs

- byte // (8 bits) -128 à 127
- short // (16 bits) -32768 á 32767
- int // (32 bits) -2147483648 à 214748364732 bits
- long // (64 bits) -9223372036854775808 à 9223372036854775807
- char // (16 bits) 'a' ou '\u0000' à '\uffff' pour un caractère quelconque
- float // (32 bits) simple précision
- double // (64 bits) double précision
- boolean // (8 bits) true ou false.

REMARQUE: Le type String n'est pas un type primitif.

1.2. Variables

Les variables d'utilisation courante sont dites locales. Elles sont déclarées dans une méthode, un constructeur ou un bloc. Sorties de leur espace, elles n'existent plus, la mémoire correspondante est libérée.

DECLARATION : <type> <identifiant> [=<expression>] ;

int x = 10; float f;

REMARQUE: Java fait la distinction entre majuscule et minuscule

1.3. Tableaux

DECLARATION : juste le pointeur est créé

- int[] tableau_entiers;
- double tableau_reels[];
- long[][] tableau_2D;
- int tab[], i; /* un tableau et un entier */
- int[] tab, i ; /* 2 tableaux int */

CREATION : réservation réelle des cases du tableau (utilisation de new)

- tableau_entiers = new int[10];

- tableau_reels = new double[2 * a];
- tableau_2D = new long[5][4];
- tab = {1, 2, 3, 4}; /* Réservation + initialisation */
- variableTableau.length; /* permet de connaitre la taille du tableau */

EXEMPLE D'UTILISATION

```
for (int i=0; i<tableau_entiers.length; i++) {
    tableau_entiers[i] = 2 * i ; }
```

REMARQUE

Un tableau multidimensionnel est un "tableau de tableaux".

1.4. Structures

- **If … else …**

If (expression booléenne) { instructions; }
[else {instructions}]

- **Switch**

Switch (expression) {
case expr1: instructions; [break;]
…
default: instructions;}

- **For**

for (init; test; incrément) { instructions; }

Il est possible de déclarer la variable dans la boucle for.

- **While**

while (expression_booléenne) { instructions; }

do { instructions; } while (expression_booléenne);

2. Les classes

2.1. Définition de classe

Les objets qui collaborent dans une application sont souvent très nombreux. Ces objets ont une structure et un comportement très proches, sinon identiques. La notion de classe correspond à la notion de types d'objets. Les classes permettent de définir les nouveaux types et un objet correspond à une instanciation de classe. Une classe est un type qui décrit une structure (variables d'état) et un comportement (méthodes)

Fichier "Circle.java"

```
class Circle {
}
```

Une classe est une structure de données regroupant des :

- **Attributs** : données possédant un type quelconque prédéfini (entier, caractère…) ou un type Classe. C'est une variable avec une visibilité plus large liée à celle de la classe.

Fichier "Circle.java"
```
class Circle {
    double rayon;      // Attribut : rayon, type : double (type primitif)
    Point2D centre;    // Attribut : Centre, type : Point2D (classe)
}
```
Fichier "Point2D.java"
```
class Point2D {
  double x, y;
}
```

- **Méthodes** (équivalent de fonctions) : traitements applicables à l'objet ou à un de ses attributs

Fichier "Circle.java"
```
class Circle {
  Point2D centre;
  double rayon;

  void reset() {              // Méthode qui réinitialise l'attribut
    rayon = 0;
  }
}
```
Fichier "Point2D.java"
```
class Point2D {
  double x, y;

  void print(){               // Méthode qui fait un affichage
    System.out.print("["+x+","+y+"]");
  }
}
```
REMARQUES

- Une classe défini un type.
- Il est préférable que le code de la classe soit écrit dans un fichier ".java".
- Il y aura autant de fichier .class produit par le compilateur que de classes dans un fichier .java
- 'void' sert à définir un type de retour nul (sans valeur) pour la méthode.
- Les méthodes System.out.print et System.out.println affichent toutes les deux un message passé en paramètre, la différence réside dans le fait que la seconde rajoute en plus un retour à la ligne suivante.

2.2. Programme java

- Un programme est simplement un ensemble de classes.
- Un programme doit absolument contenir une classe avec une méthode main. L'exécution du programme commence par cette méthode.
- Le fichier .java qui contient la méthode 'main' doit avoir le même nom de la classe qui la contient.

```
/************************************************
*** Ce programme se contente d'afficher le message ***
*** "Premier programme Java"                      ***
************************************************/
public class PremierProgramme {
  // Définition de la méthode statique main
  static public void main(String params[]){
    System.out.println("Premier Programme Java");
```

```
   }
}
```

CONVENTIONS D'UTILISATION

- Les noms de classes commencent par une majuscule (ce sont les seuls avec les constantes) : Cercle, Objet
- Les mots contenus dans un identificateur commencent par une majuscule : uneClasse, uneMethode, uneAutreVariable
- Les constantes sont en majuscules et les mots sont séparés par le caractère souligné « _ » : UNE_CONSTANTE
- Si possible, utiliser des noms pour les classes et des verbes pour les méthodes

2.3. Instanciation

Une classe sert de moule pour la création des objets, dits **instances** de classe. On instancie un objet en appliquant l'opérateur ***new*** sur un constructeur de classe.

```
Circle c1;      // déclaration, c'est une référence sur un futur objet
c1 = new Circle();             // instanciation
Circle c2 = new Circle();      // déclaration avec initialisation
Circle c3 = null;              // équivalente à Circle c3;
Point2D p1 = new Point2D();
```

REMARQUES

- Une variable contient une référence sur un objet mais pas l'objet lui-même.
- Pour comparer deux références, on utilise la méthode 'equals()'.
- ***null*** est une valeur qui dénote l'absence d'objet et n'a pas de type.
- On ne peut pas utiliser un objet avant son instanciation.

Une instance d'une classe est créée par un des constructeurs de la classe. Une fois qu'elle est créée, l'instance :

- a son propre état interne (les valeurs des variables)
- partage le code qui détermine son comportement (les méthodes) avec les autres instances de la classe

2.3.1. Variables d'instance

Ce sont les variables définies dans une classe et à l'extérieur des méthodes sans les modificateurs spéciaux. Elles sont dites aussi champs ou attributs d'instance. Elles définissent l'état d'un objet. Elles sont créées quand un objet est créé et détruites quand l'objet est détruit

REMARQUES

- A l'intérieur d'un membre de classe, les variables d'instance sont invoquées (utilisées) simplement par leurs noms. de l'extérieur, leurs invocation est : Nom_Instance.Nom_Attribut
- Si une variable d'instance a le même nom qu'une variable ordinaire, on précède les variables d'instance par 'this.'
- Avant l'instanciation (**new**), l'attribut d'instance n'a aucune existence physique.
- Les variables d'instance dont le type est primitif sont initialisées lors de l'instanciation de l'objet. Pour celles de type classe, on doit instancier explicitement des objets ou leur affecter des instances existantes.

EXEMPLE

```
c1.rayon = 2 ;
c1.centre = p1 ;
c2.rayon = 10 ;
c2.centre = new Point2D() ;
c3 = new Circle() ;
c3.rayon = 8 ;
c3.centre = new Point2D() ;    // ou bien Point2D p2 = new Point2D() ;
c3.centre.x = 10 ;             // p2.x = 10 ; p2.y = 5 ;
c3.centre.y = 5 ;              // c3.centre = p2 ;
```

2.3.2. Méthodes d'instances et leur appel

C'est les actions (fonctions) définies sans les modificateurs spéciaux qui manipulent l'état d'un objet. Les méthodes déterminent le comportement des instances de la classe quand elles reçoivent un message (ce sont les messages qu'on peut lui envoyer : quand un objet reçoit un message, il exécute la méthode correspondante).

```
class Circle { // Classe améliorée
  Point2D centre;
  double rayon;

  void reset() {
    rayon = 0;
    centre.reset();
  }
  void Move(double x, double y) {
    centre.x += x;
    centre.y += y;
  }
  void MoveH(double x) { centre.x += x; }
  void MoveV(double y) { centre.y += y; }
  void println(){
        System.out.print("Objet Circle :\n\tcentre : ");
        centre.print();
        System.out.println("\n\trayon : "+rayon);
    }
}

class Point2D {        // Classe améliorée
  double x, y;
  void reset() {
    x = 0 ;
    y = 0 ;
  }
  void print(){
    System.out.print("["+x+","+y+"]");
  }
}
```

EXEMPLE D'UTILISATION

```
Circle c1 = new Circle();
Point2D p = new Point2D();
c1.centre = p;
c1.move (2,5);
c1.println();  //Affiche: Objet Circle : centre : [2,5]  rayon : 10
c1.reset();
c1.println();  //Affiche: Objet Circle : centre : [0,0]  rayon : 0
```

AUTRE EXEMPLE

```
class Room
{
    int size = 0;
    Room next;
    int getSize()
    {  return size; }
    void setSize(int size)
    {  this.size = size; }
    Room getNextRoom() { return next; }
    void setNextRoom(Room next) { this.next = next }
}
```

EXEMPLE D'UTILISATION

```
// On déclare une cuisine et un salon
Room kitchen = new Room();
Room living = new Room();
// la cuisine fait 10m²
kitchen.setSize(10);
// la cuisine donne dans le salon
kitchen.setNextRoom(living);
// le salon a une surface double de la cuisine
living.setSize(kitchen.getSize() * 2);
// le salon mène à la cuisine
living.setNextRoom(kitchen);
// la surface de la cuisine est la même que la pièce voisine
kitchen.setSize(kitchen.getNextRoom().getSize());
```

2.3.3. Objet Java

Un objet Java est alors défini par :

- une adresse en mémoire (identifiant de l'objet ou sa référence)
- un comportement (ou interface)
- un état interne

Le comportement est donné par des méthodes. L'état interne est donné par les valeurs des variables d'instances.

2.4. Membres de classe

2.4.1. Variables de classe

Déclarées avec le mot clé '**static'** dans une classe, mais en dehors d'une méthode. Elles sont définies pour l'ensemble du programme et sont visibles depuis toutes les méthodes

```
class Point
{
    static int NbrPoints = 0;
    int x = 0, y = 0;
    int getNbrPoints() { return NbrPoints; }
}
```

- Si un membre est déclaré "**static**" alors il est commun à toutes les instances de la classe.
- Une classe qui ne contient que des membres **static** ne sert pas à créer des objets.
- Un membre **static** existe sans instanciation et est toujours au nombre de 1 quel que soit le nombre d'instanciation.
- Si une variable de classe est initialisée dans sa déclaration, cette initialisation est exécutée une seule fois quand la classe est chargée en mémoire

EXEMPLE

```
Point p1 = new Point();
```

```
Point p2 = new Point();
Point.NbrPoints = 2;
p1.getNbrPoints() ;    // retourne 2
p2.getNbrPoints() ;    // retourne 2
```

2.4.2. Méthodes de classe

Une méthode de classe (modificateur **static** en Java) exécute une action indépendante d'une instance particulière de la classe. Une méthode de classe peut être considérée comme un message envoyé à une classe.

EXEMPLE

```
class Point
{
    static Point Origin = new Point();
    int x = 0, y = 0;
    static Point getOrigin() { return Origin; }
    static void setOrigin(Point origin) {
        Origin = origin;
    }
}
```

Pour désigner une méthode static depuis une autre classe, on la préfixe par le nom de la classe. On peut aussi la préfixer par une instance quelconque de la classe (à éviter car cela nuit à la lisibilité : on ne voit pas que la méthode est static) :

```
Point p = new Point();
p.x = 1; p.y = 2;
Point.getOrigin();     // Point(0,0)
Point.setOrigin(p);
Point.getOrigin();     // Point(1,2)
```

Une méthode de classe **ne peut utiliser que des variables de classe** et jamais des variables d'instance et ne peut appeler (invoquer) que des méthodes de classe. Une méthode d'instance peut accéder aux deux catégories de variables.

REMARQUES

- Les méthodes de classe sont recherchées statiquement (compilation).
- Les méthodes d'instance sont recherchées dynamiquement (exécution).
- Comme une méthode de classe exécute une action indépendante d'une instance particulière de la classe, elle ne peut utiliser de référence à une instance courante (this).
- La méthode main() est nécessairement static car elle est exécutée au début du programme où aucune instance n'est créée.
- La méthode main() existe indépendamment de toute classe, si elle doit utiliser des attributs ou des méthodes de la classe, il faut alors que ces champs soient eux aussi déclarés **static**, sans quoi, ils n'existent pas.

EXEMPLE

```
class Start {
   static int a = 3;
   static public void main(String argv[]){
      a += 5;
```

```
      System.out.println("a^2 = " + Square(a));
   }
   static int Square(int value){
      return value*value;
   }
}
```

CAS PARTICULIER: BLOCS D'INITIALISATION STATIC

Ils permettent d'initialiser les variables static trop complexes à initialiser dans leur déclaration :

```
class UneClasse {
    private static int[] tab = new int[25];
    static {
        for (int i = 0; i < 25; i++) {
            tab[i] = -1;
        }
    }
```

Ils sont exécutés une seule fois, quand la classe est chargée en mémoire

2.5. Contrôle d'accès

Le contrôle d'accès se fait par un qualificateur (modifieur). Il permet de limiter/étendre l'accessibilité des variables et des méthodes relativement aux blocs standards. Pour les classes, la signification est différente (abordé plus tard).

- **private** accessible uniquement depuis la même classe.
- **protected** accessible depuis la classe et ses sous-classes.
- **public** accessible depuis n'importe où.
- **final**: - classe ne peut être dérivée – méthode ne peut pas être redéfinie dans une classe dérivée – attribut ne peut pas être modifié.
- **Abstract** Pour définir une classe abstraite. La classe ne peut pas avoir d'instances.

Exemple

```
class Forme {
    protected int x = 0;
    private double r = 1.5;
    private void setRadius(double r) { this.r = r; }
    public double getRadius() { return r; }
}
```

2.6. Méthodes particulières

Deux types de méthodes servent à donner accès aux variables privées ou protégées depuis l'extérieur de la classe :

- les **accesseurs** en lecture pour lire les valeurs des variables
- les **modificateurs** ou accesseurs en écriture, pour modifier leur valeur

EXEMPLE

```
public class Employe {
    private double salaire;
    …
    public void setSalaire(double unSalaire) {
        if (unSalaire >= 0.0)
            salaire = unSalaire;
    }
    public double getSalaire() {
        return salaire;
```

```
    }
    …
}
```

Les accesseurs et les modificateurs sont définies généralement '**public'** et leurs nom commence par '**set'** ou '**get'** suivi du nom de la variable à laquelle ils donnent accès ou modifient. Ces méthodes sont utiles principalement pour les variables '**private'**

3. Constructeurs et destructeurs

3.1. Constructeurs

Une classe comporte au moins un constructeur (ajouté par défaut par le compilateur si le programmeur n'en spécifie pas). Les constructeurs sont particulièrement utilisés pour initialiser les variables d'instance d'un objet.

```
class Circle
{
  double r;
  Point2D centre;

  public Circle(double x, double y, double r) {
    centre = new Point2D() ;
    centre.x = x; centre.y = y; this.r = r;
  }
}
```

EXEMPLE D'UTILISATION

```
Circle c = new Circle(2.0, 2.0, 1.0);
```

REMARQUES

- Un constructeur n'a pas de valeur de retour.
- Un constructeur porte obligatoirement le même nom que la classe.
- Une classe peut définir plusieurs constructeurs avec des arguments différents (surcharge).
- Le constructeur est appelé par new.
- Le choix du constructeur est donné par le type et le nombre d'arguments.
- Par défaut, un constructeur sans arguments est défini par le compilateur.

EXEMPLE

```
class Circle
{
  double r;
  Point2D centre;

  public Circle(double x, double y, double r) {
    centre = new Point2D() ;
    centre.x = x; centre.y = y; thisr = r;
  }
  public Circle(double x, double y) {
    centre = new Point2D();
    centre.x = x; centre.y = y; r=0.0;
  }
  public Circle(Circle c)
  {
    centre = new Point2D();
    centre.x = c.x; centre.y = c.y; this.r = c.r;
  }
```

```
  public Circle() {
    centre = new Point2D();
    centre.x = centre.y = r = 0.0;
  }
}
```

REMARQUES

- this(...) permet de déléguer l'initialisation à un autre constructeur. Ce mécanisme permet d'écrire moins de code.
- this(...) doit désigner un constructeur qui existe dans la même classe.
- this(...) doit être la première instruction du constructeur

```
class Circle
{
  double r;
  Point2D centre;

  public Circle(double x, double y, double r) {
    centre = new Point2D() ;
    centre.x = x; centre.y = y; thisr = r;
  }
  public Circle(double x, double y) {
    this(x, y, 0.0);
  }
  public Circle(Circle c)
  {
    This(c.x; c.y; c.r);
  }
  public Circle() {
    this(0.0, 0.0, 0.0);
  }
}
```

3.2. Destructeurs

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

```
public class Garbage {
    String id;

    public Garbage(int i) {
        id = "" + i;
        System.out.println("Nouvel objet " + id + " - ");
    }

    public void finalize() {
        System.out.println("Destruction de objet " + id + " - ");
    }

    public static void main(String argv[]) {
        for(int i=0;i<100000;i++) new Garbage(i);
    }
}
```

Le ramasse miettes (garbage collector) est le responsable de la destruction des objets.

3.3. Le ramasse miettes

Un programme Java a besoin de mémoire pour pouvoir s'exécuter. L'opérateur new se charge d'allouer de la mémoire à la demande. Une conséquence évidente est que si l'on ne libère pas la

mémoire des objets devenus inutiles, on peut rapidement en manquer. Le ramasse miettes (ou Garbage Collector) se charge de repérer ces objets inutiles (inaccessible) et de libérer la mémoire qu'ils utilisent désormais inutilement. Il opère de façon totalement automatisé. A noter que par l'intermédiaire des destructeurs, il est possible de définir des actions à effectuer en cas de destructions d'objet.

4. Encapsulation

L'**Encapsulation** a deux sens:

- Regrouper des caractéristiques au sein d'une même classe. Ce principe est implicite en POO.
- Cacher certains membres d'une classe à d'autres classes.

Ainsi, l'encapsulation est le mécanisme qui consiste à déclarer comme ***private*** une large partie des caractéristiques de la classe, tous les attributs et de nombreuses méthodes. Un attribut ou une méthode sera ***private***, si l'on souhaite restreindre son accès à la seule classe dans laquelle il/elle est déclaré€. Ainsi, ses méthodes se chargent de préserver l'intégrité de tous ses objets. Il/elle sera ***public*** si son accès est possible par, ou dans, toute autre classe.

- **Abstraction de données** : la structure d'un objet n'est pas visible de l'extérieur, son interface se charge de fournir l'accès de manière contrôlée.
- **Abstraction procédurale** : L'utilisateur n'a aucun élément d'information sur le mécanisme interne mise en œuvre. Par exemple, il ne sait pas si le traitement requis a demandé l'intervention de plusieurs méthodes ou même la création d'objets temporaires, etc.

EXEMPLE:

```
class Voiture {
  private int vitesse ;

  public int changeVitesse(int nouvelleVitesse) {
    if (nouvelleVitesse >= 0) && (nouvelleVitesse <=130)
           /* intégrité assurée */
      vitesse = nouvelleVitesse ;
      return vitesse ;
    }
  }
}
```

Les méthodes **private** sont responsables de l'implémentation de la classe, alors que les **public** sont responsable de l'interface de la classe.

Il faut garder à l'esprit que l'interface est formée par les méthodes publiques. Normalement, les interfaces ne peuvent évoluer dans le temps, mais uniquement leur corps, pour que ça soit sans conséquence sur les autres classes. Cette séparation force tout programmeur d'une classe à réfléchir de façon anticipée, afin de tenir clairement détachées les méthodes qu'ils prédestinent aux autres classes de celles qui font partie de la classe qu'il programme.

```
class FeuDeSignalisation {
  private int couleur;
  public FeuDeSignalisation(int couleur) {
    if ((couleur >= 1) && (couleur <=3))
      this.couleur = couleur ;
  }
  public int getCouleur() {
```

```
    return couleur;
  }
  private void pasSetCouleur(int nouvelleCouleur) {
    System.out.println ("pas bonne couleur, la: " + nouvelleCouleur);
  }
  public void setCouleur(int nouvelleCouleur) {
    if ((nouvelleCouleur >= 1) && (nouvelleCouleur <=3))
      couleur = nouvelleCouleur ;
    else pasSetCouleur(nouvelleCouleur); // appel de la méthode privée
  }
}

public class TestPrive {
  public static void main(String[] args) {
    FeuDeSignalisation unFeu = new FeuDeSignalisation(2);
    System.out.println(unFeu.getCouleur());
    unFeu.setCouleur(5);
    System.out.println(unFeu.getCouleur());
   /*unFeu.pasSetCouleur(5); ici, on ne peut appeler cette méthode privée */
  }
}
```

Dans les versions canoniques du paradigme objet, les services d'un objet ne sont invocables qu'au travers de messages composés de :

- Un nom
- Une liste de paramètres en entrée
- Une liste de paramètres en sortie

Une classe sans spécification de modifieur est visible **seulement** par toutes les autres classes du module (paquetage) où elle est définie.

5. Classes imbriquées

Une classe imbriquée est définie dans le contexte d'une classe englobante. Elle peut accéder aux variables statiques de la classe englobante. Le nom d'une classe imbriquée est préfixé du nom de la classe englobante. Une classe imbriquée peut être déclarée publique, protégée, privée, …

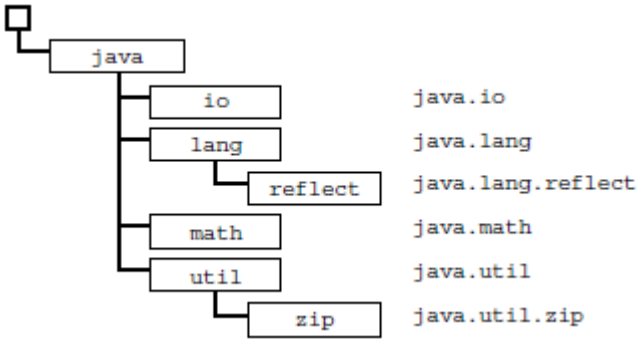
```
public class ApplicationClasses {
   abstract class ApplicationClasse1 { ... }
   public class ApplicationClasse2 { ... }
   class ApplicationClasse3 { ... }
}

class AppliTestClasses {
   ApplicationClasses.ApplicationClasse1 a1 ;
   ApplicationClasses.ApplicationClasse2 a2 ;
   ApplicationClasses.ApplicationClasse3 a3 ;
}
```

6. Paquetages (packages)

6.1. Paquetages standards

Dans Java, les classes sont regroupées en paquetages, selon leurs domaines d'action. Un paquetage donne une dimension supplémentaire pour l'encapsulation. Partant de cette idée, les concepteurs du langage ont développés une série de packages permettant d'utiliser l'ensemble des ressources de la

machine (c'est l'API de Java). Chacun de ces packages ayant un domaine d'action bien déterminé, comme, par exemple, les connexions réseaux, les composants graphiques, ... Par exemple:



- **java.io**: Regroupe différents types de classes gérant les entrées/sorties.
- **java.lang**: Classes Java de base. e.g., Object, String, System, Math, Classes "Wrapper": Integer, Long, Boolean, Byte, Number, …
  - c'est un paquetage de base - toujours importé quel que soit la machine virtuelle.
  - contient les classes décrivant les types primitifs, le système, …
- **java.util**: Différentes structures de données utiles. e.g., Vector, Hashtable, …

```
class Exemple {
   java.util.Vector v = new java.util.Vector();
   java.io.File f = new java.io.File();
   ...
}
```

REMARQUES

- Le nom complet d'une classe est le nom de la classe préfixé par le nom du paquetage dans lequel elle se trouve.
- **'import'** permet de rendre une classe directement accessible sans utiliser son nom complet.
- Les clauses **'import'** sont placées au début du fichier qui peut contenir autant de clauses import que nécessaires et qui sont limitées au fichier.
- Lorsqu'une clause **'import'** spécifie le nom d'un paquetage suivi de **.***, toutes les classes du paquetage sont automatiquement importées.

```
import java.util.Vector;
import java.io.*;
class Exemple
{
   Vector v = new Vector();
   File f = new File();
   ...
}
```

6.2. Création de paquetages

Pour qu'une classe soit définie dans un paquetage particulier, la clause '**package'** suivie du nom du paquetage doit figurer au début du fichier contenant la classe avant d'éventuelles clauses **'import'**. Notons que seules les classes publiques sont exportées par un paquetage.

Il est possible de configurer le compilateur pour placer les fichiers **.class** (byte code de la JVM) générées à un emplacement spécifique. De plus, les concepteurs du langage y ont introduit la notion d'archive Java (d'extension .jar), au format ZIP contenant les fichiers d'extension **.class**, et donc les

packages. Les archives Java permettent de manipuler un seul fichier au lieu de tous ceux constituant le programme. Pour cela, on utilise l'utilitaire **'jar'**

```
package java.shape2D;
class Circle extends Shape {
  ...
}

import java.awt.Button;   // On importe la classe Button
import java.shape2D.*;    // On importe toutes les classes

class Exemple {
  static void main(String argv[]){
    Circle c1=new Circle();
    ...
  }
}
```

6.3. Quelques classes particulières

6.3.1. Classe Object

Toutes les classes Java descendent d'***Object***. Implicitement, une classe descend d'Object. Ces méthodes servent à décrire un objet et devraient être surchargées par les descendants de la classe:

- **toString**() rend une version textuelle de l'objet: appelé implicitement si nécessaire.
- **clone**() fait une copie de l'objet (par défaut "shallowcopy").
- **equals**() compare avec un autre objet rend true si égal.

6.3.2. Classe String

Représente une chaîne de caractères qui est une suite ordonnée de caractères (Unicode). Elle a une longueur arbitraire (ce n'est pas un tableau).

Elle est créée implicitement lorsqu'une chaine de caractères se trouve dans le code source. C'est un objet constant qui ne peut être modifié. Toutes les opérations créent de nouvelles instances. Ses méthodes :

- **length**() rend la longueur de la chaîne: "toto".length() -> 4
- **concat**() construit une nouvelle chaîne qui est la concaténation de deux autres. Implicitement appellée par l'opérateur + : "toto".concat("ro") -> "totoro"
- **toLowerCase**() construit une nouvelle chaîne en minuscule.
- **subString**() construit une sous-chaîne.
- **startsWith**() vérifie si une chaîne commence avec une autre.
- **indexOf**() cherche la position dans une chaîne d'une sous chaîne / d'un caractère.

6.3.3. Classes "Wrapper"

Elles enrobent un type primitif dans une classe. Elles offrent un aspect objet aux types scalaires. Il existe une classe par type primitif: Integer (int), Boolean (boolean) ...

Toutes les classes "Wrapper" de types numériques descendent de la classe Number. Elles offrent des méthodes auxiliaires (conversion, tests, etc.): String s = "1291"; int i = Integer.parseInt(s);

6.3.4. Classes "Math"

C'est une classe "vide". Elle ne contient que des méthodes et des variables statiques, donc elle ne peut être instanciée. Elle contient les fonctions mathématiques sin, cos, abs, max, log, etc. et les constantes mathématiques E, PI

6.3.5. Classes "System"

Elle représente le système. Elle permet de communiquer avec l'extérieur de la machine virtuelle. Elle gère les sorties et entrées standard: System.in, System.out, System.err, System.exit(). Elle permet aussi le contrôle de la machine virtuelle.

Chapitre 3

Héritage et Polymorphisme

Chapitre 3

Héritage et Polymorphisme

1. Généralités

L'héritage est un principe fondamental du paradigme orienté objet. Il est chargé de traduire le principe naturel de Généralisation / Spécialisation. Pour signifier qu'une classe fille hérite d'une classe mère, on utilise le mot clé extends.

2. Héritage

2.1. Principe

L'héritage consiste à définir une classe à partir d'une autre. La classe définie à partir d'une autre sera nommée classe fille. Celle qui sert à définir une classe fille sera nommée classe mère. On dit alors que la classe fille hérite (ou dérive) de la classe mère.

La classe mère est associée au concept le plus général, nous l'appellerons *classe de base* ou *classe mère* ou *super - classe*. Pour chaque concept spécialisé, on dérive une classe du concept de base. La nouvelle classe est dite *classe dérivée* ou *classe fille* ou *sous-classe*. Ainsi, la classe dérivée est une version spécialisée de sa classe de base.

Une fois que l'héritage est spécifié, la classe fille possède aussi l'ensemble des attributs et des méthodes de sa classe mère. Une sous-classe désigne sa superclasse au moyen du mot-clé *extends*. Notons que seul l'héritage **simple** est possible en Java.

En java, toutes les classes sont dérivées de la classe Object. La classe Object est la classe de base de toutes les autres. C'est la seule classe de Java qui ne possède pas de classe mère. Tous les objets en Java, quelle que soit leur classe, sont du type Object. Cela implique que tous les objets possèdent déjà à leur naissance un certain nombre d'attributs et de méthodes dérivées d'Object.

Dans la déclaration d'une classe, si la clause *extends* n'est pas présente, la surclasse immédiatement supérieure est donc Object.

EXEMPLE

```
class A {
   int x = 0;
   int getX() { return x; }
}

class B extends A {
   int x = 1;
   int getXinB() { return x; }
   int getThisX() { return this.x; }
   int getSuperX() { return super.x; }
}

// Utilisation
B b = new B();
b.getX();       // 0
```

```
b.getXinB();    // 1
b.getThisX();   // 1
b.getSuperX(); // 0
```

2.2. Constructeur de la classe mère

On n'hérite en aucun cas des constructeurs de la classe mère. Si on ne spécifie pas explicitement un constructeur particulier, on ne pourra pas l'utiliser, et ce même s'il en existe un défini dans la classe mère. C'est super(...) qui permet d'invoquer un constructeur de la superclasse.

L'instruction super( ) doit être obligatoirement la première instruction des constructeurs de la classe fille. Tout constructeur d'une classe fille appelle forcément l'un des constructeurs de la classe mère : si cet appel n'est pas explicite, l'appel du constructeur par défaut (sans paramètre) est effectué implicitement par Java.

EXEMPLE

```
class ColorCircle extends Circle {
   int color;
   public ColorCircle(double x, double y, double r, int color) {
       super(x, y, z);
       this.color = color;
   }
   public ColorCircle(ColorCircle c) {
       super(c);
       this.color = c.color;
   }
   public afficheToi() {
       super.afficheToi();
       System.out.println("Couleur : " + couleur) ;
   }
}
```

2.3. Membres protégés

Les sous-classes n'ont pas accès aux membres private de leur classe mère. Si un attribut de la classe mère est déclaré private, cet attribut n'est pas accessible par les sous-classes.

Remarque: les membres privés de la classe mère sont bien hérités par les classes filles, mais ceux-ci ne sont accessibles qu'à travers les accesseurs et/ou les méthodes de la classe mère qui les utilisent.

Si on veut qu'un attribut ou une méthode soit encapsulé pour l'extérieur mais qu'il soit accessible par les sous-classes, il faut le déclarer protected à la place de private.

2.4. Référence

Le transtypage (conversion de type ou cast en anglais) consiste à modifier le type d'une variable ou d'une expression. Le transtypage permet de restreindre le type d'un objet.

- **Transtypage implicite**: utilisé lorsque le type cible est dans un domaine plus grand que le type d'origine : affecter directement un int à un double ()
- **Transtypage explicite**: domaine de valeurs du type cible plus restreint. Ex: double d; int i; i = (int) d;

Il est possible de convertir un objet d'une classe en un objet d'une autre classe si les classes ont un lien d'héritage (***ce sont les références aux objets et non les objets eux-mêmes qui sont transtypés***).

- Le transtypage est implicite lorsqu'on utilise une référence de la classe mère pour désigner un objet d'une classe dérivée. Cela entraîne une perte de précision puisqu'on ne pourra pas utiliser la référence de la classe mère pour appeler une méthode spécifique de la classe fille.

EXEMPLE

```
Personne p = new Personne ("Malek") ;
Employe e = new Employe("Nadir", 10000);
p = e ; // ok : Employe est une sous classe de Personne
Object obj;
obj = e ; // ok : Employe hérite de Personne qui elle-même hérite de Object
Personne[] tab = new Personne[2];
tab[0] = new Personne("Malek");
tab[1] = new Employe("Nadir", 10000);
```

- Le transtypage dans le sens « mère pointée par une fille » doit être explicite et n'est pas toujours possible.

Pour réaliser un cast explicite, on met entre parenthèses le nom du type dans lequel on veut convertir suivi du nom de la variable (ou de l'expression entre parenthèses) qu'on veut transtyper. Ainsi, une variable d'un type donné ne peut contenir que des objets du même type ou de l'un de ses sous-types (polymorphisme). Dans l'exemple suivant: le type de la variable 'adel' est convertit en type Secretary. 'mourad' ne peut être convertit en Secretary puisqu'elle contient un objet de type Engineer. (Engineer n'est pas une sous-classe de Secretary)

EXEMPLE

```
class Employee { ... }
class Secretary extends Employee { ... }
class Engineer extends Employee { ... }

Employee adel = new Secretary("Adel");
Secretary nabil = new Secretary("Nabil");
adel = nabil; // OK
nabil = adel; // ERREUR à la compilation

Employee mourad = new Engineer("Mourad");
Secretary amar;
amar = adel; // ERREUR à la compilation
amar = (Secretary) adel;
amar = (Secretary) mourad; // ERREUR à l'exécution
```

AUTRE EXEMPLE

```
class F {
   static int classF() { return 1; }
   int instanceF() { return 2; }
   static int classCall() { return classF(); }
   int instanceCall() { return this.instanceF(); }
}
class B extends F {
   static int classF() { return 10; }
   int instanceF() { return 20; }
}

B b = new B();
b.instanceCall();      // 20
F.classCall();         // 1
```

2.5. Avantages de l'héritage

Le premier point important est que l'héritage supprime les redondances dans le code. En effet, une fois la hiérarchie de classes bien établie, on localise en un point unique les sections de code (celles-ci restantes à tous moment accessibles grâce au mot clé **super**).
Aussi, si la hiérarchie des classes est bien pensée, on peut très facilement rajouter une classe, et ce à moindre coût, étant donné que l'on peut réutiliser le code des classes parentes.
Finalement, si un comportement n'a pas encore été modélisé dans une classe donnée, et qu'il faut le rajouter, il sera directement utilisable dans l'ensemble des sous-classes de celle considérée.

3. Polymorphisme

3.1. Définition

Le polymorphisme est un principe sur lequel repose le paradigme objet. Comme son nom l'indique, le polymorphisme permet à une méthode d'adopter plusieurs formes sur des classes différentes. Nous distinguons:

- le polymorphisme statique ou **la surcharge de méthode**
- le polymorphisme dynamique ou **la redéfinition de méthode** ou encore la **surcharge héritée**.

3.2. Surcharge

La surcharge de méthode consiste en le fait qu'une classe peut disposer de **plusieurs méthodes ayant le même nom**, mais avec des paramètres formels différents ou éventuellement un type de retour différent.

EXEMPLE:

```
class Un
{
   int a;
   public Un ()
   { a = 100; }
   public Un (int b )
   { a = b; }
   public Un (float b )
   { a = (int)b; }
   void f ()
   { a *=10;  }
   void f (int x)
   { a +=10*x; }
   int f (int x, char y)
   { a = x+(int)y;
   return a;  }
}
```

3.3. Redéfinition de méthode

La redéfinition de méthode (ou polymorphisme dynamique) est mise en œuvre lors de l'héritage d'une classe mère vers une classe fille dans le cas d'une méthode ayant la même signature dans les deux classes. En java aucun mot clef n'est nécessaire ni pour la surcharge ni pour la redéfinition, c'est le compilateur qui analyse la syntaxe afin de se rendre compte en fonction des signatures s'il s'agit de redéfinition ou de surcharge.

La redéfinition d'une méthode dans une classe fille cache la méthode d'origine de la classe mère. Pour utiliser la méthode redéfinie de la classe mère et non celle qui a été implémentée dans la classe fille, on utilise le mot-clé *super* suivi du nom de la méthode. Par exemple, *super.machine( )* fait appel à la méthode machine( ) implémentée dans la classe mère et non à l'implémentation de la classe fille.

EXEMPLES:

```
class ClasseMere
{
  int x = 10;
  void f ( int a)
  { x += a; }
  void g ( int a, int b)
  { x += a*b; }
}
class ClasseFille extends ClasseMere
{
  int y = 20;
  void f ( int a)      //redéfinition
  { x +=a; }
  void g (char b)      //surcharge et redéfinition de g
  { ..... }
}

class Ville {
  private String nom;
  protected int nbHab;
  public Ville(String leNom) {
    nom = leNom.toUpperCase( ); // tous les noms de ville seront en
                                // majuscule
    nbHab = -1; //-1 signifie que le nombre d'habitant est inconnu
  }
  public Ville (String leNom, int leNbHab) {
    nom = leNom.toUpperCase( );
    if (leNbHab < 0) {
       System.out.println("Un nombre d'habitant doit être positif.");
       nbHab = -1; }
    else
       nbHab = leNbHab;
  }
  public String getNom() {
    return nom; // pas d'accesseur en écriture pour le nom il est
  }             // impossible de changer le nom d'une ville
  public int getNbHab( ) {
    return nbHab; }
  public void setNbHab(int nvNbHab) {
    if (nvnbHab < 0)
        System.out.println("Un nombre d'habitant doit être positif. La
              modification n'a pas été prise en compte");
    else
        nbHab = nvNbHab;
  }
  public String presenteToi() {
    String presente = "Ville "+ nom +" nombre d'habitants ";
    if (nbHab = = -1)
      presente = presente + "inconnu";
    else presente = presente + " = " + nbHab;
    return presente;
  }
}
```

```
class Capitale extends Ville
{
  private String pays;
  public Capitale(String leNom, String lePays) {
    super(leNom); // appel du constructeur de Ville.
                  // nbHab est initialisé à -1 par ce constructeur
    pays = lePays;
  }
  public Capitale(String leNom, String lePays, int leNbHab) {
    super(leNom, leNbHab) ;
    pays = lePays;
  }
  //accesseurs supplémentaires
  public String getPays() {
    return pays;
  }
  public void setPays(String nomPays) {
    pays = nomPays;
  }

  public String presenteToi(){// méthode presenteToi() redéfinie
    String presente = super.presenteToi;( )
    presente = presente + " Capitale de "+ pays;
    return presente;
  }
}

// Classe de test
public class testVille {
  public static void main(String args[]) {
    Ville v1 = new Ville("OEB", 60000) ;
    Ville v2 = new Ville("Cne") ;
    Capitale c1 = new Capitale("Alger", "Algérie", 5000000) ;
    Capitale c2 = new Capitale("Ouagadougou", "Burkina-Faso") ;
    System.out.println(v1.presenteToi()) ;
    System.out.println(v2.presenteToi()) ;
    System.out.println(c1.presenteToi()) ;
    System.out.println(c2.presenteToi()) ;
  }
}
```

3.4. Sélection de la méthode à exécuter

C'est le compilateur Java qui fait tout le travail. Prenons un objet obj de classe Classe1, lorsque le compilateur Java trouve une instruction du genre "obj.**method1**(paramètres effectifs);", il cherche dans l'ordre suivant :

- Y-a-t-il dans Classe1, une méthode qui se nomme method1 ayant une signature identique aux paramètres effectifs ?
- si oui c'est la méthode ayant cette signature qui est appelée,
- si non le compilateur remonte dans la hiérarchie des classes mères de Classe1 en posant la même question récursivement jusqu'à ce qu'il termine sur la classe Object.
- Si aucune méthode ayant cette signature n'est trouvée il signale une erreur.

Cependant, le choix du code à exécuter (pour une méthode polymorphe définie dans la classe mère et fille) ne se fait pas statiquement à la compilation mais dynamiquement à l'exécution.

EXEMPLE

```
Vector maListe = new Vector( );
```

```
Circle c1 = new Circle (2, 2, 4) ;
ColorCircle c2 = new ColorCircle (2, 2, 4, 20) ;
maListe.addElement(c1);        // transtypage implicite de Object (type du
maListe.addElement(c2);        // paramètre) en Circle et ColorCircle
for (int i = 0; i<2 ; i++)
   (Circle)(maListe.elementAt(i)).afficheToi() ;
        // C'est la méthode afficheToi( ) de ColorCircle qui est exécutée
        // pour le 2ème élément, car le choix du code d'une méthode se fait
        // selon le type réel de l'objet et non selon le type de la référence
        // qui le désigne.
```

4. Classes abstraites

Les classes abstraites permettent de créer des classes génériques expliquant certains comportements sans les implémenter et fournissant une implémentation commune de certains autres comportements pour l'héritage de classes. Les classes abstraites sont un outil intéressant pour le polymorphisme.

- Une classe abstraite est une classe qui ne peut pas être instanciée
- Une classe abstraite peut contenir des méthodes déjà implémentées ou non.
- Une classe abstraite est héritable .
- On peut construire une hiérarchie de classes abstraites .
- Pour pouvoir construire un objet à partir d'une classe abstraite, il faut dériver une classe non abstraite en une classe implémentant toutes les méthodes non implémentées .

Une méthode déclarée dans une classe, non implémentée dans cette classe, mais juste définie par la déclaration de sa signature, est dénommée méthode abstraite. L'implémentation d'une méthode abstraite est déléguée à une classe dérivée.

Si une classe contient au moins une méthode **abstract**, elle doit impérativement être déclarée en classe **abstract** elle-même. Conséquence, une classe dérivée qui redéfinit toutes les méthodes **abstract** de la classe mère sauf une (ou plus d'une) ne peut pas être instanciée et suit la même règle que la classe mère : elle doit être déclarée en **abstract.**

EXEMPLE :

```
public abstract class A {
     public abstract void nefaitRien() ;  // pas de corps de méthode
}
public class B extends A {
     public void neFaitRien() {
        System.out.println("Je ne fais rien !") ;
     }

}
```

On peut énoncer deux principes pour justifier la création d'une classe abstraite :

- la classe que l'on écrit va être étendue par plusieurs classes ;
- ces différentes classes vont être utilisées par des méthodes communes, et seront déclarées par le nom de la super classe abstraite.

Si ces deux conditions ne sont pas réunies, alors écrire une classe abstraite ne présente aucun intérêt.

EXEMPLE:

```
abstract class Etre_Vivant {
    abstract void SeDeplacer( );
}
```

```
class Serpent extends Etre_Vivant {
    void SeDeplacer( ) {    //.....en rampant
    }
}
class Oiseau extends Etre_Vivant {
    void SeDeplacer( ) {    //.....en volant
    }
}
class Homme extends Etre_Vivant {
    void SeDeplacer( ) {    //.....en marchant
    }
}
```

Chapitre 4

Interface et implémentation

Chapitre 4

Interface et Implémentation

1. Introduction

Java n'autorise pas l'héritage multiple, une classe ne peut hériter que d'une seule autre classe à la fois. Cependant, Java introduit le mécanisme d'interfaces qui permet d'avoir certaines facilités de l'héritage multiple.

2. Concepts

Une interface définit un comportement (d'une classe) qui doit être implémentée par une classe, sans implémenter ce comportement. C'est une classe spéciale qui n'a que des méthodes qui n'ont pas de corps, seulement une signature. Elle doit contenir des méthodes non implémentées.

```
interface Nom{
    METHODES
}
```

Exemple

```
interface Vehicule  {
    void Demarrer( );
    void RepartirPassagers( );
    void PeriodiciteMaintenance( );
}
```

Important : Les méthodes de l'interface sont abstraites et publics par définition.

Les interfaces ressemblent aux classes abstraites. C'est un ensemble de méthodes abstraites et de constantes.

Les classes abstraites et les interfaces se différencient principalement par le fait qu'une classe peut implémenter un nombre quelconque d'interfaces, alors qu'une classe abstraite ne peut hériter que d'une seule classe abstraite.

```
class MyClass extends MotherClass implements Interface1, Interface2 {…}
```

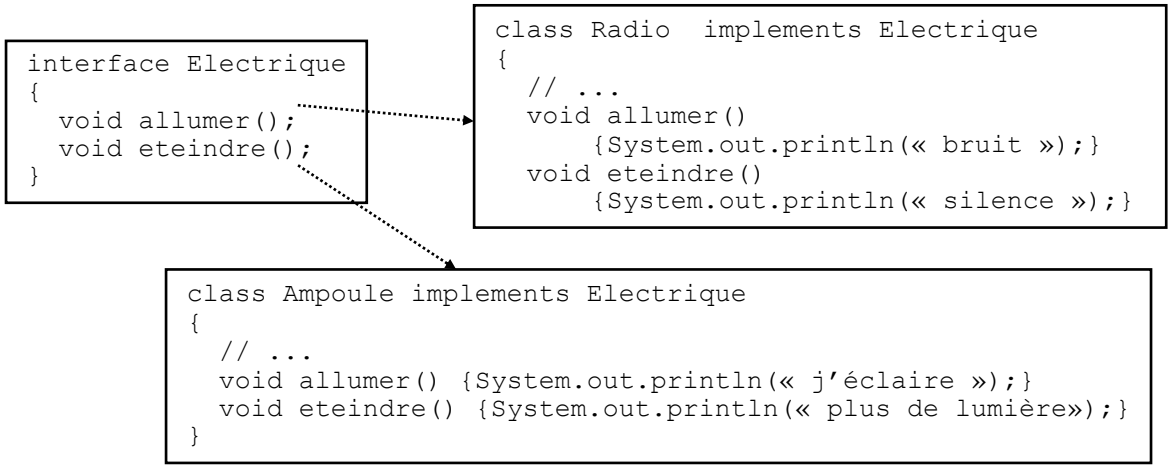
On peut construire une hiérarchie d'interfaces. Seulement, on n'hérite pas d'une interface, mais on l'implémente.

Il est possible de déclarer des variables dont le type est une interface, mais on ne peut pas instancier une interface. Pour pouvoir construire un objet à partir d'une interface, il faut définir une classe non abstraite implémentant toutes les méthodes de l'interface.

Important : Les objets de type classe clA peuvent être transtypés et référencés par des variables d'interface IntfA si la classe clA implémente l'interface IntfA.

3. Importance des interfaces

Pour définir qu'une certaine catégorie de classes doit implémenter un ensemble de méthodes, on peut regrouper les déclarations de ces méthodes dans une interface. Le but est de décrire le fait que de telles classes pourront ainsi être manipulées de manière identique.




```
// On peut écrire
Ampoule monAmpoule = new Ampoule();
Radio maRadio = new Radio();
Electrique c;
Boolean sombre;

// …
if(sombre)
   c = monAmpoule;
else
   c = maRadio;

c.allumer();
…
c.eteindre();
// …
```

REMARQUES

- La décision de créer une fonctionnalité en tant qu'interface ou en tant que classe abstraite peut parfois s'avérer difficile. Cependant, on risque moins de faire fausse route en concevant des interfaces qu'en créant des arborescences d'héritage.
- Si l'on projette de créer plusieurs versions d'un composant, on opte pour une classe abstraie. Si la fonctionnalité que l'on crée peut être utile à de nombreux objets différents, faire appel à une interface
- Une classe qui implémente une interface doit implémenter toutes les méthodes de l'interface (ou être abstraite)
- Toute classe qui implémente une interface est sous-type de cette interface. On peut utiliser : if (o instanceOf monInterface) {…}

4. Exemples d'interfaces

Exemple 1

```
interface Vehicule{
	void Demarrer( );
	void RépartirPassager( );
	void PériodicitéMaintenance( );
}
abstract class Terrestre implements Vehicule {
	public void RépartirPassager( ){…};
	public void PériodicitéMaintenance( ){…};
}
class Voiture extends Terrestre {
	public void Demarrer( ){…};
}
abstract class Marin implements Vehicule {
	public void RépartirPassager( ){…};
	public void PériodicitéMaintenance( ){…};
}
class Voilier extends Marin {
	public void Demarrer( ){…};
}
class Croiseur extends Marin {
	public void Demarrer( ){…};
}
```

Exemple 2

```
public interface Vehicule {
	 void rouler();
	 void freiner();
}

public class Velo implements Vehicule {
	 private String marque;
	 private int rayonRoue;
	 public Velo(String marque, int rayonRoue)
	 {	this.marque = marque;
		this.rayonRoue = rayonRoue;
	 }
	 public void rouler() {
		// la manière dont le vélo roule
	 {
	 public void freiner() {
		// la manière dont le vélo freine
	 {
// Autres méthodes propres à Velo
}

public class Auto implements Vehicule {
	 private String marque;
	 private int poids;
	 public Auto(String marque, int poids)
	 {	this.marque = marque;
		this.poids = poids;
	 }
	 public void rouler () {
		// la manière dont l'auto roule
	 {
	 public void freiner () {
		// la manière dont l'auto freine
	 {
```

```
// Autres méthodes propres à Auto.
}
```

Si, par exemple, nous avions une classe Personne possédant une méthode conduire (Vehicule v), on peut alors écrire :

```
Personne p = new Personne();
p.conduire(new Velo()); //comme la méthode attend un Vehicule en argument,
               // on peut passer tout objet implémentant cette interface.
p.conduire(new Auto()); //idem
```

Chapitre 5

Interface graphique et Applet

Chapitre 5

Interface Graphique et Applet

1. Composants, gestionnaire d'affichage

Les interfaces graphiques assurent le dialogue entre les utilisateurs et une application.

Dans un premier temps, Java proposait l'API AWT pour créer des interfaces graphiques, puis une nouvelle API nommée Swing permettant de faire certaines opérations plus simplement, mais réutilisant les concepts de base de awt fut proposée. Ces deux API peuvent être utilisées pour développer des applications ou des applets. Face aux problèmes de performance de Swing, IBM a créé sa propre bibliothèque nommée SWT utilisée pour développer l'outil Eclipse.

Les applications utilisent maintenant plutôt swing, mais pour comprendre les concepts de base en peu de temps, awt est plus commode.

1.1. API AWT

On distingue deux grandes parties dans la création d'une GUI:

- la définition de tout ce qui peut être affiché: fenêtre, dessins, bouton, barre d'outils, menu déroulant, etc.
- la gestion des évènements: interagir avec l'utilisateur (souris, clavier, etc.)

Les classes du toolkit AWT (Abstract Windows Toolkit) permettent d'écrire des interfaces graphiques indépendantes du système d'exploitation sur lequel elles vont fonctionner. Cette librairie utilise le système graphique de la plate-forme d'exécution (Windows, MacOS, X-Window) pour afficher les objets graphiques. Le toolkit contient des classes décrivant les composants graphiques, les polices, les couleurs et les images.

Le diagramme ci-dessous définit une vue partielle de la hiérarchie des classes (les relations d'héritage). Les deux classes principales d'AWT sont Component et Container. Chaque type d'objet de l'interface graphique est une classe dérivée de Component. La classe Container, qui hérite de Component est capable de contenir d'autres objets graphiques (tout objet dérivant de Component).

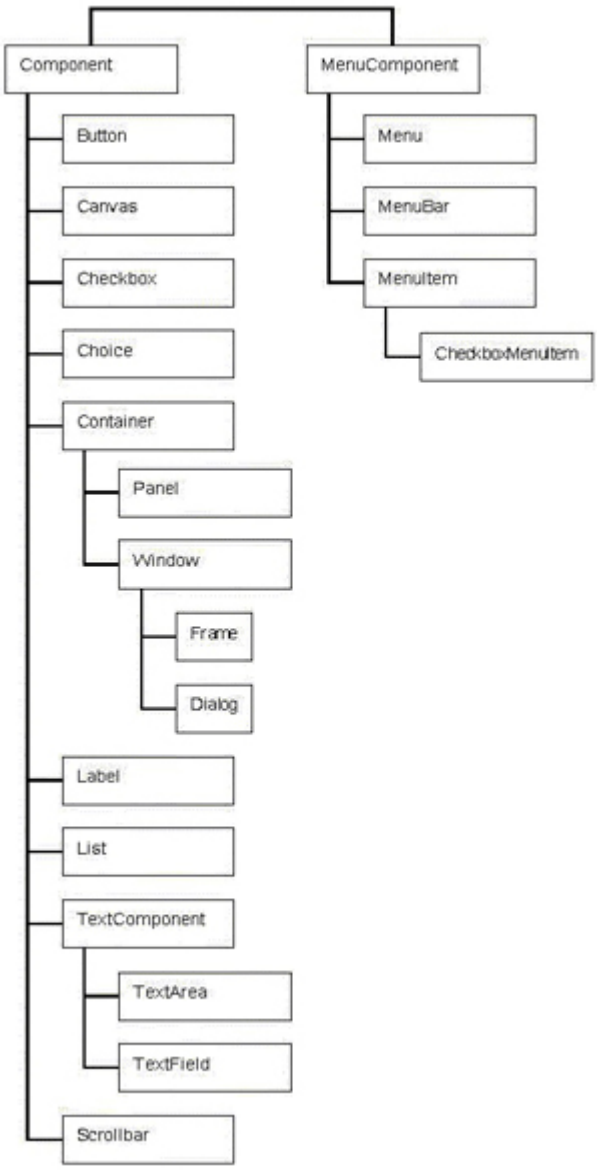
1.2. Utilisation de AWT

Fenêtres

Pour ouvrir une fenêtre on commence par créer une classe qui hérite de la classe Frame (fenêtre générique) et de ses méthodes :

```
public class Fenetre extends Frame
{
  public Fenetre(){
    setTitle("Fenêtre");
    setSize(500,700);
    setVisible(true);
  }
}
class Essai{
  public static void main(String [] arg){
```

```
    Fenetre f = new Fenetre();
  }
}
```



Contenant et environnement graphique

L'objet fenêtre en particulier et tout 'composant graphique contenant' en général a deux qualités:

- Il dispose d'un environnement graphique : objet qui gère l'affichage
  L'environnement graphique d'un composant est un objet ***g*** de la classe java.awt.Graphics. Cet objet dispose de méthodes graphiques variées telles que drawLine(), drawString(), drawImage(), setColor(), fillRectangle() utilisant un système de coordonnées cartésiennes (origine en haut à gauche)

  ```
  void monLogo(Graphics g, int x, int y){
    g.drawOval(x,y,10,10);
    g.drawRect(x+5,y+10,10,20);
  }
  ```

- Il peut contenir d'autres composants graphiques (applets, menus, boutons, etc.)

Chaque composant hérite les méthodes paint(Graphics g) qui par défaut ne fait rien et repaint() de la classe *abstraite* java.awt.Component (qui force l'utilisation de la méthode paint()). La méthode paint()

est appelée automatiquement avec l'environnement graphique de la fenêtre lorsque le '*window manager*' décide qu'il faut la rafraichir, ou lorsqu'on appelle explicitement repaint(). C'est pourquoi, pour dessiner dans une fenêtre, il faut redéfinir sa méthode paint() :

```
public class Fenetre extends Frame
{
  public Fenetre(){
    setTitle("Fenêtre");
    setSize(500,700);
    setVisible(true);
  }
  public void paint(Graphics g){
    monLogo(g,50,50);
    g.drawString("Très bien !",50,100);
  }
}
```

Les sous-classes de Component qui nous intéressent à ce niveau sont:

- **Canvas** : Un objet de cette classe n'a que son environnement graphique, pas de sous-composants. Il peut être inclus dans un contenant.
- **Container** : Les objets de cette classe peuvent contenir des sous-composants. Les sous-classes de cette classe sont:
  - **Panel** : les contenants qui peuvent être eux-mêmes inclus dans un autre contenant.
  - **Window** : les contenants qui interagissent directement avec le window manager. Ils ne peuvent pas être inclus dans un autre contenant
  - **Frame** : les vraies fenêtres
  - **Dialog** : les boites de dialogue

La méthode pour ajouter un composant à un conteneur est add() de la classe Container.

EXEMPLES:

import java.awt.*;

```
Label la = new Label( );

la.setText("une etiquette",
               Label.RIGHT);

la.setText("nouveau texte");

la.setAlignment(Label.LEFT);
```

```
class TestList {

  static public void main (String arg
                           []) {

    Frame frame = new Frame("Une
                           liste");
```

```
public class MonCanvas extends Canvas
{

  public void paint(Graphics g) {

    g.setColor(Color.black);

    g.fillRect(10, 10, 100,50);

    g.setColor(Color.green);

    g.fillOval(40, 40, 10,10); } }
```

```
Choice maCombo = new Choice();

maCombo.addItem("element 1");

maCombo.select(0);

maCombo.select("element 1");

int n;
```

```
    List list = new List(5,true);       n = maCombo.countItems();

    list.add("element 0");              String c = new String( );

    list.add("element 1");              c = maCombo.getItem(n-1);

    list.add("element 2");              c = maCombo.getSelectedItem();

    frame.add(List);                    n = maCombo.getSelectedIndex();

    frame.show(); } }

Panel p = new Panel();

Button b1 = new button(" Premier ");

p.add(b1);

Button b2;

p.add(b2 = new Button("Deuxième"));

p.add(new Button("Troisième"));
```

2. Mise en page

Lorsque l'on veut exploiter des composants prédéfinis par les classes java sans passer par la méthode paint(), il est possible de faire appel à la mise en page offerte également dans AWT.

2.1. Dimensionnement des composants

En principe, il est automatique grâce au LayoutManager. Pour donner à un composant une taille donnée, il faut redéfinir la méthode getPreferedSize() de la classe Component.

```
import java.awt.*;
public class MonBouton extends Button {
  public Dimension getPreferredSize() {
    return new Dimension(800, 250); } }
```

La méthode getPreferedSize() indique la taille souhaitée mais pas celle imposée. En fonction du Layout Manager, le composant pourra ou non imposer sa taille.

Trois méthodes de la classe Component permettent de positionner et dimensionner des composants:

- `setBounds(int x, int y, int largeur, int hauteur)`
- `setLocation(int x , int y)`
- `setSize(int largeur, int hauteur)`

2.2. Positionnement des composants

Lorsqu'on intègre un composant graphique dans un conteneur, il n'est pas nécessaire de préciser son emplacement car il est déterminé de façon automatique (mise en forme dynamique). On peut influencer cette mise en page en utilisant un gestionnaire de mise en page (*Layout Manager*) qui définit la position de chaque composant inséré.

Chaque *Layout manager* implémente l'interface java.awt.LayoutManager.

Il est possible d'utiliser plusieurs gestionnaires de mise en forme pour définir la présentation des composants. Par défaut, c'est la classe FlowLayout qui est utilisée pour la classe Panel et la classe BorderLayout pour Frame et Dialog.

Pour affecter une nouvelle mise en page, il faut utiliser la méthode setLayout() de la classe Container.

```
Panel p = new Panel();
GridLayout gl = new GridLayout(5,5);
p.setLayout(gl);
```

Les *Layout manager* ont 3 avantages:

- l'aménagement des composants graphiques leur est délégué (il est inutile d'utiliser les coordonnées absolues)
- en cas de redimensionnement de la fenêtre, les contrôles sont automatiquement agrandis ou réduits
- ils permettent une indépendance vis à vis des plates-formes.

Pour créer un espace entre les composants et le bord de leur conteneur, il faut redéfinir la méthode getInsets() d'un conteneur : cette méthode est héritée de la classe Container.

```
public Insets getInsets() {
    Insets normal = super.getInsets();
    return new Insets(normal.top+10, normal.left+10, normal.bottom+10,
           normal.right+10);}
```

Cet exemple permet de laisser 10 pixels en plus entre chaque bords du conteneur.

2.3. Utilisation des Layout

- La classe FlowLayout : place les composants ligne par ligne de gauche à droite. Chaque ligne est centrée par défaut.
- La classe BorderLayout : la disposition des composants est commandée par une mise en page en bordure qui découpe la surface en cinq zones : North, South, East, West, Center.
- La classe CardLayout : Ce layout manager aide à construire des boîtes de dialogue composées de plusieurs onglets.
- La classe GridLayout : Ce Layout Manager établit un quadrillage invisible. Les composants sont organisés en lignes et en colonnes. Les éléments insérés dans la grille ont tous la même taille.

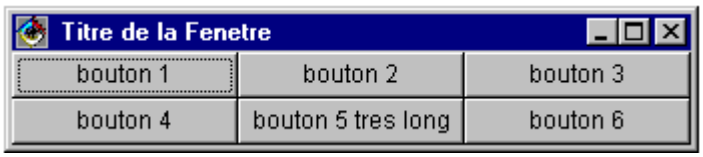
EXEMPLE 1

```
import java.awt.*;
public class MaFrame extends Frame {
  public MaFrame() {
    super();
    setTitle(" Titre de la Fenêtre ");
    setSize(300, 150);
    setLayout(new FlowLayout());
    add(new Button("Bouton 1"));
    add(new Button("Bouton 2"));
    add(new Button("Bouton 3"));
    pack();
    show();
  }
  public static void main(String[] args) {
    new MaFrame();
  }
```

```
}
```

EXEMPLE 2

```
import java.awt.*;
public class MaFrame extends Frame {
  public MaFrame() {
    super() ;
    setTitle("Titre de la Fenêtre");
    setSize(300,150);
    setLayout(new GridLayout(2, 3));
    add(new Button("Bouton 1"));
    add(new Button("Bouton 2"));
    add(new Button("Bouton 3"));
    add(new Button("Bouton 4"));
    add(new Button("Bouton 5 très long"));
    add(new Button("Bouton 6"));
    Pack();
    Show();
  }
  public static void main(String[] args} (
    new MaFrame();
  }
}
```



3. Gestion des événements et écouteur

Les programmes qui possèdent une IHM avec clavier, écran graphique et souris suivent un modèle de programmation appelé programmation par événements. Le programme attend que se produisent des événements, (par exemple, sur le clavier, la souris ou l'écran) pour réagir en déclenchant les procédures appropriées dans les objets qui sont intéressés. En java, les événements utilisateurs sont gérés par plusieurs interfaces EventListener.

Les événements sont en fait des objets. Il existe un thread de la JVM chargé de récupérer les interactions utilisateurs. Si on clique sur un bouton par exemple, ce thread crée un événement du type "clic bouton" en utilisant le bouton comme source pour le constructeur. Ainsi, les événements sont gérés par leur écoute. Tout le principe de la programmation événementielle repose sur la **définition des événements** et des objets qui les **écoutent**.

Un écouteur est un objet destiné à recevoir et à gérer les événements générés par le système. Les écouteurs principaux se trouvent dans le package java.awt.event. Les interfaces EventListener permettent de définir les traitements en réponse à des événements utilisateurs générés par un composant. N'importe quel objet peut devenir un écouteur du moment qu'il implémente les méthodes définies dans une interface.

Pour chaque type d'événements à traiter, une classe doit implémenter une interface réceptrice (qui hérite de l'interface EventListener):

- ActionListener : clic de souris ou enfoncement de la touche Enter
- ItemListener : utilisation d'une liste ou d'une case à cocher
- MouseMotionListener : événement de souris
- WindowListener : événement de fenêtre

L'ajout d'une interface EventListener impose plusieurs ajouts dans le code :

1. importer java.awt.event : `import java.awt.event.*;`
2. la classe doit déclarer qu'elle utilisera une ou plusieurs interfaces d'écoute (un objet de cette classe sera un écouteur) :

```
public class A implements ActionListener, MouseListener
```

3. Appel à la méthode addXXX() pour enregistrer l'objet qui gérera les événements XXX du composant :

```
Button b = new Button("bouton");
b.addActionListener(this);
```

La méthode addActionListener() permet de préciser l'instance qui va gérer l'événement utilisateur de type ActionListener du bouton. Cette classe doit impérativement implémenter l'interface de type EventListener (ActionListener dans ce cas). this indique que la classe elle-même recevra et gérera l'événement utilisateur. Une autre classe indépendante peut être utilisée : dans ce cas il faut préciser une instance de cette classe en tant que paramètre :

```
b.addActionListener(new MonEcouteur());
```

avec :

```
public class MonEcouteur implements ActionListener {
  public void actionPerformed(ActionEvent e) {
        System.out.println("Click sur bouton");
  }
}
```

4. Implémenter les méthodes déclarées dans les interfaces. Chaque récepteur possède des méthodes différentes qui sont appelées pour traiter leurs événements. Par exemple, l'interface ActionListener envoie des événements à une méthode nommée actionPerformed( ).
5. Pour identifier le composant qui a généré l'événement, il faut utiliser la méthode getSource() de l'objet ActionEvent fourni en paramètre de la méthode :

```
Button b = new Button("bouton");
...
void public actionPerformed(actionEvent evt) {
   Object source = evt.getSource();
   if (source == b) // action a effectuer
}
```

La méthode getActionCommand() (à la place de getSource()) renvoie une chaîne de caractères. Si le composant est un bouton, il renvoie le texte du bouton, si le composant est une zone de saisie, c'est le texte saisi qui sera renvoyé (il faut appuyer sur «Entrer» pour générer l'événement), etc ...

REMARQUE : Il est possible d'ajouter plusieurs écouteurs à un seul objet, comme Il est possible d'écouter plusieurs objets avec un seul écouteur.

EXEMPLE COMPLET D'UNE APPLET (VOIR APPLET):

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

public class AppletAction extends Applet implements ActionListener{
  public void actionPerformed(ActionEvent evt) {
    String composant = evt.getActionCommand();
    showStatus("Action sur le composant : " + composant);
  }
  public void init() {
    super.init();
    Button b1 = new Button("boutton 1");
    b1.addActionListener(this);
    add(b1);
    Button b2 = new Button("boutton 2");
    b2.addActionListener(this);
    add(b2);
    Button b3 = new Button("boutton 3");
    B3.addActionListener(this);
    add(b3);
  }
}
```

4. Applet

Une Applet est :

- un programme écrit en java et intégré dans une page web
- utilisée pour créer des applications interactives qu'on ne peut pas faire avec HTML
- chargée depuis une machine distante (serveur) et exécutée sur la machine locale (client)

EXEMPLE :

```
import java.awt.*;
import java.applet.*;
public class MonApplet extends Applet
{
  public void paint(Graphics gc){
    gc.drawString("Hello Web",40,40);
  }
}
```

Ainsi, une applet est un objet de la classe Applet du package java.applet. La classe Applet hérite de la classe Panel définie dans la bibliothèque d'interface graphique awt: une applet est donc un objet graphique.

4.1. Intégration d'applet dans une page HTML

Il faut utiliser le tag APPLET comme dans l'exemple suivant :

```
<HTML>
<HEAD><TITLE> test </TITLE></HEAD>
<BODY>
<APPLET
CODE="Exemple.class"
WIDTH=200 HEIGHT=300
ALIGN=middle>
</APPLET>
</BODY>
</HTML>
```

'Exemple.class' est le nom de l'applet. Le tag APPLET peut comporter, entre autres, le tag PARAM (entre les tags APPLET et /APPLET) qui permet de passer des paramètres à l'applet :

```
<PARAM NAME="nomParametre" value= "valeurParametre"> </APPLET>
```

Qu'on peut récupérer grâce à getParameter("nomParametre").

4.2. Utilisation générale d'une applet

Le mécanisme d'initialisation d'une applet se fait comme suit :

- La machine virtuelle Java instancie l'objet Applet en utilisant le constructeur par défaut puis lui envoie le message **init**() une seule et unique fois après le chargement de l'applet.
- La méthode **start**() est après appelée automatiquement lors du premier affichage de l'applet.
- Lorsqu'on quitte la page web, le navigateur appelle automatiquement la méthode **stop**(). Par défaut, elle interrompt les traitements de tous les processus en cours.
- La méthode **destroy**() est appelée après l'arrêt de l'applet ou lors de l'arrêt de la machine virtuelle. Par défaut, elle libère les ressources et détruit les threads restants.
- La méthode **update**() est appelée à chaque rafraîchissement de l'écran ou appel de la méthode repaint(). Elle efface l'écran et appelle la méthode paint(). Ces actions provoquent souvent des scintillements. Il est préférable de redéfinir cette méthode pour qu'elle n'efface plus l'écran :

```
public void update(Graphics g) { paint (g);}
```

d'une manière générale, on a le cycle d'exécution suivant : init() (start() stop())* destroy()

4.3. Quelques méthodes de l'applet

Méthode showStatus()

Affiche un message dans la barre de statut de la page web :

```
public void paint(Graphics g) {
   super.paint(g);
   showStatus("message affiché dans la barre d'état");
}
```

Méthodes getCodeBase() et getDocumentBase()

Ces méthodes renvoient respectivement l'emplacement de l'applet sous forme d'adresse Web ou de dossier et l'emplacement de la page HTML qui contient l'applet.

Méthodes getImage()

Pour la manipulation des images, le package nécessaire est java.awt.image.

La méthode getImage() possède deux signatures : getImage(URL url) et getImage (URL url, String name).

On procède en deux étapes : le chargement puis l'affichage. Si les paramètres fournis à getImage() ne désignent pas une image, aucune exception n'est levée.

```
public void paint(Graphics g) {
   super.paint(g);
   Image image = getImage(getDocumentBase(), "monimage.gif");
   g.drawImage(image, 40, 70, this);
}
```

Exercices

Exercices

EXERCICE N°1

1. Ecrire un programme qui calcule et affiche la somme des 1000 premiers entiers. On définit dans une classe Somme la méthode main qui effectue tout le traitement.

Modifier le programme précédent pour créer une classe TestSomme qui crée un objet Somme, appelle la méthode CalculSomme de cet objet et affiche le résultat à l'écran. Pour cela, on définit dans la classe Somme la méthode CalculSomme() qui fait le calcul.

EXERCICE N°2

Ecrire un programme Java qui fait la conversion de la température du °C vers °F (farenheit) et inversement. Pour cela, écrire une classe « ConverTemp » appelée par main() qui contient une méthode « converTemp » à laquelle on envoi le sens de conversion ('C' pour F→C et 'F' pour C→F) et la valeur de la température à convertir. Cette dernière retourne le résultat sous forme d'une chaine de caractères. La conversion se fait comme suit :

- Temperature en ° Celcius = (5f/9f) * (temperature°F - 32);
- Temperature en ° Farenheit = (9f/5f) * (temperature°C + 32);

EXERCICE N°3

Ceci est le texte d'une classe représentant un compte bancaire et les opérations bancaires courantes.

```
class Compte{
  int solde = 0;
  void depot(int montant) {
    solde = solde + montant;
  }
  void retrait(int montant) {
    solde = solde - montant;
  }
  void virement(int montant, Compte autre) {
    autre.retrait(montant);
    this.depot(montant);
  }
  void afficher() {
    System.out.println ("solde: "+ solde);
  }
}
```

1. Comment fonctionne la méthode virement ? Combien de comptes fait-elle intervenir ?
2. Créez deux comptes que vous affecterez à deux variables. Ecrivez le code correspondant aux opérations suivantes :
    a. dépot de 5000 DA sur le premier compte.
    b. dépot de 10000 DA sur le second compte.
    c. retrait de 1000 DA sur le second compte.
    d. virement de 3500 DA du premier compte vers le second.
    e. affichage des soldes des deux comptes.

    Mettez le code java correspondant à cette question dans la méthode main d'une nouvelle classe appelée TestC.

3. Créez un tableau de vingt comptes. Dans chaque case, il faut déposer 2000 DA plus une somme égale à 10 fois l'indice du compte dans le tableau. Enfin, écrire le code qui affiche les soldes de tous les comptes.
4. Complétez la classe Compte avec le nom du titulaire du compte (type String). Créez un constructeur pour la classe Compte. Ce constructeur doit prendre en paramètre le nom du titulaire du compte.
5. Modifiez la méthode d'affichage pour qu'elle affiche cette information.

SOLUTION

1. L'instance à laquelle est envoyé le message est débitée par la valeur « montant » envoyée qui est en même temps créditée du compte « autre ». Donc elle fait intervenir deux comptes.

2. 3. 4. 5.

```
public class Compte1 {
   int solde = 0;
   String titulaire;
   public Compte1 (String nomTitulaire) {
      titulaire = nomTitulaire;
   }
   void depot(int montant) {
      solde = solde + montant;
   }
   void retrait(int montant) {
      solde = solde - montant;
   }
   void virement(int montant, Compte1 autre) {
      autre.retrait(montant);
      this.depot(montant);
   }
   public void afficher() {
      System.out.println ("Propriétaire: " + titulaire + " solde: "+ solde);
   }
}

public class TextC {
   public static void main(String[] args) {
      Compte1 c1 = new Compte1("Amar");
      Compte1 c2 = new Compte1("Kadour");
      c1.solde = 5000;
      c2.solde = 10000;
      c2.retrait(1000);
      c2.virement(3500, c1);
      c1.afficher();
      c2.afficher();

   Compte1[] cpts = new Compte1[20];
      for (int i=0; i<20; i++) {
         cpts[i] = new Compte1("Client"+i);
         cpts[i].depot(2000+i*10);
         cpts[i].afficher();
      }
   }
}
```

EXERCICE N°4

Parmi les méthodes de la classe suivante, lesquelles peuvent être statiques et lesquelles ne peuvent en aucun cas être statiques ?

```
class C {
  int x, y;
  String nom;
  void afficher() {
    System.out.println (nom + " " + x + " " + y);
  }
  void ajouter(C obj) {
    x = x + obj.x;
    y = y + obj.y;
    nom = nom + obj.nom;
  }
  C nouveau(int n) {
    C res = new C();
    res.x = n;
    res.y = n*2;
    res.nom = "Auto_"+n;
    return res;
  }
  boolean plusGrand(C obj) {
    if (obj.x == x){
      return y>obj.y;
    }else{
      return x>obj.x;
    }
  }
  boolean compare(C obj1, C obj2) {
    if (obj1.x == obj2.x){
      return obj1.y>obj2.y;
    }else{
      return obj1.x>obj2.x;
    }
  }
}
```

EXERCICE N°5

Donner les sorties du programme Java suivant:

```
class Truc {
  public int i ;
  public Truc(int a) { i = a ; }
  public Truc(Truc t) { i = t.i ; }
  public boolean equals( Truc t ) { return (t.i==i) ; }
  public static void main(String args[]) {
    Truc y = new Truc(1);
    Truc z = y;
    Truc w = new Truc(y);
    if (z==y) System.out.println ( " 1 ") ;
    if (w==y) System.out.println ( " 2 ") ;
    if (z.equals(y)) System.out.println ( " 3 ") ;
    if (w.equals(y)) System.out.println ( " 4 ") ;
  }
}
```

EXERCICE N°6

Donnez les sorties obtenues par l'exécution de ce programme.

```
class Compteur {
  int x;
  Compteur(int n){
    x = n;
```

```
  }
  Compteur incremente(){
    x++;
    return this;
  }
  int value(){
    return x;
  }
}
class C {
  public static void main(String[] argv){
    Compteur c1, c2, c3;
    c1 = new Compteur(0);
    c1.incremente();
    c2 = new Compteur(1);
    c3 = c1;
    if (c1 == c3)
      System.out.println ("c1 et c3 sont égaux");
    else
      System.out.println ("c1 et c3 ne sont pas égaux");
    if (c1.value() == c2.value())
      System.out.println ("c1 et c2 ont même valeur");
    else
      System.out.println ("c1 et c2 n'ont pas la même valeur");
    if (c1 == c2)
      System.out.println ("c1 et c2 sont égaux");
    else
      System.out.println ("c1 et c2 ne sont pas égaux");
    if (c1.value() == c1.incremente().value())
      System.out.println ("c1 et c1 incremente ont même valeur");
    else
      System.out.println ("c1 et c1 incremente n'ont pas la même valeur");
    if (c1 == c1.incremente())
      System.out.println ("c1 et c1 incremente sont égaux");
    else
      System.out.println ("c1 et c1 incremente ne sont pas égaux");
  }
}
```

SOLUTION

```
c1 et c3 sont égaux
c1 et c2 ont même valeur
c1 et c2 ne sont pas égaux
c1 et c1 incremente n'ont pas la même valeur
c1 et c1 incremente sont égaux
```

EXERCICE N°7

Soit le programme JAVA suivant :

```
class A {
  public static A notreA;
  public static int j;
  public int i;
  public A monA;
  public A(int a) { i = a; monA = null; j++; }
  public static void p(A a) {
    j = 0;
    notreA = new A(0);
    m(notreA);
  }
```

```
  public static void q(A a) {
    System.out.print("A.notreA = "); n(notreA);
    System.out.println("A.j = " + j);
  }
  public static void m(A a) {
    System.out.println("A, i = " + a.i);
  }
  public static void n(A a) {
    m(a);
   System.out.print("monA = ");
   if (a.monA!=null) a.monA.m(a.monA);
   else System.out.println("null");
  }
}
class B {
  public static void main() {
    A.p(null);
    A a1 = new A(1);
    A.notreA.monA = r1;
    A a2 = new A(2);
    System.out.print("a2 = ");
    A.n(a2);
    a1.monA = a2;
    System.out.print("a1 = ");
    A.n(a1);
    A.q(a1);
  }
}
```

Le mot-clé ‘static’ est correctement placé pour la déclaration des attributs mais pas pour celle des méthodes. Réécrire ce programme pour que les mots-clés ‘static’ soient partout correctement placés, et que les paramètres inutiles soient enlevés.

EXERCICE N°8

Créer une classe compte dont les attributs membres sont : int Numero; float Solde; String Proprietaire

1. Créer un constructeur par défaut
2. Créer un constructeur qui utilise le numéro et le solde comme argument
3. Créer une méthode qui accède au solde
4. Créer une méthode qui accède au numéro de compte
5. Créer une méthode appelée "débiter" qui retire un montant du solde. Il faut contrôler que le montant M doit être inférieur au solde
6. Créer une méthode appelée "créditer" qui ajoute un montant au solde
7. Redéfinir la méthode toString pour qu'elle affiche le "Compte/Propriétaire/solde"
8. Créer une variable de classe qui permet de compter le nombre des comptes bancaires qui sont ouverts
9. Créer un destructeur pour un compte supprimé

SOLUTION

```
public class Compte2 {
   int numero;
   float solde;
   String propriete;
   static int nbreCompte;
   public static void afficher() {
      System.out.println("Nombre de comptes: " + nbreCompte);
```

```
    }
    public Compte2 () {
       nbreCompte++;
    }
    public Compte2 (int n, float s) {
       this();
       numero = n;
       solde = s;
    }
    void finalyse() {
       nbreCompte--;
    }
    public String toString() {
      String s = "Compte:"+numero+"/Propriétaire:"+propriete+"/Solde:"+solde;
      System.out.println(s);
      return s;
    }
    static public void main(String[] s) {
       Compte2 c1 = new Compte2(150,1000);
       Compte2 c2 = new Compte2(200,2000);
       Compte2 c3 = new Compte2();
       c1.propriete = "Amar";
       c2.propriete = "Mohamed";
       Compte2.afficher();

       c1.toString();
       c2.toString();
       c3.toString();
    }
}
```

EXERCICE N°9

Écrivez une classe Piece qui représente une pièce à deux faces que l'on peut lancer. La classe doit posséder, outre un constructeur sans paramètre, trois méthodes dont les signatures sont les suivantes :

- public void lancer()
- public boolean estcePile ()
- public boolean estceFace()

La première méthode permet de lancer la pièce. La seconde permet de tester si la face visible de la pièce est pile et la troisième méthode teste si la face visible est face.

NB. La méthode Math.random() renvoie un double entre 0 (compris) et 1 (non-compris).

Écrivez ensuite une classe TestPiece qui crée une pièce, la lance 100 fois d'affilé et affiche le nombre total de pile et de face qu'il y a eu en tout sur les 100 lancées.

EXERCICE N°10

Ecrire le code d'une classe qui permet d'attribuer un numéro unique à chaque nouvel objet créé (1 au premier, 2 au suivant...). On ne cherchera pas à réutiliser les numéros d'objets éventuellement détruits. On dotera la classe d'un constructeur, d'une méthode getIdent retournant le numéro attribué à l'objet et d'une méthode getIdentMax fournissant le numéro du dernier objet créé.

Écrire un programme d'essai.

EXERCICE N°11

Soit la classe Point ainsi définie :

```
class Point
{
   public Point (int abs, int ord) {x = abs; y = ord ;}
   public void affiche () {
      System.out.println ("Coordonnées " + x + " " + y) ;
   }
   private double x ; // abscisse
   private double y ; // ordonnee
}
```

Ajouter une méthode *maxNorme* déterminant parmi deux points lequel est le plus éloigné de l'origine et le fournissant en valeur de retour. On donnera deux solutions :

- *maxNorme* est une méthode statique de Point,
- *maxNorme* est une méthode d'instance de Point.

Quelle solution préfèrerez-vous ? Pourquoi ?

EXERCICE N°12

Ecrire trois classes : Figure, Carre, et Rectangle, telles que :

1. Figure a des attributs abscisse et ordonnée, ainsi qu'une couleur (encodée par un entier).
2. Carre et Rectangle héritent de Figure, mais ont en plus une ou deux longueurs pour les côtés (selon le cas).
3. Figure a un attribut privé Vector référençant toutes les instances de sa classe et de ses sous classes (pour cela, on doit importer java.util.Vector).
4. Figure a une méthode statique getInstances() renvoyant ce vecteur.
5. Carre et Rectangle redéfinissent cette méthode getInstances() de manière à ne récupérer que les instances qui correspondent à leur type. Pour cela, utilisez le comparateur 'instanceof'.

SOLUTION

```
import java.util.Vector ;
public class Figure {
   private static Vector instances = new Vector();
   private int abscisse;
   private int ordonnee;
   private int couleur;
   public Figure(int abscisse, int ordonnee, int couleur) {
      this.abscisse = abscisse;
      this.ordonnee = ordonnee;
      this.couleur = couleur;
      instances.add(this);
   }
   public static Vector getInstances() { return instances ; }
   public String toString(){
     return (abscisse+" "+ordonnee+" "+couleur);
   }
}

import java.util.Enumeration;
import java.util.Vector;
public class Carre extends Figure {
   private int cote ;
   public Carre(int abscisse, int ordonnee, int couleur, int cote) {
```

```
      super(abscisse, ordonnee, couleur);
      this.cote = cote;
   }
   public static Vector getInstances() {
      int nCarre = 0;
      Vector<Figure> instancesCarre = new Vector();
      Vector<Carre> instances = Figure.getInstances();
      //Enumeration e = instances.elements();
      Figure uneFigure;
      //while(e.hasMoreElements()) {
      for (Enumeration<Carre> e=instances.elements();e.hasMoreElements();){
         uneFigure = (Figure)e.nextElement();
         if (uneFigure instanceof Carre)
            instancesCarre.add(uneFigure);
      }
      return instancesCarre;
   }
   public String toString(){
      return (super.toString()+" "+cote);
   }
}

import java.util.Vector ;
import java.util.Enumeration ;
public class TestFig {
   public static void main(String[] argv) {
      Figure f1 = new Figure(1,1,1);
      Figure f2 = new Figure(2,2,2);
      Carre c1 = new Carre(3,3,3,3);
      Figure f4 = new Figure(4,4,4);
      Carre c2 = new Carre(5,5,5,5);
      System.out.println("Liste des figures") ;
      Enumeration e = Figure.getInstances().elements() ;
      while(e.hasMoreElements())
         System.out.println(e.nextElement()) ;
      System.out.println("Liste des carrés") ;
      e = Carre.getInstances().elements() ;
      while(e.hasMoreElements())
         System.out.println(e.nextElement()) ;
   }
}
```

EXERCICE N°13

Soit le programme Java suivant:

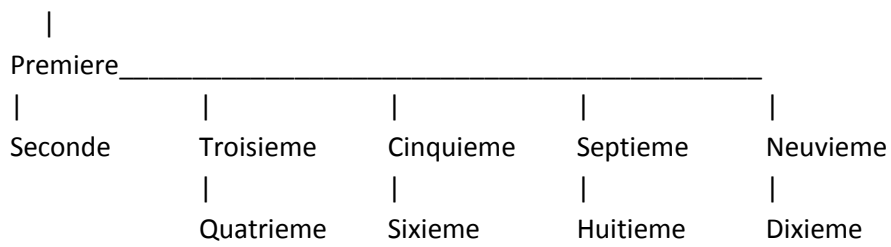
```
class Premiere {
   Premiere(){ System.out.println ("constructeur de Première"); }}
class Seconde extends Premiere {
   Seconde(boolean b) { super();
      System.out.println ("constructeur de Seconde"); }}
class Troisieme extends Premiere {
   Troisieme(int i){ super();
      System.out.println ("constructeur de Troisième"); }}
class Quatrieme extends Troisieme {
   Quatrieme(double d) {super(14);
      System.out.println ("constructeur de Quatrième"); }}
class Cinquieme extends Premiere{
   Cinquieme() { System.out.println ("constructeur de Cinquième"); }}
class Sixieme extends Cinquieme {
   Sixieme() { System.out.println ("constructeur de Sixième"); }}
class Septieme extends Premiere {
   Septieme(int i) { System.out.println ("constructeur de Septième"); }}
```

```
class Huitieme extends Septieme {
   Huitieme() { System.out.println ("constructeur de Huitième"); }}
class Neuvieme extends Premiere {
   Neuvieme(int i){System.out.println ("premier constructeur de Neuvième");}
   Neuvieme(boolean b) {
      System.out.println ("2ème constructeur de Neuvième"); }}
class Dixieme extends Neuvieme {
   Dixieme(double d){
      super(true); System.out.println ("premier constructeur de Dixième"); }
   Dixieme(int x, boolean y) {
      super(x); System.out.println ("second constructeur de Dixième"); }}
class Test{
       public static void main(String[] args) {
              new Premiere(); new Seconde(true);
              new Troisieme(15); new Quatrieme(12.3);
              new Cinquieme(); new Sixieme();
              new Huitieme(); new Dixieme(10.5); new Dixieme(45, true); }
}
```

1. Dessinez le graphe d'héritage de ce programme (y compris la classe "Object").
2. Le programme provoque une erreur à la compilation. Une classe est incorrecte, laquelle ? Quel est le problème ? Proposez une solution.
3. Donnez l'affichage réalisé par ce programme dans la console.

SOLUTION

1. Le graphe d'héritage est le suivant :



2. La classe qui présente un problème de compilation est la classe Huitieme. Elle étend la classe Septieme qui n'a pas de constructeur par défaut ; en même temps, Huitieme ne fait pas d'appel explicite à ce constructeur. Comme le compilateur insère un appel au constructeur par défaut du fait qu'il n'y a aucun appel au constructeur de la classe mère, il génère une erreur de compilation. Deux solutions sont possibles : soit définir un constructeur par défaut dans la classe mère et cette solution est déconseillé (particulièrement pour ce cas, ce n'est pas méchant mais en général cela implique une modification de toutes les classes filles), soit faire un appel explicite au constructeur de la classe mère au niveau de la classe fille, soit « super(i) ; ».

3. L'affichage réalisé est :

```
constructeur de Première
constructeur de Première
constructeur de Seconde
constructeur de Première
constructeur de Troisième
constructeur de Première
constructeur de Troisième
constructeur de Quatrième
constructeur de Première
constructeur de Cinquième
```

```
constructeur de Première
constructeur de Cinquième
constructeur de Sixième
constructeur de Première
constructeur de Septième
constructeur de Huitième
constructeur de Première
2ème constructeur de Neuvième
premier constructeur de Dixième
constructeur de Première
premier constructeur de Neuvième
second constructeur de Dixième
```

EXERCICE N°14

Considérons un éleveur de lapins et de poulets. Les lapins de reproduisent localement mais les jeunes poulets sont reçus à un certain âge. Les deux animaux sont élevés jusqu'à ce qu'ils aient la taille nécessaire à leur commercialisation. Chaque animal est caractérisé par son poids et un numéro d'identification reporté sur une bague qu'il porte à sa patte ou à l'oreille.

Le prix de vente est différent selon l'animal et est exprimé en DA par kilo. Ces prix varient chaque jour. Le poids auquel on abat les bêtes est différent

Ecrire une classe des animaux élevés avec deux sous-classes des poulets et des lapins. Il faut pouvoir enregistrer les prix du jour, les poids d'abattage, le poids d'un animal donné.

Ecrire une classe permettant de représenter l'ensemble des animaux de l'élevage au moyen d'un tableau. Des méthodes doivent permettre d'identifier les animaux à abattre et d'évaluer le prix obtenu pour ces animaux. Il faut également pouvoir enregistrer les jeunes animaux qui arrivent ou qui naissent.

Références

1. Site officiel d'Oracle : www.oracle.com/technetwork/java/index.html
2. « Java de l'esprit à la méthode ». Michel Bonjour, Gilles Falquet, Jacques Guyot et André LEGRAND. Distribution d'applications sur Internet, 2ème edition (1999). http://cui.unige.ch/java/livre/Java2EaM.pdf
3. « Java pour les enfants, les parents et les grands-parents ». Yakov Fain (2007, Mis à jour en 2015). http://java.developpez.com/livres-collaboratifs/javaenfants/
4. « La programmation orientée objet ». Hugues Bersini, 6ème Édition (Eyrolles). https://slashtsdi.files.wordpress.com/2011/04/la-programmation-orientee-objet.pdf
5. « Programmation objet avec Java ». http://deptinfo.cnam.fr/Enseignement/CycleA/APA/nfa032
6. « Exercices en langage JAVA ». H. Frezza-Buet et M. Ianotto. 2003.