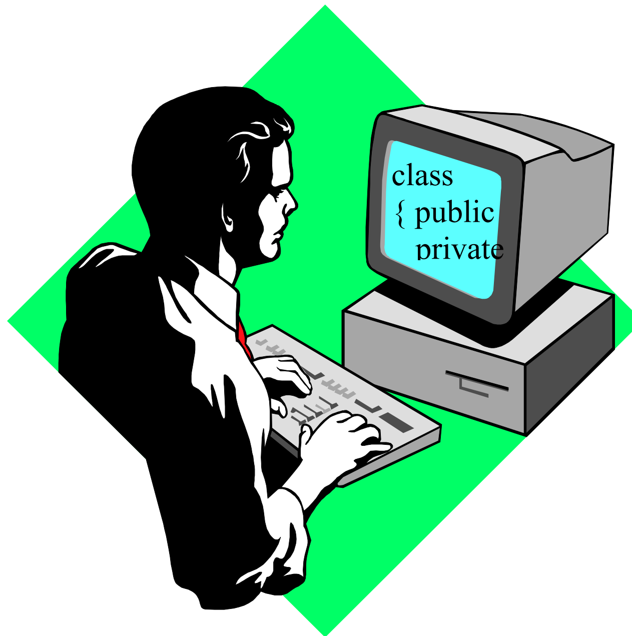
DI GALLO Frédéric

Programmation Orientée Objets en JAVA

Cours du Cycle d'Approfondissement



Cours dispensé par B. Chaulet

CNAM ANGOULEME 2000-2001

CONSERVATOIRE NATIONAL DES ARTS ET METIERS

PROGRAMMATION ORIENTEE OBJETS :

Généralités sur JAVA

PROGRAMMATION JAVA: GENERALITES

I. PROGRAMMATION CLASSIQUE ET ORIENTEE OBJETS 4
II. CLASSE ET OBJET 5
III. LES DONNEES D'UNE CLASSE 7
IV. LES METHODES D'UNE CLASSE 9
4.1) Méthodes publiques de type « Constructeur » 12
4.2) Méthodes publiques de type « Accesseur » 14
4.3) Les méthodes de type binaire 18
4.4) La transmission de paramètres 22
4.5) Le mot clé « this » 23
V. LES VARIABLES ET METHODES DE CLASSE 24
VI. TRAVAUX PRATIQUES 25
6.1) Utilisation de la classe String 25
6.2) Création d'une classe 27
VII. ANNEXE : LES ETAPES DE DEVELOPPEMENT EN JAVA 28

LES COLLECTIONS EN JAVA

I. INTRODUCTION 31
II. LES TABLEAUX 33
III. LES VECTEURS 36
3.1) La classe générique « Object » 36
3.2) Les méthodes de la classe « Vector » 38
3.3) La classe « Enumeration » 40
3.4) Les packages 41
IV. LES DICTIONNAIRES 42
V. LES FLUX 44
5.1) Opérations sur un fichier 44
5.2) La gestion des erreurs 46

L'HERITAGE EN JAVA

I. INTRODUCTION 54
II. DEFINITION DE L'HERITAGE 56
2.1) Le constructeur d'une classe dérivée 57
2.2) Les propriétés d'une classe dérivée 58
2.3) Les méthodes d'une classe dérivée 59
III. MANIPULATIONS D'OBJETS ISSUS DE CLASSE MERE ET FILLE 60

EXEMPLES DE CLASSES JAVA

INFORMATIQUE – CNAM ANGOULEME 2000-2001

PROGRAMMATION JAVA: GENERALITES

I. Programmation classique et orientée objets

En <u>programmation classique</u> le développement d'une application se décompose en deux étapes bien distinctes :

- La définition des structures de données capables d'accueillir l'information que l'on veut gérer. *Ainsi, en programmation classique, pour stocker en mémoire centrale une fraction on a introduit un enregistrement composé de 2 champs de type «Entier» nommés « Numerateur » et « Denominateur ».*

- La conception de fonctions et de procédures qui travaillent sur les structures de données conçues à l'étape précédente et qui permettent de mettre en œuvre tous les besoins exprimés dans le cahier des charges.

Cette forme de programmation, qui sépare les données des traitements, rend difficile la mise en place de <u>contrôles</u>, destinés à garantir la cohérence et l'intégrité des données. L'utilisateur peut en effet accéder directement aux données, sans utiliser les fonctions mises à sa disposition. Par exemple, le programme d'utilisation des fractions ci-dessous, définit partiellement la fraction F ce qui va entraîner des <u>erreurs</u> lors de l'appel à certaines fonctions présentes dans la bibliothèque « Fraction.h ».

```
#include<fraction.h>
main()
{       struct Fraction F, R;
        F.Numerateur = 3 ;
        R=Inverse (F) ;
}
```

De plus, toute modification de l'implémentation des données à des conséquences sur l'ensemble des fonctions travaillant sur ces données, et donc sur l'ensemble des traitements utilisant ces fonctions. Si l'on choisit par exemple, pour des raisons d'optimisation, d'implanter une fraction au moyen d'une chaîne de caractères (« 4/5 »), toutes les fonctions de la bibliothèque « Fraction.h » sont à modifier au niveau du type des paramètres d'entrée ou de sortie. Il va de soi que ces changements imposent, à leur tour, une modification de tous les programmes qui référencent la bibliothèque « Fraction.h ».

La programmation orientée objets a été introduite afin de remédier à ces inconvénients. Son principe de base consiste :

- d'une part à regrouper (ou à "encapsuler") au sein d'une même unité (appelée classe) les données et les traitements. Par exemple, on définira la classe « Fraction » de la manière suivante :

```
class Fraction
{       private  int Numerateur ;          } déclaration de variables destinées
        private int Denominateur ;           à mémoriser une fraction.
        public void initialiser(int n, int d)
        {Numerateur = n ;                  } traitements sur les fractions.
        Denominateur = d ;
        }
        …
}
```

Une fraction est définie par un numérateur, un dénominateur ainsi que par toutes les opérations mathématiques existantes sur les fractions.

- d'autre part à contrôler l'accès aux données en les déclarant privées ce qui obligera l'utilisateur de la classe à employer les traitements encapsulés pour agir sur celles-ci.

En conclusion, la programmation orientée objets a deux avantages principaux:
- **contrôler l'accès aux données**,
- **permettre la modification des classes** sans avoir d'incidence sur les programmes des utilisateurs.

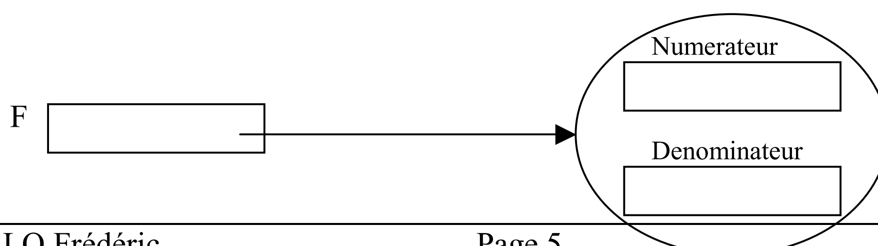
II. Classe et objet

Une classe est une description abstraite de structure de données et de traitements destinée à mémoriser et à gérer une réalité ou une notion. *Par exemple la classe « Fraction » permet de mémoriser des fractions et de mettre en œuvre des opérations sur ces dernières.*

A partir d'une classe, tout langage objet permet (par l'intermédiaire de l'opérateur **new**) de créer des objets qui seront des emplacements en mémoire centrale destinés à contenir effectivement l'information manipulée. Ces objets sont appelés **instance** et l'opération de création : **instanciation**. Dans l'ouvrage « Java, la synthèse », les auteurs définissent une classe comme un "moule" à partir duquel on peut obtenir autant d'objets que nécessaires.

Le programme Java suivant illustre la création d'un objet de la classe « Fraction ».

```
Fraction F;              // déclaration d'une variable qui va pointer sur un objet : « Fraction » ;
                         // F contient la valeur null
F= new Fraction() ;      // création d'un objet « Fraction », c-à-d. allocation en mémoire centrale
                         // d'un espace destiné à recevoir la valeur d'une fraction.
                         // la variable F contient l'adresse de cet emplacement.
```



Il est important de noter que la variable F contient l’adresse de l’objet et non l’objet lui-même. Ainsi, si on déclare une autre variable « G » de la classe « Fraction », l’affectation «G=F» aura pour conséquence d’affecter dans la variable « G » l’adresse contenue dans « F ». A l’issue de cette instruction « F » et « G » pointe le même objet.

En langage Java, l’opérateur « **new** » recherche l’espace nécessaire à la représentation de l’objet et renvoie l’adresse de cet objet. Dans le cas où l’opérateur n’obtient pas l’allocation nécessaire, il termine l’exécution en renvoyant l’exception « OutOfMemoryError » (qui sera étudiée ultérieurement). Dès que l’espace alloué à un objet n’est plus désigné par une variable, l’espace mémoire est récupéré par un système de ramasse-miettes.

Application: On considère la classe « Date » :

Elle est définie par 3 données: Jour, Mois et Année, ainsi que par toutes les opérations sur une date telles que l’ajout de 2 dates, le calcul de la date du lendemain, etc.

Analyser l’extrait du programme Java suivant et indiquer combien d’objets de la classe «Date » ont été créés au cours de l’exécution.

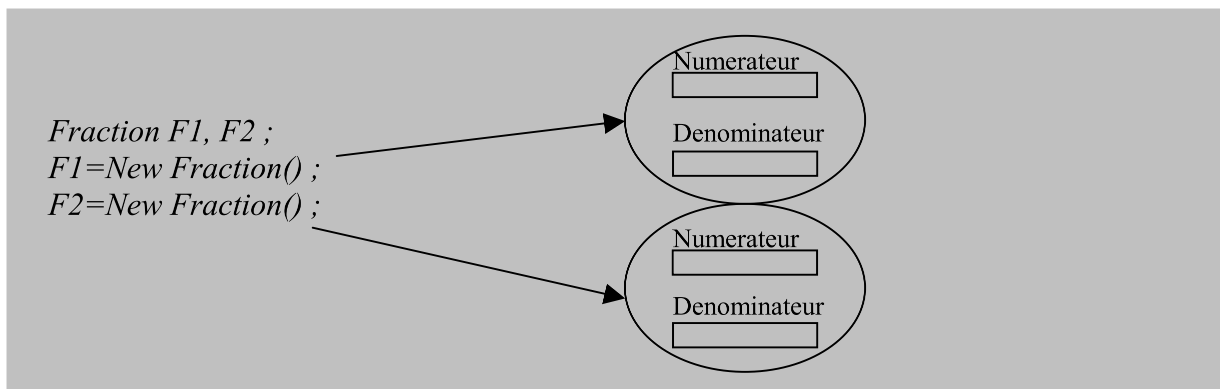
```
0   int I ;
1   Date D1,D2 ;
2   D1=new Date() ;
3   D2=D1 ;
4   D1=new Date() ;
```

N°	Type	Image
0	ne variable de type primitive	I
1	D NULL ariables D1 et D2 destinées à contenir des objets de type Date	D1 D2
2	In NULL D1	D1 D2 Jour Mois Annee
3	In D2 vers l'o de D1	D1 D2 Jour Mois Annee
4	N iation de D	D1 Jour Mois Annee

III. Les données d'une classe

Toute classe comporte une partie données, dans laquelle on décrit les emplacements mémoires - appelés **variables d'instance** - qui seront utiles à la mémorisation de l'information que l'on veut gérer. Ces emplacements sont en fait alloués à chaque instanciation et sont caractérisés par un nom et un type qui peut être simple ou structuré.

La classe « Fraction » utilise, pour mémoriser une fraction, deux variables d'instances de type simple « Entier » : Numerateur et Denominateur. A chaque instanciation de la classe «Fraction», il y aura création de ces deux variables comme le montre le schéma ci-dessous :



On remarque donc, qu'à un moment donné, une variable d'instance existe en autant d'exemplaires qu'il y a d'objets instanciés.

La syntaxe Java pour déclarer les variables d'instance d'une classe est la suivante:

SpécificateurAccès NomType NomVariable [=ValeurInitiale]

- Spécificateur d'accès représente le niveau de protection de la donnée; 2 niveaux Java:
  - **public** : l'accès direct à la donnée est possible depuis tout programme d'utilisation de la classe par la référence : *VariableObjet.VariableDInstance*
  - **private** : l'accès direct à la donnée n'est pas possible depuis un programme d'utilisation de la classe. L'utilisateur devra utiliser des traitements encapsulés pour lire ou mettre à jour ces données.

Le programme ci-dessous illustre les conséquences de l'affectation des spécificateurs :
- *« private » à la variable « Numerateur »,*
- *« public » à la variable « Denominateur » dans l'utilisation de la classe « Fraction » :*

```
Fraction F ;
F=New Fraction()
F.Numerateur =2          // affectation refusée à la compilation car la variable « Numerateur » est privée
F.Denominateur = 3       // affectation autorisée car la variable « Denominateur » est publique
```

- <u>La valeur initiale</u> sera la valeur affectée dans la variable d'instance lors de la création d'un objet. Dans le cas où cette valeur n'est pas précisée, c'est une valeur qui dépend du type de la variable d'instance qui est affectée (*par exemple 0 pour le type « Entier »*).

La partie données de la classe « Fraction » pourrait s'écrire ainsi :

```
class Fraction
{    private int Numerateur ;
     private int Denominateur =1;
     … }
```

Application: On suppose que la partie « Données » de la classe « Date » :

```
class Date
{    public int Jour;
     public int Mois;
     public int Annee ;
     …
}
```

1. On considère le programma Java de test de la classe « Date » suivant. Indiquer le message affiché par le programme lors de son exécution.

```
class test
{    public static void main(String args[])
     {        Date D1,D2 ;
              D1=new Date() ;
              D1.Jour = 1 ;
              D1.Mois = 1 ;
              D1.Annee = 2001 ;
              D2 = New Date() ;
              D2.Jour = 1 ;
              D2.Mois = 1 ;
              D2.Annee = 2001 ;
              if (D1==D2)   System.out.println(« égalité ») ;
              else      System.out.println (« Différence ») ;
     }
}
```

Le message affiché est "**Différence**" car D1 et D2 sont différents. Ils contiennent les adresses distinctes des classes, seul leur contenu est égal.

2. A l'issue de l'instanciation d'un objet de la classe « Date », indiquer le contenu de ses variables d'instance.

Jour, Mois et Annee sont des entiers, donc leur contenu "initial" est 0.

3. La définition de la classe « Date » autorise-t-elle à créer (dans un programme d'utilisation de la classe) des objets qui contiennent des valeurs qui ne correspondent pas à une date (par exemple le :32/02/2000).

Oui, on a le droit de mettre n'importe quel entier, même s'il ne correspond pas à une date

IV. Les méthodes d'une classe

Les méthodes d'une classe (encore appelées comportements ou traitements) se répartissent en deux catégories :

- Les **méthodes publiques** qui correspondent à tous les traitements que l'utilisateur d'une classe pourra appliquer aux objets de la classe. Certains auteurs qualifient ces méthodes publiques de messages que l'on peut envoyer aux objets.
- Les **méthodes privées** qui sont des traitements introduits pour simplifier la conception de la classe, mais qui ne pourront pas être utilisés depuis un programme d'utilisation de la classe. *Exemple: dans la classe « Fraction » la méthode destinée à calculer le PGCD de deux nombres entiers est une méthode privée dont l'existence rend plus simple le traitement destiné à mettre sous forme irréductible une fraction.* Dans ce chapitre, on limitera l'étude aux méthodes publiques.

En Java la déclaration d'une méthode publique respecte la syntaxe suivante :

```
Public TypeRésultat NomFonction (Liste Paramètres formels)
{Déclaration de variables locales
…
Instructions
}
```

Remarque : on appelle signature de la méthode le nom de la fonction suivi de la liste de ses paramètres formels.

Comme pour le langage C, dans le cas où la fonction ne renvoie aucun résultat, on doit spécifier le mot clé « **void** » pour le « TypeRésultat ». Dans le cas où la fonction renvoie un résultat, ce dernier devra être retourné par l'instruction « **return** Résultat ». La liste des paramètres formels et des variables locales se déclare de la même manière qu'en C. Remarquons que ces deux listes peuvent être vides.

En plus des paramètres d'entrée et des variables locales, toute méthode référence tout, ou une partie, des variables d'instance de la classe. Une méthode publique étant toujours appliqué à un objet de la classe, ce seront les variables de cet objet qui seront effectivement référencées lors de l'exécution de la méthode. *Ce principe est illustré ci-dessous par la méthode « Afficher » de la classe « Fraction ».*

```
class Fraction
{	private int Numerateur ;
	private int Denominateur ;

	public void Afficher()
	{
	System.out.println(Numerateur + « / » + Denominateur) ;
	}
}
```

En appliquant la méthode « Afficher » à l'objet de gauche, ce seront les contenus des variables «Numerateur» et «Denominateur» de cet objet qui seront affichés c-à-d. « 4/8 ».

L'application d'une méthode sur un objet s'écrit : VariableObjet.NomMéthode(..°). *Par exemple, en supposant que la variable « F » pointe sur l'objet ci-dessus, l'affichage de la fraction contenue dans cet objet s'écrira : « F.Afficher() ».*

Comme il a été déjà dit dans le chapitre précédent, si la méthode renvoie un résultat, l'application de la méthode à un objet doit être intégrée à une instruction, sinon elle devra être spécifiée comme une instruction. *La méthode « Afficher() » s'utilise comme une instruction car elle n'admet aucun paramètre de sortie.*

Exemple:

concepteur

```
class Date
{       public int Jour;
public int Mois;
public int Annee ;

        public void Afficher()
        {    System.out.println(Jour + « / » + Mois+ « / » + Annee) ;    }

        public int MonJour()
        {    return Jour ;    }
}
```

utilisateur

```
class test
{       public static void main(String args[])
{       Date D1 ;
        D1=new Date() ;
        D1.Jour = 12 ;
                D1.Mois = 02 ;
        D1.Annee = 2001 ;
        D1.Afficher() ;
        System.out.println (D1.MonJour()+25) ;
}
}
```

Application: Compléter la description de la classe « Fraction » :

1. En écrivant le contenu des méthodes suivantes :

```
public void Initialiser(int N, int D)
{    // valorise les variables d'instance de l'objet créé avec les valeurs fournies en paramètre d'entrée. }

public void Inverse()
{    // inverse le contenu de  la fraction      }

public float Valeur()
{    // renvoie la valeur en réelle de la fraction        }
```

- ```
  public void Initialiser(int N, int D)
  {   Numerateur = N ;
      Denominateur = D ; }
  ```

- ```
  public void Inverse()
  {   int x ;
      x = Numerateur ;
      Numerateur = Denominateur ;
      Denominateur = x ; }
  ```

- ```
  public float Valeur()
  {   return ( Numerateur / (float) Denominateur ) ; }
  ```

2. Construire un programme Java de test de la classe « Fraction » qui réalise les différentes opérations suivantes: création de la fraction 6/5, affichage de sa valeur réelle, affichage de la valeur réelle de son inverse.

```
class test
{          Fraction F1 ;   // F1 contient NULL
           F1=new Fraction () ;                      // F1 ——▶Numerateur: 0 - Denominateur: 0
           F1.Initialiser (6,5) ;                    // F1 initialisé à  6 / 5
           System.out.println (F1.Valeur() ) ;       // afficher la valeur réelle de F1
           F1.Inverse () ;
           System.out.println (F1.Valeur() ) ;       // on ne peut pas utiliser la méthode
}                 // Afficher() car elle affiche uniquement le contenu de la fraction, pas sa valeur réelle.
```

Pour ne pas perdre le contenu de F1 avant l'inversion, il est faut modifier la classe Fraction:

- ```
  public Fraction Inverse()
  {   Fraction Res ;
      Res = new Fraction () ;
      Res.Numerateur = Denominateur ;     Res.Denominateur = Numerateur ;
      return Res ; }
  ```

- ```
  {Fraction F1, F2 ;                                  // F1 et F2 contienent NULL
  F1=new Fraction () ;       F1.Initialiser (6,5) ;   // F1 initialisé à  6 / 5
  F2 = F1.Inverse () ;
  System.out.println (F2.Valeur() ) ; }
  ```
  ```

4.1) Méthodes publiques de type « Constructeur »

Un constructeur est une méthode qui porte le nom de la classe et qui est chargé d'initialiser les variables de l'objet au moment de sa création. Le langage Java définit à chaque classe un constructeur par défaut qui affectera des valeurs dépendantes du type de la variable d'instance si aucune valeur d'initialisation n'est précisée dans la déclaration. Le tableau ci-dessous récapitule, pour chaque type, les valeurs affectées par défaut.

Type	**Valeur**
Entier	0
Réel	0.0
Chaîne de caractères	Null

En respectant ce principe, lors de la création d'un objet de la classe « Fraction » définie au début de cette page, le constructeur « Fraction() » initialise les variables « Numerateur » et « Denominateur » à 0.

Si on se limitait à l'existence du constructeur par défaut, toute instanciation devrait être suivie d'une phase d'affectation pour valoriser les variables d'instances de l'objet. Or plus le nombre d'étapes pour atteindre un objectif est grand plus le risque d'oublier une étape est grand. *Par exemple si on oublie de valoriser les variables de l'objet Fraction créé, on travaillera avec une fraction ayant un numérateur et un dénominateur = 0 ce qui peut entraînera des erreurs d'exécution.* Pour remédier à ce problème, Java propose au concepteur de la classe de construire ses propres constructeurs. La définition d'un nouveau constructeur devra respecter la syntaxe suivante :

```
public NomClasse(Liste des paramètres formels)
{…
}
```

On remarque qu'un constructeur n'est pas typé. Une fois le constructeur défini celui-ci est appelé par l'opérateur « new » comme pour le constructeur par défaut.

Le constructeur suivant permet de créer une fraction à partir de la donnée d'un numérateur et d'un dénominateur passés en paramètre d'entrée.

```
public Fraction (int N, int D)          // Fraction étant le nom de la classe, ceci est le constructeur
{       Numerateur = N;
        If (Denominateur ==0)
        {       System.out.println (« le dénominateur ne peut être égal à 0 ») ;
                System.exit(1) ;
        }
        else
           Denominateur = D;            // Après cela, le constructeur d'origine n'existe plus.
}
```

L'affectation F=New Fraction (2,4) créera un objet « Fraction » avec comme numérateur la valeur 2 et un dénominateur égal à 4.

Il est possible de créer pour une même classe plusieurs constructeurs dès lors qu'ils ont des signatures différentes. *Par exemple, pour la classe Fraction on pourrait définir le second constructeur suivant :*

```
public Fraction(int N)
{       Numerateur = N ;
        Denominateur = 1 ;
}
```

Lors de l'appel d'un constructeur Java recherche le constructeur qui est conforme avec l'appel et exécute ses instructions. *Ainsi, lors de l'exécution de l'instruction F=New Fraction(5), Java exécutera le second constructeur, alors que l'instruction F=New Fraction(5,3) déclenchera l'exécution du premier constructeur.* Lorsqu'une classe comporte plusieurs constructeurs on dit que le constructeur est surchargé.

Application: On considère la classe Horaire :

Dont la partie Données est fournie ci-dessous :

```
class Horaire
{   private int heure ;
    private int Minute ;
    …
}
```

Ecrire le contenu du constructeur suivant :

```
public Horaire (int H, int M)
{   if ( H > 23 || H < 0 )
        { system.out.println ("L'heure doit être comprise entre 0 et 23 ») ;
          system.exit (1) ;
        }
    else heure = H ;
    if ( M > 59 || M < 0 )
        { system.out.println ("Les minutes doivent être comprises entre 0 et 59 ») ;
          system.exit (1) ;
        }
    else Minute = M ;
}

class TestHoraire
{   Horaire H ;
    H = new Horaire (5,55) ;
}
```

On peut créer un constructeur *public Horaire ()* (c'est à dire sans paramètre). « F = new Horaire () » ne prendra pas en compte le constructeur par défaut mais celui-ci.

4.2) Méthodes publiques de type « Accesseur »

Lorsque les variables d'instances d'une classe sont déclarées avec le spécificateur d'accès « private » elles ne sont pas accessibles directement depuis un programme d'utilisation de la classe. Ce dispositif permet donc de garantir la cohérence et l'intégrité des données contenues dans les différents objets de la classe. Pour permettre à un utilisateur de la classe de consulter ou de modifier les données ainsi protégées, le concepteur de la classe doit définir des fonctions d'accès publiques encore appelées accesseurs. Celles-ci peuvent être de deux types :

- **en consultation seulement** : la fonction d'accès renvoie la valeur de la donnée mais ne permet pas de la modifier,
- **en modification** : la fonction modifie la valeur de la donnée.

Les deux accesseurs en consultation suivants permettent d'obtenir le numérateur et le dénominateur d'une fraction:

```
public int SonNumerateur ()
{       return Numerateur ;
}

public int SonDenominateur ()
{       return Denominateur ;
}
```

Avec de telles fonction d'accès pour connaître la valeur du numérateur d'un objet, on écrira la référence : VariableObjet.SonNumerateur()

Remarque : certaines classes peuvent comporter des données privées pour lesquelles il n'existe aucun accesseur. Dans ce cas il s'agit de données d'implémentation qui sont introduites pour faciliter l'écriture de la classe mais dont l'existence ne doit pas être connue de l'utilisateur de la classe.

```
                class Horaire
                {   private int heure ;
                    private int Minute ;

accès en           public int SonHeure ()
consultation         { return heure ; }
                   public int SaMinute ()
                     { return Minute ; }

accès en            public void Changer ()
modification          { heure = 12;
                        Minute = 0 ;
                      }
                }
```

```
H1 = new horaire (8,20) ;
system.out.println (H1.SonHeure ()) ;
system.out.println (H1.SaMinute ()) ;
```

Auto-évaluation n°6: On considère la classe « Compte » :

Destinée à gérer des comptes bancaires. Chaque objet de cette classe sera décrit au moyen des 3 données privées suivantes :

Nom de la propriété	Type	Rôle
Numero	Entier	Identifie un compte
Type	Caractère	Fournit le type du compte : J → compte joint C → compte courant
Solde	Réel	Mémorise le solde du compte

1. Définir les variables d'instance de la classe « Compte »

```
class Compte
{       private  int  Numero ;              // Il vaut mieux déclarer les variables en privé
private char Type ;
private float Solde ;
}                   // Types primitifs : la variable contient la valeur et pas un pointeur.
```

2. On considère le programme de test de la classe « Compte » suivant :

```
class TestCompte
{public static void main (String args[])
    { Compte C;
     C = new Compte ();
     }
}
```

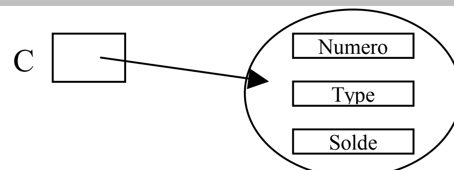
2.1 Indiquer le contenu de la variable C:

- à l'issue de l'exécution de la déclaration : « Compte C » :

C contient la valeur NULL

- à l'issue de l'exécution de l'instruction d'instanciation :

C « pointe » vers un objet de type Compte.



2.2 Indiquer le contenu des variables d'instance de l'objet créé :

Numero : 0 / Type : NULL / Solde : 0.0

2.3 Ecrire la méthode privée TestType définie ci-dessous :

Renvoie true si le type passé en paramètre est égal à J ou C, et false sinon.

```
private boolean TestType (char T)          // renvoie true si le type passé en paramètre
{ if ( T = = "J" || T = = "C" ) return true;   // est égal à J ou C, et false sinon.
  else return false;
}
```

3. Utiliser la notion de surcharge de constructeur pour construire un objet de la classe "Compte" des deux façons suivantes:

- les valeurs initiales Numero et Type d'un compte sont passées en paramètre; le solde est mis à 0. *Par exemple, C = new Compte (123, "J") créé le compte joint de numéro 123.*
- les valeurs initiales Numero et Type sont saisies au clavier; le solde est mis à 0. *Par exemple, l'instruction C = new Compte() demande à l'utilisateur de lui communiquer le numéro de compte et son type, et créé l'objet associé. On utilisera les méthodes Lire.i() et Lire.c() pour acquérir respectivement un entier et un caractère.*

Pour ces deux constructeurs, des contrôles sur le numéro (seules les valeurs positives sont autorisés) et sur le type, devront être mis en place.

```
public Compte (int N, char T)
{   if (N<=0)
        { System.out.println ("Le numéro de compte doit être positif");
          System.exit (1); }
    if (! TestType (T) )
        { System.out.println ("Le compte doit être de type J ou C ");
          System.exit (1); }
    else Type = T;
    Solde = 0;
}

public Compte ()
{   int N; char T;
    N = Lire.i();
    if (N<=0)
        { System.out.println ("Le numéro de compte doit être positif");
          System.exit (1); }
    else Numero = N;
    T = Lire.c();
    if (! TestType (T) )
        { System.out.println ("Le compte doit être de type J ou C");
          System.exit (1); }
    else Type = T;
    Solde = 0;
}
```

4. Ecrire les quatre accesseurs suivants:

accès en consultation:

```
public float Solde ()              // renvoie le solde d'un compte.
{ return (Solde); }

public int Numero ()               // renvoie le numéro d'un compte.
{ return (Numero); }

public char Type ()                // renvoie le type d'un compte.
{ return (Type); }
```

```
            public void ModifType ()   // modifie le type d'un compte automatiquement
                                       // en appliquant la logique suivante:
                                       // si c'est un compte courant, il devient compte joint;
                                       // si c'est un compte joint, il devient compte courant.
accès en    { char change;                         ou   { if (Type = = 'C') Type = 'J';
modification  change = Type;                              else Type = 'C';
              if (change = 'C') Type = 'J';             }
              else Type = 'C';
            }
```

5. Pour agir sur le solde d'un compte on souhaite disposer des traitements "Debiter" et "Crediter". Ecrire les méthodes associées.

```
public void Crediter (float S)        // crédite de la somme S, le solde d'un compte.
{ Solde += S; }

public void Debiter (float S)         // débite de la somme S, le solde d'un compte.
{ Solde -= S; }
```

6. Ecrire le programme de test de la classe « Compte» qui réalise les traitements suivants:

- Création des deux comptes suivants
  - Compte Cl   Numero: 250 Type : « C»
  - Compte C2   Numero: 320 Type : «J»
- Affichage du numéro et du solde du compte Cl
- Modification du type du compte C2
- Apport de 500 F sur le compte Cl
- Débit de 200 F sur le compte C2

```
class TestCompte
{ public static void main (String args[])
  { compte C1, C2;
    C1 = New Compte (250, "C");
    C2 = New Compte (320, "J");
    System.out.println ("Numéro du compte:" + C1.Numero ());
    System.out.println ("Solde du compte: " + C1.Solde () );
    C2.ModifType ();
    C1.Crediter (500);
    C2.Crediter (200);                 // on remarque que C2 est en solde négatif.
  }
}
```

7. On souhaite enrichir la classe "Compte" par la méthode binaire publique "EstSuperieur" dont la signature et l'objectif sont donnés ci-dessous:

```
public boolean EstSuperieur (Compte C)
{       // renvoie true si le solde de l'objet est supérieur au solde de C et false sinon
}
```

7.1 Ecrire la méthode « EstSupérieur»

```
public boolean EstSuperieur (Compte C)
{ if (C.Solde < Solde) return (true);
  else return (false);
}
```

7.2 Dans le programme de test précédent, comparer le compte "Cl" et le compte "C2" et faire afficher le message "Cl est supérieur à C2" ou "Cl est inférieur ou égal à C2" selon le résultat du test.

```
{ …
  if (C1.EstSuperieur (C2) ) System.out.println ("C1 est supérieur à C2");
  else System.out.println ("C1 est inférieur ou égal à C2");
}
```

4.3) Les méthodes de type binaire

A l'instar des opérateurs binaires, une méthode est binaire lorsqu'elle s'applique sur deux objets d'une même classe. *Par exemple, le traitement qui consiste à ajouter deux fractions donnera lieu à une méthode binaire.* Lors de la conception d'une telle méthode, les deux objets sur lesquels elle travaillera n'auront pas le même rôle. L'un sera l'objet sur lequel sera appliqué la méthode, et l'autre sera fourni en argument de la méthode. *L'exemple ci-dessous présente la méthode binaire « Multiplier » de la classe Fraction et son utilisation dans un programme de test.*

```
class Fraction
{  private int Numerateur ;
   private int Denominateur ;

   public Fraction Multiplier (Fraction F)
   {  Fraction Res;
      Res = new Fraction (F.Numerateur*Numerateur,F.Denominateur*Denominateur);
      return Res ;
   }
```

Remarque : dans la méthode d'une classe, on peut référencer les propriétés privées d'un autre objet de cette classe. Les références « F.Numerateur » et « F.Denominateur » sont donc autorisés dans l'écriture de la méthode « Ajouter ».

```
class Test
{   public static void main (String args[])
    {Fraction F1, F2 ;
     F1 = new Fraction (4,5) ;
     F2 = new Fraction (6,8) ;
     F3 = F1.Multiplier (F2) ;
    }
}
```

Application: On considère la classe "PuissanceDe10" :

Destinée à réaliser les opérations suivantes sur les puissances de 10 :

- Initialiser une puissance à partir de la donnée de l'exposant,
- Afficher une puissance de 10 sous la forme 10^Exposant
- Multiplier deux puissances de 10 (on rappelle que 10^4 x 10^5 donne 10^9
- Fournir la valeur associée à une puissance de 10 (rappel 10^0 = 1 et 10^{-2} = 0.01 (réel)

```
class PuissanceDe10
{   private int Exposant;
    public PuissanceDe10 (int E)                  // on ne met pas de type pour le constructeur.
    {   Exposant = E ;  }

    public void Afficher ()
    {   System.out.println ("10^" + Exposant) ;  }

    public void Multiplier (PuissanceDe10 P)
    {   Exposant += P.Exposant ;  }

    public PuissanceDe10 MultiplierBis (PuissanceDe10 P)
    {   PuissanceDe10 Result;
        Result = New PuissanceDe10 (P.Exposant + Exposant);
        Return (Result) ;  }

    public PuissanceDe10 MultiplierTer (PuissanceDe10 P)
    {   Exposant += P.Exposant ;
        return this ;                             // this retourne l'objet lui-même.
    }

    public void MultiplierQua (PuissanceDe10 P)
    {   P.Exposant += Exposant ;  }               // ici c'est P2 qui va être modifié.

    public float Valeur ()
    {   int P = 1, i ;
        for (i = 1; i <= Math.abs(Exposant); i++;) P = P*10 ;
        if (Exposant < 0) P = 1 / P ;
        return (P) ;
    }
}
```

Auto-évaluation n°7: On considère la classe « Capacité » :

Destinée à gérer des capacités exprimées en octets. Son implantation est fournie ci-dessous:

```
class Capacite
{ private float Valeur;
  private char Unite;
  public Capacite (float V, char U)
  { if (U != 'O' && U != 'K' && U !=' M')
     { System.out.println (" capacité incorrecte");
       System.exit(1);
     }
     else
     { Valeur = V; Unite = U;  }
  }

  private long ValeurEnOctets ()
  { long L = (long) Valeur;
     if (Unite = = 'K')   L = (long) Valeur* 1024;
     else  if(Unite = = 'M')  L = (long) Valeur * 1024*1024;
     return L;
  }

  public void Ajouter (Capacite C)
  { Valeur = this.ValeurEnOctets () + C.ValeurEnOctets ();
     Unite = 'O';
  }

  public void Afficher ()
  {  System.out.println (Valeur +" "+ Unite);  }
}
```

1. Fournir les résultats affichés à l'écran à l'issue de l'exécution du programme suivant:

```
class TestCapacite ()
{ public static void main (String args[])
  {  Capacite C1,C2;
     C1= new Capacite (10,'O');
     C2 = new Capacite (1,'K');
     Cl.Ajouter (C2);
     Cl.Afficher ();
     C2.Afficher();
}
```

2. Modifier la méthode « Ajouter» afin de consigner le résultat de l'addition non plus dans l'objet sur lequel elle est appliquée, mais dans l'objet passé en paramètre.

3. On décide de surcharger le constructeur avec la méthode suivante:

```
public Capacite (float V)
{ if ( V >= 1024*1024)
   {  V = V / (1024*1024);
      Unite = 'M';
   }
  else  if (V >=1024)
         {    V = V / 1024;
              Unite = 'K';
         }
        else  Unite = 'O';
  Valeur = V;
}
```

Fournir le contenu des variables et de l'objet créé à l'issue de l'exécution du programme suivant (la valeur 2048 est saisie au clavier):

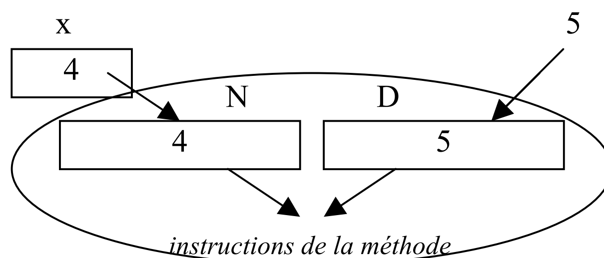
```
class TestCapacite 1
{ public static void main (String args[])
   {  Capacite C;
      float D;
      D = Lire.f();
      C = new Capacite (D);
   }
}
```

4.4) La transmission de paramètres

En Java, la transmission des paramètres se fait toujours par valeur. Cela signifie que lors de l'appel d'une fonction qui admet des paramètres d'entrée on devra communiquer pour chacun de ces paramètres une valeur. Celle-ci pourra être soit le contenu d'une variable, soit un littéral. En fait, on peut assimiler les paramètres formels à des variables locales qui reçoivent les valeurs des paramètres effectifs lors d'un appel. Le schéma suivant illustre ce principe sur le constructeur de la classe « Fraction »:

Programme d'utilisation
x=4
F = new Fraction(x,5)

Fonction Fraction (int N, intD)
{ Numerateur =N ;
Denominateur = D ;
}



De ce principe découle le fait que toute modification d'un paramètre formel (de type primitif), réalisée à l'intérieur d'une méthode, n'aura aucun effet sur une variable passée en paramètre effectif. *Cela signifie que si dans le constructeur de la classe « Fraction », une modification d'un paramètre d'entrée a été réalisée par erreur, la variable fournie en paramètre effectif ne sera pas modifiée.*

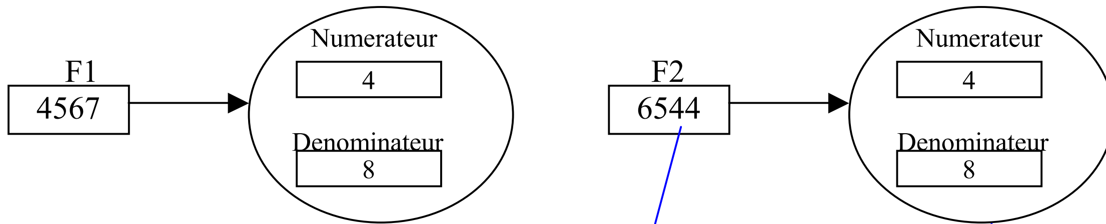
Fonction Fraction (int N, intD) *{ Numerateur =N ;* *N=N+1 ;* *Denominateur = D ;* *}*	*x = 4 ;* *F = new Fraction(x,5) ;* *Après exécution du constructeur, x contient toujours la valeur 4.*

Dans le cas où le paramètre formel est un objet, c'est la référence à l'objet qui est passée en paramètre effectif. Dans ce cas, toute modification réalisée sur l'objet via sa référence sera effectivement réalisée sur l'objet passé en paramètre. Le schéma ci-dessous illustre ce principe :

Considérons, pour la classe « Fraction », la méthode « Diviser » ci-dessous :

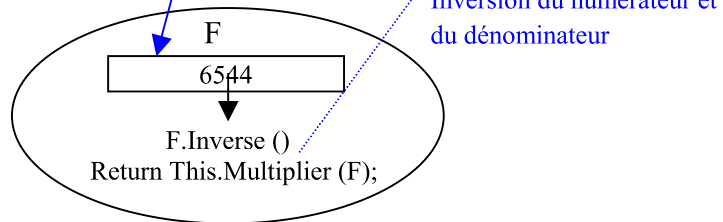
```
public Fraction Diviser (Fraction F)
{ F.Inverse() ;            // on inverse l'objet (division de fraction = multiplication par l'inverse)
  return this.Multiplier (F) ;     // le mot clé this désigne l'objet lui-même
}
```

et les deux objets « F1 » et « F2 » de la classe « Fraction » suivants :



L'appel de la méthode F1.Diviser(F2) (*par exemple F3= F1.Diviser(F2)*) va provoquer :

- l'affectation dans la variable locale associée au paramètre F de la valeur 6544.
- l'exécution des instructions de cette méthode avec cette valeur ce qui aura pour conséquence de modifier le numérateur et le dénominateur de l'objet pointé par F2.



On remarque que F2 n'est pas modifié mais c'est l'objet pointé qui est modifié.

4.5) Le mot clé « this »

Le mot clé « this » dans une méthode publique est une référence sur l'objet sur lequel est appliqué la méthode. Cette référence est en général implicite. *Par exemple pour définir l'accesseur en consultation chargé de retourner le numérateur on aurait pu écrire :*

```
public int SonNumerateur()
{       return this.Numerateur
}
```

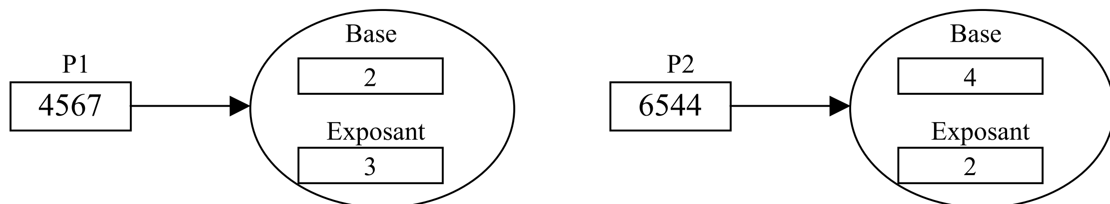
Ce mot clé peut par exemple être utilisé lorsqu'il y ambiguïté entre le nom d'un paramètre et le nom d'une variable d'instance. Le constructeur associé à la classe Fraction illustre ce cas:

```
public Fraction (int Numerateur, int Denominateur)
{       this.Numerateur = Numerateur ;
        this.Denominateur = Denominateur ;
}
```

La méthode « Diviser » présentée dans le paragraphe précédent présente un autre intérêt du mot clé « this ».

V. Les variables et méthodes de classe

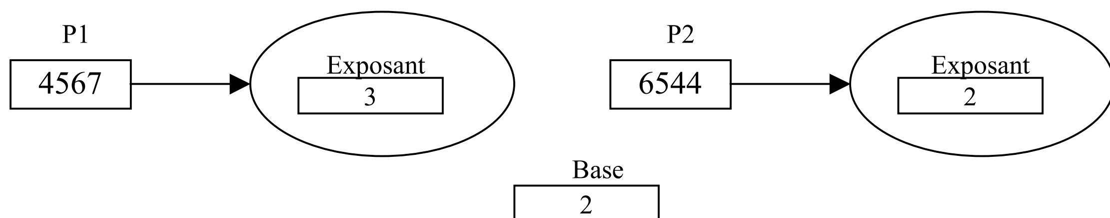
On suppose que l'on ait à construire une classe permettant d'effectuer des opérations sur des puissances ayant la même base. On rappelle que dans l'expression a^n, a est appelée base et n est appelé exposant. La solution qui consiste à définir pour la classe « Puissance », deux variables d'instance représentant respectivement la base et l'exposant n'est pas satisfaisante car aucun dispositif ne pourra être mis en place pour garantir que tous les objets de cette classe auront la même base. On pourra donc avoir la situation suivante qui empêche la réalisation de toute opération.



Pour pallier ce problème, on doit introduire une variable commune à tous les objets de la classe et qui mémorisera la base avec laquelle on souhaite travailler. Cette variable, appelée variable de classe, existe avant même qu'un objet soit instancié. Elle est déclarée au moyen du modificateur « static », et peut être privée ou publique. Avec le statut « Private », elle pourra être référencée uniquement par les méthodes de la classe dans laquelle elle est définie, alors qu'avec le statut « Public », elle pourra être référencée depuis une autre classe. Le début de la classe Puissance pourrait s'écrire ainsi :

```
class Puissance
{ private static int base = 2 ;
  private int exposant ;
  …
}
```

Compte tenu de l'initialisation de la variable « base », les objets de la classe « Puissance » représenteront des puissance de 2.



Pour permettre à l'utilisateur de modifier la variable « base », on doit définir la méthode de classe publique « ChoisirBase » de la manière suivante :

```
class Puissance
{ private static int base = 2 ;
  private int exposant ;
  public static void ChoisirBase ()     // on suppose que la fonction Lire.i()
  { base = Lire.i() ;  }                // permet d'acquérir au clavier un entier (int).
}
```

Celle-ci pourra être appelée, sans qu'aucun objet existe, en utilisant la référence: Puissance.ChoisirBase. Il est important de remarquer que l'exécution d'une méthode de classe pouvant être réalisée avant toute instanciation d'objet, il n'est pas sensé de référencer dans une méthode de classe une variable d'instance. Par exemple, le programme suivant n'aurait pas de sens :

```
class Puissance
{ private static int base = 2 ;
  private int exposant ;
  public static void ChoisirBase()
  { base = Lire.i() ;            // « exposant » est une variable d'instance dont
    exposant = Lire.i() ;  }     // l'existence est lier à l'existence d'un objet.
}
```

Par contre, il est possible de référencer dans une méthode d'instance, une variable de classe, comme l'illustre le programme suivant :

```
class Puissance
{ private static int base = 2 ;
  private int exposant ;
  public static void ChoisirBase()
  { base = Lire.i() ;  }
    exposant = Lire.i() ;  }
  public void Afficher ()
  { System.out.println (base + « ^ » + exposant) ;  }
}
```

VI. Travaux Pratiques

6.1) Utilisation de la classe String

Le langage Java admet les trois catégories suivantes de types primitifs :

- numérique : byte, short, int, long, float et double
- caractère : char
- booléen : boolean

Tous ces types primitifs permettent de manipuler directement des valeurs. Ainsi pour mémoriser un caractère, il suffira de l'affecter dans une variable de type caractère. Toutes les données qui ne sont pas d'un type primitif devront être mémorisées sous forme d'objets. C'est, par exemple, le cas des chaînes de caractères pour lesquelles Java propose les deux classes suivantes :

- « String » dont les objets sont des chaînes de caractères de longueur fixe,
- « StringBuffer » dont les objets sont des chaînes dont la longueur peut être modifiée.

A partir de la liste des méthodes donnée ci-dessous, et se rapportant à la classe « String », écrire le programme Java qui réalise les opérations suivantes :

- déclare les variables « Fichier » et « Extension » destinées à pointer sur des objets de la classe « String ».
- instancie un objet de la classe « String » avec la valeur « Test.xls ». Le constructeur admet l'interface suivante : public String (Chaîne). L'objet ainsi créé devra être pointé par la variable « Fichier ».
- extrait, dans l'objet pointé par la variable « Extension », l'extension contenue dans l'objet pointé par la variable « Fichier ».
- affiche par l'instruction System.out.println («l'extension est : » + Extension), l'extension ainsi récupérée.

Remarque : le nombre de caractères de l'extension est supposé quelconque

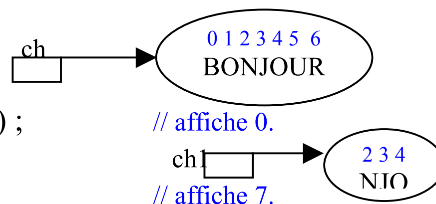
```
public class ManipChaine
{ public static void main (String args[])
   {  int Longueur, Point ;
      String Fichier, Extension ;
      Fichier = New String ('Test.xls') ;
      Point = Fichier.indexOf ('.') ;
      Longueur = Fichier.length () ;
      Extension = Fichier.Substring (Point + 1, Longueur -1) ;
      System.out.println ("L'extension est : " + Extension) ;
   }
}
```

Annexe : méthodes de la classe « String »

int indexOf(Caractère)	Localise le caractère fourni en paramètre dans l'objet et renvoie sa position. Le premier caractère a la position 0. La valeur -1 est renvoyée si le caractère ne se trouve pas dans l'objet chaîne.
String substring(Deb, Fin)	Extrait, dans l'objet chaîne, une sous-chaîne de la position "Deb" à la position "Fin".
int length()	Renvoie le nombre de caractères de l'objet chaîne

Exemple:

```
String ch, ch1;
ch = New String ('Bonjour') ;
System.out.println (ch.indexOf ('B') ) ;
ch1 = ch.Substring (2,4) ;
System.out.println (ch.length () );
```



6.2) Création d'une classe

Le programme de test suivant manipule des durées exprimées en heures et minutes.

Analyser les différentes méthodes référencées et le résultat de leur exécution afin de proposer une définition de la classe « Durée » cohérente avec son utilisation.

<table>
<tr><td><pre>public class TestDurée
{ public static void main (String [] arg)
    { Durée D1,D2 ;
       D1=new Durée (1,30) ;
       D1.Afficher ;
       D2 = new Durée (2,30) ;
       D1.Ajouter (D2) ;
       D1.Afficher ;
       System.out.println ("Valeur de la durée en minutes: "
+ D1.TotMinutes) ;
    }
}</pre></td><td>Résultat affiché à l'écran:

1 :30

4 :0
Valeur de la durée en minutes : 240</td></tr>
</table>

VII. Annexe : les étapes de développement en Java

Tout programme écrit en langage évolué doit être compilé afin d'être exécuté par l'ordinateur. Cette opération consiste plus précisément à traduire chaque instruction du programme en une suite d'éléments binaires que l'ordinateur saura interpréter et exécuter. *Par exemple à l'issue de la compilation d'un programme écrit en TURBOC C, on dispose d'un fichier exécutable portant l'extension .exe que l'on peut lancer à partir de l'environnement DOS par la commande : NomFichier*. Il est important de savoir que le code binaire est spécifique à chaque machine ; cela signifie que le fichier issu d'une compilation exécutée sur une machine de type PC n'est pas reconnu sur une station UNIX.

Pour remédier à ce problème et rendre le langage Java indépendant de l'ordinateur, le concepteur du langage Java (J. Gosling) a introduit une étape intermédiaire dans la compilation. Le principe consiste à traduire le programme écrit en java en un pseudo-code (appelé bytecode) indépendant de toute machine. Ce pseudo-code sera alors interprété et exécuté par un programme spécifique à chaque machine. L'ensemble des interpréteurs constitue la JVM : Java Virtual Machine.

Ce principe est illustré par le schéma suivant extrait de l'ouvrage « Le livre de Java premier langage » de Anne Tasso (éditions Eyrolles):

L'avantage d'un tel système est que le développeur est certain de créer un programme totalement compatible avec les différents ordinateurs du marché. Compte tenu du principe précédemment exposé, la boite à outils minimale pour réaliser des développements Java doit comporter les deux logiciels suivants :

- Le compilateur chargé de traduire le programme écrit en Java en pseudo-code: **javac Fichier.java** où le fichier ayant l'extension « java » contient la description d'une ou plusieurs classes. Le résultat est un fichier portant l'extension « class ».
- L'interpréteur, propre à chaque machine, et destiné à interpréter et exécuter le pseudo-code : **java Fichier**. Lors de l'exécution de ce fichier, l'interpréteur Java déclenche l'exécution de la méthode publique main() qui est le point d'entrée de l'application. Si dans cette méthode « main() », d'autres classes sont référencées, il doit disposer de leur définition pour réussir l'exécution

Ces deux outils sont disponibles dans le kit de développement Java (JDK : Java Development Kit ou SDK Standard Development Kit) téléchargeables depuis le site Internet de Sun. Il existe des logiciels tels que Kawa ou J buider qui offrent, sous forme d'interface graphique conviviale, un ensemble d'outils de développement. Il est important de savoir que ces derniers sont particulièrement gourmands en ressources machines.

PROGRAMMATION ORIENTEE OBJETS :

Les Collections JAVA

LES COLLECTIONS EN JAVA

I. INTRODUCTION 31
II. LES TABLEAUX 33
III. LES VECTEURS 36
3.1) La classe générique « Object » *36*
3.2) Les méthodes de la classe « Vector » *38*
3.3) La classe « Enumeration » *40*
3.4) Les packages *41*
IV. LES DICTIONNAIRES 42
V. LES FLUX 44
5.1) Opérations sur un fichier *44*
5.2) La gestion des erreurs *46*

INFORMATIQUE – CNAM ANGOULEME 2000-2001

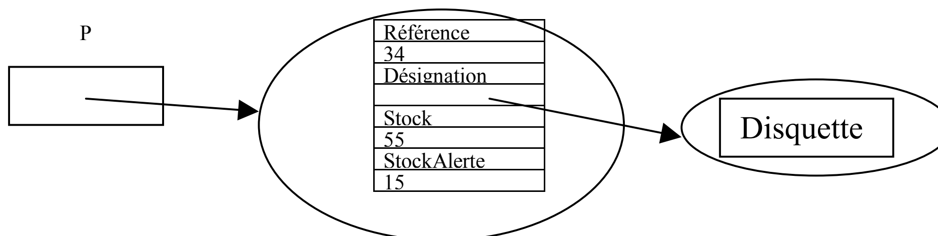
LES COLLECTIONS EN JAVA

I. INTRODUCTION

Le chapitre précédent a montré que l'accès à un objet d'une classe, nécessite l'existence d'une variable pointeur, déclarée en début de traitement de la manière suivante :

NomClasse NomVariablePointeur ;

Dans le schéma qui suit, la variable « P » permet d'accéder au produit de référence « 34 » et ainsi de lui appliquer différentes méthodes par la référence P.NomMéthode.



Avec un tel principe l'accès à n instances d'une classe, implique la déclaration de n variables pointeurs de cette même classe ce qui va entraîner :

- d'une part l'impossibilité d'utiliser des structures itératives pour manipuler ces objets du fait des noms de variables différents,
- d'autre part le risque d'un nombre insuffisant de variables pour des problèmes dont on ne connaît pas à priori le nombre d'instances créées.

Application: On considère la classe « Produit » définie partiellement ci-dessous :

```
class Produit
{ private int Reference ;
  private String Designation ;
  private int Stock ;
  private int StockAlerte ;

public Produit () ;   // demande toutes les caractéristiques d'un produit et créé l'objet associé
{ System.out.print ("Référence du produit: ");
  Reference = Lire.i();
  System.out.print ("Désignation du produit: ");
  Designation = Lire.S();
  System.out.print ("Stock: ");
  Stock = Lire.i();
  System.out.print ("Stock alerte: ");
  StockAlerte = Lire.i();   }
```

```
public boolean ACommander ()
{                    // Renvoie true si le stock est inférieur au stock alerte et false sinon
  return (Stock < StocIcAlerte);
}
public int SaRéférence ()
{                    // Renvoie la référence du produit
  return this.Reference;
}
public int SonStock ()
{                    // Renvoie le stock du produit
  return this.Stock;
}
```

On suppose que les variables « P1 », « P2 » et « P3 » pointent sur 3 instances de la classe « Produit ». Ecrire les instructions qui affichent, parmi ces trois produits, la référence de ceux qui sont à commander.

```
class TestProduit () ;
{ public void static main (String args[])
  { Produit P1, P2, P3;
    P1 = new Produit();
    P2 = new Produit();
    P3 = new Produit();
    if (P1.Acommander()) System.out.print (P1.SaRéférence());
    if (P2.Acommander()) System.out.print (P2.SaRéférence());
    if (P3.Acommander()) System.out.print (P3.SaRéférence());
  }
}
```

Pour faciliter la manipulation de plusieurs objets d'une même classe, le langage Java propose différents concepts dont les caractéristiques de stockage sont récapitulées ci-dessous :

Nom concept	Type du stockage	Nombre d'objets collectionnés
Tableau	Volatile	Fixe
Vecteur	Volatile	Indéterminé
Dictionnaire (ou table de hachage)	Volatile	Indéterminé
Flux (fichiers "normaux")	Permanent	Indéterminé

II. LES TABLEAUX

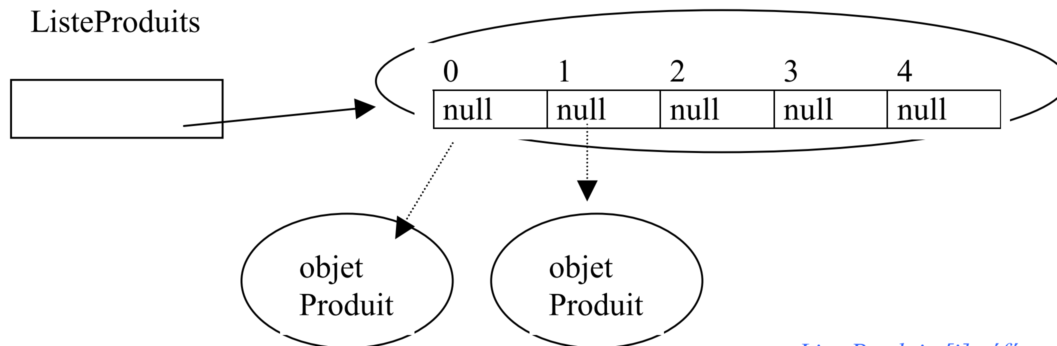
Pour créer un tableau destiné à mémoriser N pointeurs sur des objets d'une classe il faut procéder aux deux étapes suivantes :

- déclarer une variable qui sera destinée à pointer sur le tableau par la syntaxe suivante :
  NomClasse [] NomVariable
- écrire l'instruction chargée de créer « l'objet tableau » par la syntaxe :
  NomVariable = new NomClasse[N]

A l'issue de cette déclaration, la variable spécifiée dans le membre gauche de l'affectation pointe sur un objet contenant N « cases », destinées chacune à recevoir l'adresse d'un objet de la classe. *Par exemple, pour créer un tableau pointant sur 5 produits on devra écrire les instructions suivantes :*

Produit[] ListeProduits ;
ListeProduits = new Produit[5] ;

Le contenu des variables après l'exécution de ces instructions pourrait être représenté ainsi :



ListeProduits[i] référence l'adresse

Il est important de savoir que la taille d'un tableau, définie lors de l'exécution par l'opérateur « new », ne peut plus être modifiée ultérieurement. Il est, par contre, possible d'acquérir, lors de l'exécution, la taille du tableau souhaité et de communiquer cette valeur à l'opérateur « new » comme l'illustre l'exemple suivant :

int N ;
Produit[] ListeProduits ;
System.out.println (« Nombre maximal de produits ? ») ;
N=Lire.i() ;
ListeProduits = new Produit[N] ;

Les opérations que l'on peut mettre en œuvre sur un tableau se répartissent en 2 catégories :

- la première catégorie regroupe les traitements qui agissent sur le tableau dans son intégralité. *Par exemple l'instruction ListeProduits.length renvoie le nombre de cases du tableau.* « length » est en fait une propriété publique de l'objet « ListeProduits ».
- la seconde catégorie concerne les traitement qui agissent sur chaque composant d'un tableau. Pour la programmation de ces traitements, il faut savoir que l'accès au ième élément du tableau est réalisé par la référence : NomVariableTableau[i] (le premier élément porte le numéro 0).

Remarquons enfin que les tableaux permettent de mémoriser non seulement des adresses d'objets mais aussi des valeurs de type primitif.

Application 1: On considère que le tableau ListeProduits est valorisé :

et pointe sur 5 objets de la classe « Produits ». Réécrire le traitement qui consiste à afficher la référence des produits à commander.

```
class TestProduit () ;
{ public void static main (String args[])
  { Produit [ ] ListeProduits;
    int i = 0;
    ListeProduits = new Produit [5];
  …                                                // Chargement du tableau
    for (i = 0; i ≤ 4, i ++)
        { if (ListeProduits[i].Acommander())
              System.out.println(ListeProduits[i].SaRéférence());
        }
// ou                              propriété publique
//  for (i = 0; i ≤ ListeProduits.length - 1; i ++)
//      { if (ListeProduits[i].Acommander())        méthode
//            System.out.println(ListeProduits[i].SaRéférence());
//      }
}
```

Application 2: On souhaite construire la classe « Inventaire » :

destinée à gérer l'ensemble des produits proposés par une entreprise. La description du début de cette classe a la forme suivante :

```
class Inventaire
{ private Produit [] ListeProduits ;
  private int NbProduits = 0;
  public Inventaire()
  { ListeProduits = new Produit[100] ;  }
}
```

<u>a) Ecrire les trois méthodes suivantes :</u>

- **AjouterUnProduit ()** qui doit provoquer la création d'un objet de la classe « Produit » en demandant à l'utilisateur les différentes caractéristiques du produit.
- **RechercherUnProduit (int RefProduit)** qui renvoie le produit dont la référence est spécifiée en paramètre d'entrée. La valeur "null" est renvoyée si le produit n'existe pas.
- **AfficherProduitsACommander()** qui affiche la référence de tous les produits dont le stock est inférieur au stock alerte.

```
class Inventaire
{ private Produit [] ListeProduits;
   private int NbProduits = 0;

  public Inventaire()
  { ListeProduits = new Produit [100];  }

  public void AjouterunProduit()
  { ListeProduits[NbProduits] = new Produit;
    NbProduits ++;
  }
```

```
public Produit RechercherUnProduit (int RefProduit)
{ int i=0;
  Produit tmp=null;
  boolean Trouve = false;
  while (! Trouve && i <=NbProduits-1)
  if (RefProduit = = ListeProduits[i].SaReference())
       { Trouve = true;
         tmp = ListeProduits[i];
       }
  else  i ++;
  return tmp;
}

public void AfficherProduitsACommander()
{ int i;
  for (i=0; i<= NbProduits - i; i ++)
       if (ListeProduits[i].Acommander())
         System.out.print (ListeProduits[i].SaReference());
}
}
```

b) Proposer un programme de test qui réalise les opérations suivantes :

- crée un objet de la classe « Inventaire ».
- ajoute dans la classe les produits suivants (une structure itérative à mettre en place)

Référence	Désignation	Stock	StockAlerte
34	Disquettes	55	15
56	Cartouche encre	5	10
72	ZIP	4	5

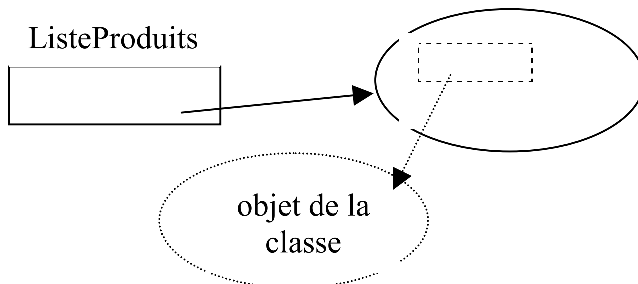
- recherche un produit dont la référence est saisie. Si le produit existe, son stock sera affiché, sinon le message « référence inexistante » sera affiché.
- affiche la référence de tous les produits à commander.

```
class TestInventaire
{ public static void main (String args[])
  { Inventaire INV; int R; char Rep;
    INV= new Inventaire();
    do
    { INV.AjouterunProduit();
      System.out.println ("Autre produit (O/N)?");
      Rep = Lire.c();
    } while (Rep == 'O');
    System.out.println ("Saisir une référence de produit");
    R = Lire.i();
    if (INV.RechercherunProduit(R) != null)
         System.out.println (INV.RechercherunProduit(R).SonStock());
    else
         System.out.println ("Pas de produit pour cette référence");
    System.out.println ("Liste des produits à commander");
    INV.AfficherProduitsACommander();
  }
}
```

III. LES VECTEURS

Les vecteurs ont été introduits afin de remédier à l'impossibilité de redimensionner un tableau une fois que celui-ci a été créé par l'opérateur « new ». Ainsi, les objets de la classe « Inventaire » présentée précédemment, sont limités à la mémorisation de 100 produits. La classe « Vector » est prédéfinie (comme la classe String), et permet de gérer (en mémoire centrale) un nombre indéterminé d'objets, qui appartiennent à la classe générique « Object ». *Par exemple, le programme suivant crée un vecteur qui sera destiné à pointer sur un ensemble d'objets de la classe « Object ». Seul le nom attribué à ce vecteur laisse penser que ce sont des objets de la classe « Produit » qui seront mémorisés dans cette structure.* Au départ, aucune case n'est affectée au vecteur et c'est au fur et à mesure des ajouts que le vecteur va s'agrandir. Les vecteurs ne sont pas typés, au départ, on ne sait donc pas de quels objets il s'agit. En fait, on utilise une classe "générique".

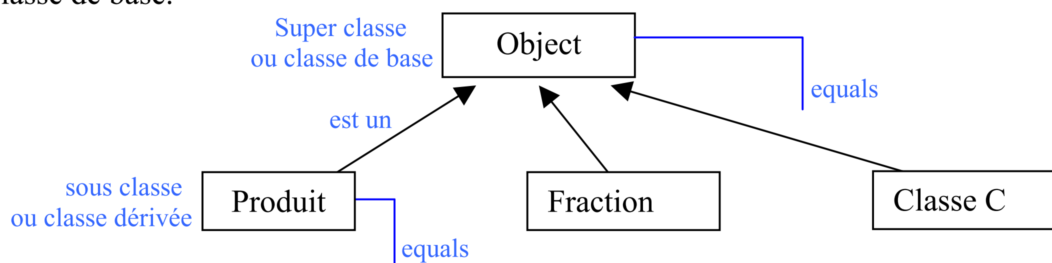
Vector ListeProduits ;
ListeProduits = new Vector() ;



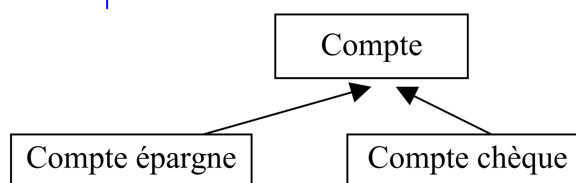
3.1) La classe générique « Object »

ou une première approche de l'héritage

Tout objet issu d'une classe est un objet de la classe « Object » et à ce titre, hérite des méthodes publiques de cette classe. Cette relation « est un » peut être schématisée ainsi et montre que les objets des classes « Fraction »,. « Produit » ou d'une classe quelconque « C » sont des objets de la classe « Object ». On dit que les classes « Produit », « Fraction » et « C » sont des sous-classes de « Object » ou encore des classes dérivées de la classe « Object ». La classe « Object » est une super-classe, ou c'est la classe de toutes les classes, ou encore une classe de base.



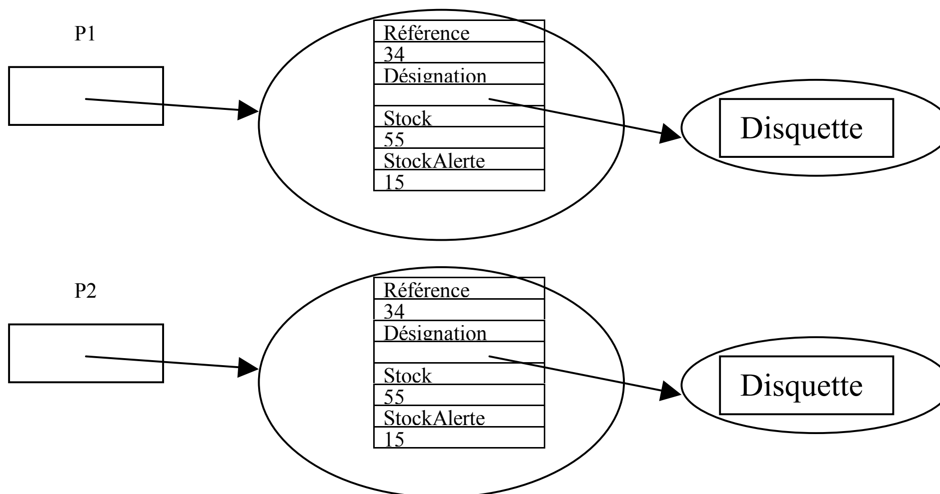
Exemple:



Compte tenu de cette hiérarchie sur un produit de la classe « Produit » on peut appliquer les méthodes de la classe « Produit » mais aussi les méthodes de la classe « Object ». *Par exemple la méthode «clone()» de la classe « Object » qui permet de créer le clone d'un objet peut être appliqué aussi bien à un objet de la classe « Produit » qu'à un objet de la classe « Fraction ».*

```
Fraction F1,F2 ;
Produit P1, P2 ;
P1 = new Produit() ;
P2 = P1.clone() ;          // Deux fractions identiques et non pas une seule fraction
F1 = new Fraction(4,5) ;
F2 = F1.clone() ;          // car Fraction "hérite" la méthode clone() de Object.
```

Il est possible de redéfinir dans une sous-classe une méthode de la classe générique. Dans ce cas, on dit que la méthode est polymorpohe c'est-à-dire a plusieurs formes. A chaque appel d'une telle méthode, la forme à exécuter est choisie en fonction de l'objet associé à l'appel. *Considérons par exemple la méthode « equals » de la classe « Object ». Cette méthode permet de comparer les adresses de deux objets et renvoie true si ces adresses sont égales et false sinon. Ainsi dans l'exemple qui suit P1.equals(P2) renvoie false car P1 et P2 contiennent des adresses différentes.*



On peut redéfinir la méthode « equals » au niveau de la classe « Produit » afin de renvoyer « true » si deux objets distincts contiennent les mêmes valeurs. A l'issue de cette conception, la méthode « equals » a deux formes et le programme suivant montre la puissance du polymorphisme :

```
Produit P1,P2 ;
Fraction F1,F2 ;
…
if (P1.equals(P2))      // c'est la méthode « equals » de la classe « Produit » qui est exécutée
    System.out.println (« P1 et P2 sont deux objets différents mais contiennent les mêmes valeurs ») ;
if (F1.equals(F2))      // c'est la méthode « equals » de la classe « Object » qui est exécutée
        System.out.println (« F1 et F2 désignent le même objet ») ;
```

Cette hiérarchie entre classes induit aussi la nécessité d'effectuer des transtypages lors d'affectation entre les variables de ces différentes classes. Le programme ci-dessous présente ce principe pour les classes « Object » et « Produit » ainsi que le mécanisme du polymorphisme issu du transtypage.

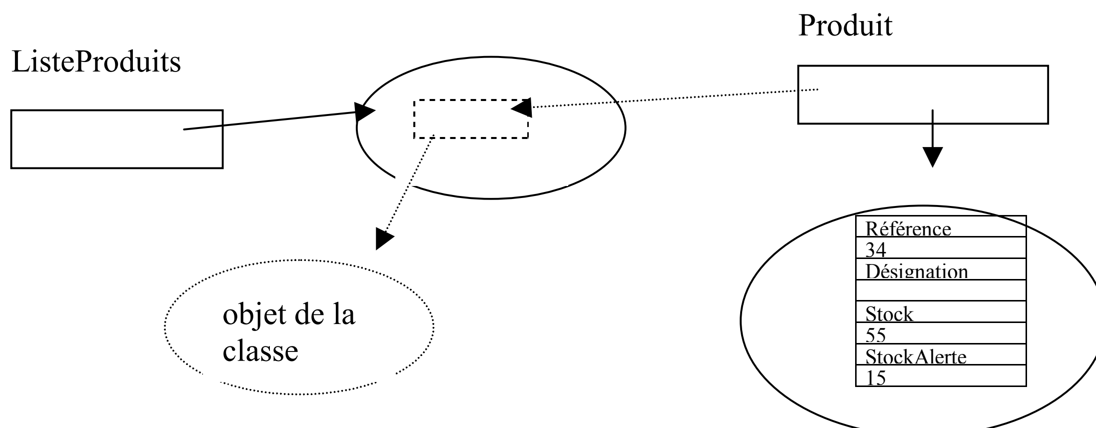
```
Object O1, O2 ;
Produit P1, P2 ;
....
O1 = P1;            // un produit est un objet donc l'affectation est possible et le transtypage est implicite
O2 = P2 ;
P1 = (Produit) O1 ; // un objet n'est pas forcément un produit donc il faut spécifier explicitement un cast
                    //pour effectuer le transtypage ; si le cast n'est pas spécifié, le compilateur refusera
                    //l'affectation car un objet n'est pas forcement un produit. A l'exécution, il peut y
                    //avoir une erreur si la variable « O1 » n'est pas un objet de la classe Produit.
if (O1.equals(O2))  //c'est la méthode equals de la classe « Produit » qui est exécutée car les objets
                    //« O1 » et « O2 » appartiennent à la classe « Produit »
```

Même si la notion de polymorphisme sera reprise et détaillée dans le chapitre suivant, il est important de retenir d'ores et déjà les éléments suivants (extraits de l'ouvrage "Java, la synthèse") :

- le polymorphisme est la capacité pour une même méthode de correspondre à plusieurs formes de traitement, selon les objets auxquels la méthode est appliquée,
- une telle facilité, qui permet le choix dynamique entre plusieurs formes de traitement suppose que le choix de la méthode soit fait au moment de l'exécution. Dans l'instruction précédente (if O1.equals(O2)), le choix de la forme de méthode « equals » ne peut pas se faire à la compilation car on ne sait pas sur quel type d'objet les variables « O1 » et « O2 » pointent.

3.2) Les méthodes de la classe « Vector »

Rappelons que chaque composant d'un vecteur est un pointeur sur un objet de la classe « Object ». Compte tenu de ce qui vient d'être dit, ce pointeur peut adresser n'importe quel objet d'une classe car « Object » est la super-classe de toutes les classes. Le schéma suivant illustre ce principe basé sur un transtypage implicite.



Quelques unes des méthodes de la classe "Vector" sont présentées et commentées ci-dessous:

Nom de la méthode	**Rôle**
add(objet)	Ajoute un élément objet en fin de liste
add(indice, objet)	Ajoute un objet dans la liste à l'indice spécifié en paramètre
elementAt(indice)	Retourne l'objet stocké à l'indice spécifié en paramètre. Le premier élément porte le numéro 0.
clear()	Supprime tous les éléments de la liste
IndexOf (Objet)	Retourne l'indice dans la liste du premier objet donné en paramètre ou –1 dans la liste
remove(indice)	Supprime l'objet dont l'indice est spécifié en paramètre
setElementAt(Objet,i)	Remplace l'élément situé en position i par l'objet spécifié en paramètre
size()	Retourne le nombre d'éléments placés dans la liste

Ainsi, pour ajouter un produit dans le vecteur « ListeProduits », il suffira d'écrire l'instruction : ListeProduits.add(new Produit()) . L'appel de cette méthode induit un transtypage automatique entre une variable de classe « Object » et une variable de la classe « Produit ». Un transtypage explicite devra être par contre mis en place pour récupérer dans la variable pointeur P de la classe « Produit », l'élément n° i du vecteur ListeProduits.

P=(Produit) ListeProduits.elementAt(i)

Il faut souligner que la classe « Vector » ne permet pas de mémoriser des valeurs de type primitif. Pour remédier à ce problème, le langage « Java » a défini une classe pour chacun des types primitifs. *Par exemple la classe « Int » est associé au type « int » (le nom de la classe est obtenu en écrivant avec la première lettre majuscule, le nom du type primitif)*. Chacune de ces classes possède un constructeur qui instancie un objet à partir du type primitif. Les instructions suivantes créent un objet de la classe « Int » qui pourra ainsi être mémorisé par un vecteur.

```
int i = 4 ;
Int ObjetEntier ;
ObjetEntier = new Int(i) ;
```

Application: On suppose que l'on modifie la classe « Inventaire »:

de la manière suivante (la partie données ainsi que le constructeur) :

```
class Inventaire
{ private Vector ListeProduits ;
  public Inventaire()
  ListeProduits = new Vector() ;
}
```

Réécrire les méthodes AjouterUnProduit, RechercherUnProduit, AfficherProduitsACommander

```
import java.util.Vector;
class Inventaire
{ Vector ListeProduits;
  public Inventaire()
    { ListeProduits = new Vector();  }

  public void AjouterunProduit()
    { ListeProduits.add (new Produit());  }
```

```
    public Produit RechercherUnProduit (int RefProduit)
    { int i=0;
      Produit tmp=null;
      boolean Trouve = false;
      while (! Trouve && i <=ListeProduits.size()- 1)
      if (RefProduit = = ((Produit) ListeProduits.elementAt(i)).SaReference())
            { Trouve = true;
              tmp = (Produit) ListeProduits.elementAt(i);
            }
      else  i ++;
      return tmp;
    }

  public void AfficherProduitsACommander()
    { int i;
      for (i=0; i <= ListeProduits.size()- 1; i ++)
            if (((Produit) ListeProduits.elementAt(i)).ACommander())
              System.out.print (((Produit) ListeProduits.elementAt(i)).SaReference());
    }

  }
```

3.3) La classe « Enumeration »

ou une approche de la notion d'interface

L'intégration, dans une structure itérative de type « for », de la méthode « elementAt() » de la classe « Vector » permet de passer de revue tous les objets contenus dans un vecteur. *Par exemple l'instruction suivante affiche la référence de tous les produits contenus dans le vecteur ListeProduits.*

```
  for (i = 0 ; i <= ListeProduits.size() - 1 ; i++)
      System.out. println (((Produit)ListeProduits.elementAt(I)).SaReference()) ;
```

Une autre possibilité pour parcourir un vecteur consiste à utiliser la classe « Enumeration » . Cette classe a les particularités suivantes :

- elle n'admet pas de données ; elle n'est donc pas instanciable et l'on ne pourra donc jamais créer un objet de cette classe,
- elle contient uniquement des signatures de méthodes. L'implantation des différentes méthodes étant réalisée dans toutes les classes qui sont rattachées à cette classe particulière.

Une telle classe est appelée interface et sa définition est donnée ci-dessous :

```
public interface Enumeration
{ boolean hasMoreElements() ;
  Object nextElement() ;
}
```

~~E = new Enumeration();~~ // impossible !

Les méthodes de cette classe repose sur l'existence d'un curseur qui pointe à tout moment sur un composant particulier de l'énumération. La méthode « nextElement » renvoie l'objet pointé par le curseur et déplace le curseur sur le composant suivant. La méthode « hasMoreElement » renvoie la valeur « true » si le curseur pointe sur un élément et « false » si la fin de l'énumération est atteinte.

C'est en appliquant la méthode « elements() » sur un objet de la classe « Vector » que l'on peut obtenir un objet de la classe énumération et donc utiliser ses facilités de parcours. *L'exemple suivant illustre l'emploi des méthodes de la classe « Enumeration» pour afficher toutes les références des produits contenus dans le vecteur « ListeProduits ».*

```
Enumeration E ;
E=ListeProduits.elements()
while (E.hasMoreElements())
    System.out. println (((Produit) E.nextElement()).SaReference())
```

Produit

Application: Réécrire la méthode RechercherUnProduit() :
de la classe inventaire en utilisant l'interface « Enumeration ».

```
public Produit RechercherUnProduitBis (int RefProduit)
{ Produit tmp=null;
  boolean Trouve = false;
  Enumeration E;
  E = Listeproduits.elements();
  while (! Trouve && E.hasMoreElements())
  { tmp = (Produit) E.nextElement();
    if (RefProduit = = tmp.SaReference())
      { Trouve = true;  }
  }
  return tmp;
}
```

3.4) Les packages

Pour pouvoir référencer la classe « Vector » dans un programme, il est nécessaire de spécifier l'instruction d'importation « import java.util.Vector » qui indique au compilateur et à l'interpréteur la localisation de cette classe dans l'organisation fonctionnelle des classe java. De même l'utilisation de la classe « Enumeration » n'est possible qu'en spécifiant en début de programme l'instruction « import java.util.Enumeration ». Cette organisation repose sur la notion de package dont les grands principes sont donnés dans ce qui suit.

Un package est un regroupement (au sein d'un répertoire) de plusieurs classes qui ont un dénominateur fonctionnel commun. Les classes regroupées dans la package « java.lang » sont directement référençables depuis n'importe quelle autre classe sans avoir à préciser quoi que ce soit dans le programme appelant. *C'est par exemple le cas de la classe « String » qui a été utilisée dans la première partie des travaux pratiques du chapitre précédent.* Pour une classe

appartenant à un package autre que « java.lang », son niveau d'utilisation dépend de son modificateur d'accès :

- public : la classe est accessible depuis n'importe quelle autre classe appartenant au package ou non
- pas de modificateur : la classe est accessible seulement depuis les classes appartenant au même package.

Pour accéder à une classe d'un package ayant le spécificateur public il est nécessaire de mentionner, dans le programme d'utilisation, l'instruction : *import NomPackage.NomClasse*

*Ainsi pour utiliser les classes « Vector » et « Enumeration » qui se trouvent dans le package « java.util », on écrit en début de traitement : import java.util.**

Pour intégrer une classe à un package, il est nécessaire de respecter la syntaxe ci-dessous :

```
package NomPackage ;
public class NomClasse
{….
}
```

<u>Quelques remarques :</u>

- Une classe ne peut appartenir qu'à un seul package.
- Dans le cas où l'on ne précise pas la clause package lors de la création d'une classe, un package par défaut, qui correspond au répertoire courant, lui est automatiquement attribué.
- Dans certains systèmes, la variable système « CLASSPATH » permet de définir tous les répertoires susceptibles de contenir des packages. Ces répertoires sont alors visités par le compilateur pour qu'il procède aux importations demandées. L'affectation de différents chemins d'accès à la variable CLASSPATH est réalisée par la commande « SET ». Par exemple, la commande suivante définit comme localisation possible des packages le répertoire courant et le répertoire « Lib ».

```
SET CLASSPATH = . ;C:\JAVA\JDK1.3\Lib
```

IV. Les dictionnaires

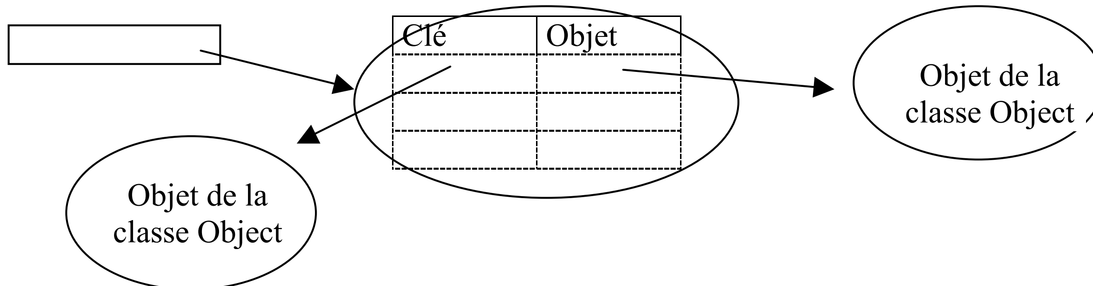
Dans un vecteur, l'accès direct à un composant n'est possible que si l'on connaît son indice. Si on ne détient pas cette information on doit procéder à une recherche séquentielle. Ainsi la méthode RechercherUnProduit (Refproduit) met en œuvre un parcours séquentiel du vecteur « ListeProduits » afin de trouver le produit dont la référence est fournie en paramètre.

Les objets de la classe « Hashtable » encore appelés «dictionnaires » permettent de mémoriser un nombre indéterminé d'objets d'une classe et mettent à la disposition de l'utilisateur des méthodes capables de réaliser des accès directs à partir d'une valeur de clé. Une clé est une donnée de la classe qui est telle que tous les objets de cette classe ont des valeurs différentes. Elle permet donc d'identifier chaque instance de la classe. *Dans la classe « Produit » la donnée « Reference » peut être considérée comme une clé.*

Pour créer un dictionnaire, il faut écrire les deux instructions suivantes :

```
Hashtable NomVariable ;
NomVariable= new Hashtable()
```

A l'issue de l'exécution de ces instructions la structure du dictionnaire pourrait être schématisée ainsi :



Les méthodes de la classe « Hashtable » les plus couramment utilisées sont décrites et commentées dans le tableau ci-dessous :

Nom de la méthode	**Rôle**
put(objetClé,Objet)	Ajoute une clé et son objet dans le dictionnaire
get(objetClé)	Renvoie l'objet associé à la clé spécifiée en paramètre
remove(ObjetClé)	Supprime dans le dictionnaire l'élément ayant la clé spécifiée en paramètre
size()	Retourne le nombre d'éléments placés dans la liste
keys ()	Renvoie sous forme d'énumération la liste des clés mémorisées.

Application: On souhaite implanter la classe « Inventaire »
destinée à mémoriser un ensemble de produits au moyen d'un dictionnaire. En supposant que tout produit est identifié par sa référence, le début de description de la classe « Inventaire » a la forme suivante :

```
import java.util.* ;
public class Inventaire
{ private Hashtable ListeProduits ;
  public Inventaire()
  { ListeProduits = new Hashtable() ;  }
```

1. Réécrire les méthodes AjouterUnProduit et RechercherUnProduit
2. Ecrire la méthode AfficherTout qui affiche la liste de tous les produits (réf et désignation).

V. Les flux

5.1) Opérations sur un fichier

Jusqu'à maintenant les objets créés étaient stockés dans la mémoire vive de l'ordinateur et avaient de ce fait une durée de vie limitée à l'exécution du programme. Pour mémoriser de façon permanente des objets, le langage Java propose les deux mécanismes suivants :

- la sérialisation qui consiste à écrire des informations relatives à un objet sur un fichier,
- et la désérialisation qui correspond à l'opération réciproque c'est-à-dire la reconstitution de l'objet en mémoire vive à partir des informations lues sur le fichier.

La mise en place de ces deux mécanismes est basée sur l'utilisation des classes « ObjectOutputStream » et « ObjectInputStream ». Un objet de la classe « ObjectOutputStream » est appelé flux sortant et sera utile pour effectuer une opération d'écriture sur un fichier. Un objet de la classe « ObjectOutpuStream » est appelé flux entrant et sera utile pour réaliser une opération de lecture sur un fichier. L'extrait de programme suivant illustre la création de ces deux objets.

```
ObjectInputStream FE ;
ObjectOutputStream FS ;
FE = new ObjectInputStream (new FileInputStream(« Produits.DTA ») ;
FS = new ObjectOutputStream (new FileOutputStream (« Produits.DTA ») ;
```

Remarques :

- la création d'un objet de type flux entrant échoue si le fichier spécifié en paramètre n'existe pas dans l'arborescence. Dans ce cas, une erreur de type "File Not Found Exception" est émise. Dans le cas contraire, « la tête de lecture » associé à ce flux se positionne sur le premier objet stocké.
- la création d'un objet de type flux sortant échoue seulement dans le cas où le chemin d'accès au fichier, spécifié en paramètre, n'est pas cohérent avec l'arborescence. Dans le cas où le fichier existe déjà son contenu est effacé. Dans le cas où le fichier n'existe pas et que son chemin d'accès est correct, il y a création du fichier. Pour toutes ces situations « la tête d'écriture » se positionne en début de fichier.
- une fois que tous les traitements ont été réalisés, il faut fermer les flux en leur appliquant la méthode « close() ». Par exemple FE.close() ferme le flux « entrant » pointé par la variable « FE ».

Les objets que l'on peut écrire ou lire sur un fichier peuvent être issues de n'importe quelle classe : prédéfinie (« Vector », « Hastable », « String », etc) ou non. Dans le cas d'une classe créée par le programmeur, il faudra rendre ses objets sérialisables en précisant dans l'en-tête de la classe la clause « implements Serializable ». Par exemple, pour enregistrer de manière permanente les objets de la classe « Produit » on devra préciser la clause « implements » de la manière suivante .

```
public class Produit  implements Serializable
{…}
```

La lecture d'un objet sur un fichier, est réalisé en appliquant la méthode « readObject() » à l'objet « Flux entrant ». L'objet pointé par la tête de lecture est alors transféré en mémoire centrale et la tête progresse automatiquement sur l'objet suivant. Dans le programme suivant, la variable P pointe sur l'objet à l'issue de la lecture sur le fichier

```
Produit P ;
P = (Produit) FE.readObject() ;
```

Remarques :
- le cast est nécessaire pour effectuer le transtypage de l'objet lu.
- dans le cas où la tête de lecture pointe aucun objet, la méthode « readObject » émet l'erreur « ClassNotFoundException ».

Pour écrire un objet sur un fichier, il suffit d'appliquer la méthode « writeObject » à l'objet « Flux sortant ». L'objet, présent en mémoire vive, est alors écrit sur le fichier dans l'emplacement pointé par la tête d'écriture et celle-ci avance automatiquement sur le prochain emplacement libre.

```
Produit P ;
P = new Produit() ;
FS.writeObject(P) ;
```

Application: La classe ProdCatalogue

Dont un extrait est présenté ci-dessous est destinée à mémoriser les produits proposés à la vente par la société X. Tous ces produits étant soumis au même taux de T.V.A., celui-ci est stocké en un exemplaire dans la variable de classe TauxTVA. Pour mémoriser de façon permanente ce taux et prendre en compte d'éventuelles modifications, celui-ci est enregistré dans le fichier « Param.DTA ».

```
class ProdCatalogue
{public static float TauxTVA ;
private static ObjectInputStream FE;
private static ObjectOutputStream FS;
private int Reference ;
private String Designation ;
private PrixHT ;
```

1. Ecrire les méthodes de classe LireTaux, ModifierTaux et EnregistrerTaux définies ci-dessous :
   – La méthode LireTaux lit dans le fichier « Param.DTA » le taux de TVA enregistré et le stocke dans la variable de classe TauxTVA
   – La méthode ModifierTaux (float NouveauTaux) modifie le taux en vigueur en lui affectant la valeur fournie en paramètre
   – La méthode EnregistrerTaux enregistre dans le fichier « Param.DTA » le taux en vigueur

2.Ecrire le constructeur de la classe « ProdCatalogue » destiné à créer un produit du catalogue en acquérant ses différentes caractéristiques par l'intermédiaire du clavier.

3. Ecrire la méthode AfficherPrixTTC qui affiche le prix TTC du produit.

5.2) La gestion des erreurs

Comme il a été dit dans le paragraphe précédent certaines opérations sur les fichiers peuvent conduire à des erreurs d'exécution. Par exemple, la création d'un flux entrant provoque une erreur si le fichier que l'on veut lire n'existe pas dans l'arborescence. Chacune de ces erreurs, encore appelées exceptions, correspond à un objet d'une classe particulière qui est instancié lors de l'incident. Ainsi, si l'on suppose que le fichier « Param.DTA » n'existe pas, l'exécution de l'instruction :

FE = new ObjectInputStream (new FileInputStream(« Param.DTA ») ;

provoque l'instanciation d'un objet de la classe « FileNotFoundException » (on dit encore qu'une exception de la classe « FileNotFoundException » est levée). A l'issue de cette instanciation, le traitement en cours est interrompu et l'exception est propagée aux méthodes appelantes. Si aucune méthode ne traite l'erreur, l'exécution se termine avec un ou plusieurs messages d'erreurs.

Toute méthode référençant des méthodes pouvant lever des exceptions telles que « readObject » ou le constructeur d'un flux doit avertir le compilateur de ce fait en précisant la clause supplémentaire « trhows liste des classes d'exceptions ». Par exemple, l'en-tête de la méthode « LireTaux » de la classe « ProdCatalogue » devra, pour être acceptée par le compilateur avoir la forme ci-après :

```
public static void LireTaux ()
throws IOException, ClassNotFoundException
{…
}
```

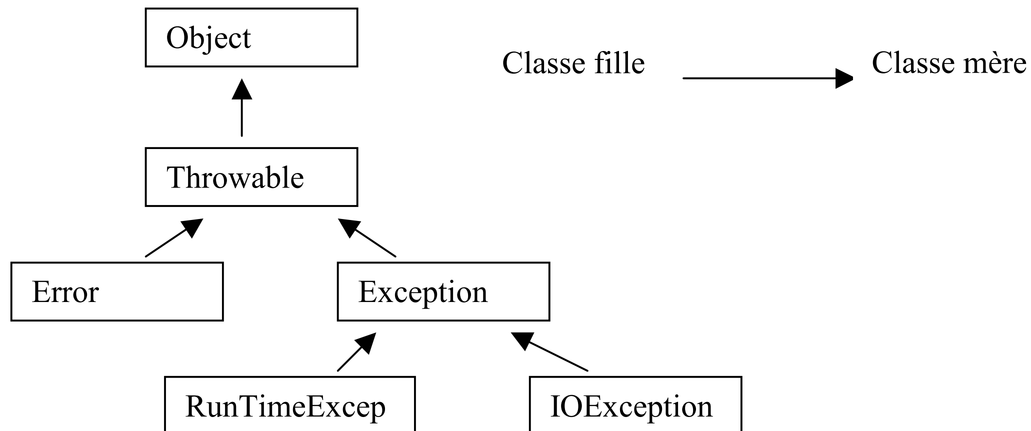
Si la clause « Throws » est omise le compilateur signalera pour chaque type d'erreur susceptible de se produire, le message d'erreur suivant : "NomClasseException must be caught, or it must be declared in the throws clause of this method" car il sait pour chaque méthode appelée, les exceptions qu'elle peut lever.

Dans le cas où la méthode appelante veut traiter l'erreur et non la propager, il faut utiliser les instructions « try … catch ». La clause « try » englobe le traitement susceptible de lever des exceptions, la (ou les) clause(s) « catch » mentionne(nt) le traitement qui sera exécutée pour chaque type d'erreur commise. Le programme suivant illustre ce principe sur la méthode Lire pour laquelle on ne veut plus propager les erreurs levées mais les traiter en retournant la valeur « null » ou un message approprié. La clause « throws » devient inutile si on mentionne autant de clauses « catch » que de type d'erreurs.

```
public static void LireTaux()
{
try
    {FE = new ObjectInputStream (new FileInputStream ("Param.DTA"));
    TauxTVA = ((Float) FE.readObject()).floatValue();
    FE.close();
    }
catch (IOException Ex)
    {System.out.println (« Erreur de lecture sur Param.DTA »);  }
catch (ClassNotFoundException e)
    {System.out.println (« Le fichier Param.DTA ne contient pas les objets attendus ») ;  }
```

Dans le cas où le traitement de l'erreur est le même quelque soit son type, on écrira une seule instruction catch avec un nom de classe d'exception générique. Cette notion est basée sur la hiérarchie des classes d'exceptions dont un extrait (issu de l'ouvrage « Java la synthèse) est présenté ci-après. Pour le programme précédent on pourrait, par exemple, proposer la solution qui suit :

```
public static void LireTaux()
{
try
     {FE = new ObjectInputStream (new FileInputStream ("Param.DTA"));
     TauxTVA = ((Float) FE.readObject()).floatValue();
     FE.close();
     }
catch (Exception Ex)
     {System.out.println (« Erreur sur le fichier Param.DTA»);
     }
}
```



Application:

Du fait de la présence de la clause « throws » dans les en-têtes des méthodes « LireTaux » et « EnregistrerTaux » la classe « ProdCatalogue » présentée ci-dessous n'entraîne plus d'erreurs à la compilation. En effet, lors de l'exécution de la méthode « LireTaux », si le fichier « Param.DTA » n'existe pas, une erreur de type « IOException » est levée et propagée aux méthodes appelantes ce qui entraîne l'arrêt de l'exécution. On souhaite maintenant modifier cette méthode afin de ne plus stopper l'exécution du programme mais de demander à l'utilisateur de communiquer un taux de TVA dans le cas où le fichier « Param.DTA » n'existe pas dans l'arborescence du support. Proposer une solution.

```
import java.io.*;
class ProdCatalogue
{public static float TauxTVA;
private static ObjectInputStream FE;
private static ObjectOutputStream FS;
private int Reference;
private String Designation;
private float PrixHT;

public static void LireTaux() throws IOException, ClassNotFoundException
{
FE = new ObjectInputStream (new FileInputStream ("Param.DTA"));
TauxTVA = ((Float) FE.readObject()).floatValue();
FE.close();
}

public static void EnregistrerTaux() throws IOException
{
FS = new ObjectOutputStream (new FileOutputStream ("Param.DTA"));
FS.writeObject (new Float(TauxTVA));
FS.close();
}

public static void ModifierTaux (float NouveauTaux)
{ TauxTVA = NouveauTaux;  }

public ProdCatalogue()
{System.out.println ("Référence ?");
Reference = Lire.i();
System.out.println ("Désignation ?");
Designation = Lire.S();
System.out.println ("Prix HT ?");
PrixHT = Lire.f();
}
public void AfficherPrixTTC()
{ System.out.println (PrixHT*TauxTVA);  }
}
```

Auto-évaluation n°8 : les collections

Dans le cadre du développement d'un logiciel relatif au suivi du trafic d'un réseau INTRANET, on s'intéresse aux classes « Connexion» et « ListeConnexions» définies partiellement ci-dessous

```
class Connexion
{ private int Duree;
  private float VolumeEntrant;
  private float VolumeSortant;

  public Connexion (int D, float VE, float VS)          // Constructeur de la classe
  { … }

  // Accesseurs en consultation qui renvoient la durée, le flux entrant et le flux sortant associés à la connexion
  public int SaDuree()
  {...}
  public float SonvolumeEntrant()
  {...}
  public float SonvolumeSortant()
  {…}
}
```

Remarques:

– On suppose que la durée d'une connexion est exprimée en minutes entières et que les volumes sont exprimés en KO.

– Aucun contrôle n'est à prévoir pour le constructeur de la classe.

1. Ecrire l'ensemble des methodes

```
class Connexion
{ private int Duree;
  private float VolumeEntrant;
  private float VolumeSortant;

  public Connexion(int D, float VE, float VS)    // Constructeur de la classe
  { Duree = D;
    VolumeEntrant = VE;
    VolumeSortant = VS;
  }

// Accesseurs en consultation qui renvoient la durée, le flux entrant et le flux sortant associés à la connexion
  public int SaDuree()
  { return (Duree);  }

  public float SonvolumeEntrant()
  { return (VolumeEntrant);  }

public float SonvolumeSortant()
{ return (VolumeSortant);  }
```

```
import java.util.*;
class ListeConnexions
{ private Vector Vect;
  public ListeConnexions()              // Constructeur de la classe ListeConnexions
    {Vect = new Vector();  }

// Méthode qui ajoute en dernière position de la liste une nouvelle connexion fournie en paramètre dans la liste
  public void AjouterNouvelleConnexion (Connexion C)
    { Vect.add(C);  }

  private int NbTotal()                 // Méthode qui affiche le nombre total de connexions
    { return Vect.size();  }

// Méthode qui affiche la durée total des connexions mémorisées dans la liste
  private int DureeTotale()
  { int i, tot=0;
    for (i=0; i <= NbTotal()- 1; i ++)
        tot+=((Connexion) Vect.elementAt(i)).SaDuree();
    return tot;
  }

  public float DureeMoyenneO        // Méthode qui renvoie la durée moyenne de connexion
  { return DureeTotale() / NbTotal();  }

  public void AfficherListe()           // Méthode qui affiche toutes les connexions de la liste
  { int i;
    for (i=0; i <= NbTotal()-1; i ++)
    { System.out.print (((Connexion) Vect.elementAt(i)).SaDuree()+"mn");
  System.out.print (((Connexion) Vect.elementAt(i)).SonVolumeEntrant() + "K");
  System.out.print (((Connexion) Vect.elementAt(i)).SonVolumeSortant() + "K");
    }
  }

// Méthode qui extrait de la liste les connexions dont la durée est supérieure à 5 minutes
// et dont les flux sortant et entrant sont inférieurs à 50 K. L'extraction n'altère pas la liste initiale.
  public ListeConnexions Extract()
  { ListeConnexions LTmp;
    LTmp = new ListeConnexions();
    int i, tot=0;
    for (i=0; i <= NbTotal()- 1; i ++)
        if (((Connexion) Vect.elementAt(i)).SaDuree()> 5 &&
          ((Connexion) Vect.elementAt(i)).SonVolumeEntrant() < 50 &&
          ((Connexion) Vect.elementAt(i)).SonVolumeSortant() < 50)
            LTmp.AjouterNouvelleConnexion((Connexion) Vect.elementAt(i));
    return (LTmp);
  }
}
```

2. Ecrire le programme de test qui réalise les opérations suivantes:

– création des quatre connexions suivantes dans la liste « L»

Durée	Volume entrant	Volume Sortant
1	14.4	7
8	10	5
12	535	5678
7	2	2

– Affichage de la liste « L»

– Affichage du temps moyen de connexion

– Création de la liste des connexions « LS » dont la durée est supérieure à 5 minutes et dont les flux sortant et entrant sont inférieurs à 50 K

– Affichage de la liste « LS»

```
class TestListeConnexions
{ public static void main (String args[])
  { ListeConnexions L,LS;
    L = new ListeConnexions();
    L.AjouterNouvelleConnexion(new Connexion (1, 14.4f, 7f));
    L.AjouterNouvelleConnexion(new Connexion (8, 10f, 5f));
    L.AjouterNouvelleConnexion(new Connexion (12, 535f, 5768f));
    L.AjouterNouvelleConnexion(new Connexion (7, 2f, 2f));
    L.AfficherListe();
    System.out.println (L.DureeMoyenne());
    LS=L.Extract();
    LS.AfficherListe();
  }
}
```

PROGRAMMATION ORIENTEE OBJETS :

L'HERITAGE en JAVA

L'HERITAGE EN JAVA

I. INTRODUCTION 54
II. DEFINITION DE L'HERITAGE 56
2.1) Le constructeur d'une classe dérivée 57
2.2) Les propriétés d'une classe dérivée 58
2.3) Les méthodes d'une classe dérivée 59
III. MANIPULATIONS D'OBJETS ISSUS DE CLASSE MERE ET FILLE 60

INFORMATIQUE – CNAM ANGOULEME 2000-2001

L'HERITAGE EN JAVA

I. INTRODUCTION

Compte tenu des connaissances acquises jusqu'à présent, pour implémenter une relation «est un» entre deux entités «A» et « B », la seule solution envisageable consiste à intégrer l'entité « A» dans l'entité « B». Celle technique appelée composition est illustrée dans le tableau ci-dessous au travers des classes «Produit» et «ProduitAchete». *Un produit est caractérisé par trois propriétés: sa référence, sa désignation, et son prix de vente HT. Un produit acheté est un produit qui a, en plus, les deux caractéristiques suivantes:le nom de son fournisseur, et son prix d'achat HT.*

```
class Produit
{ private String Reference;
  private String Designation;
  private float PrixventeHT;

public Produit(String R, String D, float PV)     // Constructeur
{ Reference = R;
  Designation = D;
  PrixVenteHT = PV;
}

// Méthode qui augmente le prix de vente d'un pourcentage passé en paramètre
public void AugmenterPrixVenteHT (float TauxM)
{ PrixVenteHT = PrixVenteHT *(l+TauxM);
}

// Méthode qui renvoie sous la forme d'une chaîne de caractères toutes les caractéristiques du produit
public String InfosQ
{ retum  (Reference + « » + Designation + « » + PrixVente HT);
}

public float SonPrixVenteHT()          // Accesseur en consultation qui renvoie le prix
{ return (PrixVenteHT);
}
```

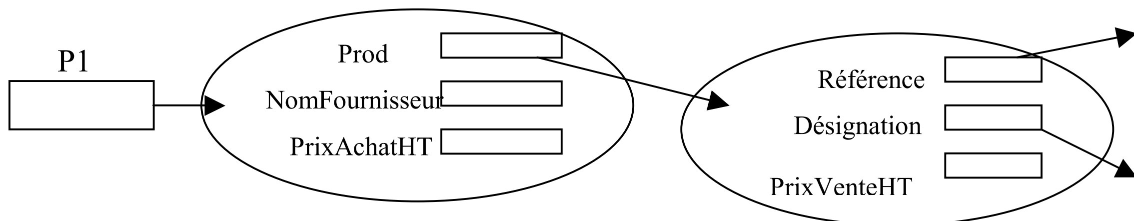
```
class ProduitAchete
{ private Produit Prod;
  private String NomFournisseur;
  private float PrixAchatHT;

// Constructeur
public ProduitAchete (String R, String D, float PV, String F, float PA)
{ Prod = new Produit(R,D,PV);
  NomFournisseur = F;
  PrixAchatHT = PA;
}

public float Marge()              // méthode qui calcule la marge sur un produit acheté
{ return (Prod.SonPrixVenteHT() - PrixAchatHT);
}

// Méthode qui renvoie sous la forme d'une chaîne de caractères toutes les caractéristiques du produit acheté
public String Infos()
{ return (Prod.Infos() + « » + NomFournisseur + « » + PrixAchatHT);
}
```

Cette solution qui consiste à dire qu'un produit acheté contient un produit et non « est un produit » peut être représentée ainsi: (On suppose que P1 est une variable pointant sur un objet de la classe « ProduitAchete »).



Avec une telle implémentation, pour augmenter de 5 % le prix de vente d'un produit acheté pointé par la variable Pi, on devra écrire 1'instruction:

P1.SonProduit().AugmenterPrixVenteHT(0.05f);
Accesseur à Prod

Cette écriture n 'est pas naturelle car compte tenu du frit qu 'un produit acheté est un produit, on devrait pouvoir écrire: Pi.AugmenterPrix VenteHT(O. OS». C'est le concept d'héritage qui va permettre de représenter le lien «est un» de manière plus pertinente.

II. Définition de l'héritage

Pour exprimer le fait que tout objet de la classe « B» est un objet de la classe «A» on devra écrire l'en-tête de la classe « B» de la manière suivante:

```
class B extends A
{          // données et méthodes spécifiques
}
```

La définition de cette classe peut être interprétée ainsi:

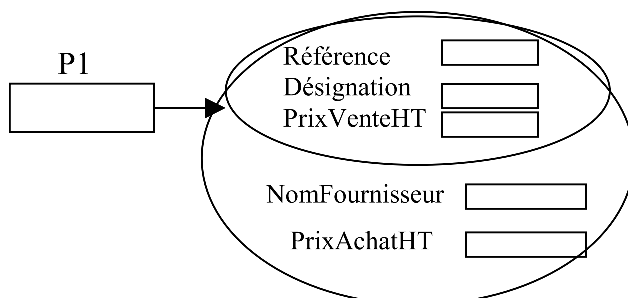
- la clause « extends» spécifie qu'un objet de la classe « B » hérite de toutes les fonctionnalités de la classe «A ». Cela signifie que l'on pourra, par exemple, appliquer sur un objet de la classe «B» les méthodes publiques de la classe « A».
- le contenu de la classe «B » décrit uniquement les fonctionnalités (propriétés et méthodes) spécifiques à la classe «B». L'ajout de fonctionnalités pour la classe «B» est appelée spécialisation. Sans celle spécialisation, la classe « B » n'aurait aucun intérêt car un objet de la classe « B» serait strictement identique à un objet de la classe « A».

D'un point de vue terminologie, on rappelle que « B » est une sous-classe de « A» ou encore une classe dérivée de « A». « A» est une super-classe ou encore une classe de base. L'héritage permet donc de réutiliser très facilement le code déjà écrit. Il garantit en outre que toute modification au niveau de la classe « A » sera automatiquement répercutée au niveau de la classe dérivée.

Le début de la définition de la classe « ProduitAchete » s'écrirait donc:

```
class ProduitAchete extends Produit       // ProduitAchete est la classe dérivée ou classe fille.
{ private String NomFournisseur;          // Produit est la classe de base ou classe mère.
  private float PrixAchatHT;
}                                         // ProduitAchete hérite de toutes les méthodes publiques de Produit.
```

Un objet de cette classe pourrait être schématisée ainsi:



Application: On considère la classe « Compte » étudiée à l'auto-évaluation n^0 6 et destinée à gérer des comptes. Un extrait de cette classe est fourni ci-dessous

```
class Compte
{ private int Numero;
  private char Type;
  private float Solde;
  public Compte (int N, char T)
  { Numero = N;
    Type = T;
    Solde = 0;
  }

  public float SonSolde()  { return (Solde);  }

  public int SonNumero() { return (Numero);  }

  public void Crediter(float Montant) { Solde += Montant;  }

  public void AfficherInfos()
  { System.out.println (« Compte » + Numero + « » + Type + « » + Solde);  }
```

Proposer une solution pour implémenter la notion de compte d'épargne: un compte d'épargne est un compte bancaire qui a, en plus, un taux de rémunération

```
class CompteEpargne extends Compte
{ private float TauxRenumeration;
…
}
```

2.1) Le constructeur d'une classe dérivée

Pour créer un objet de la classe dérivée, on doit obligatoirement créer un objet de la classe de base par un appel au constructeur de celle-ci. Cette obligation garantit la relation « est un » entre l'objet de la classe dérivée et l'objet de la classe de base. *Ainsi, le constructeur de la classe « ProduitAchete » doit avoir la forme suivante:*

```
public ProduitAchete (String R, String D, float P frs String F, float PA)
{ super(R,D,P V);               // appel au constructeur de la classe de base
  NomFournisseur = F;
  PrixAchatHT = PA;
}
```

Remarques:
- Le mot-clé « super » désigne l'objet de la classe supérieure. Ainsi, l'instruction "super(R,D,PV)" crée un produit qui deviendra un produit acheté par la valorisation (au moyen des deux dernières instructions) des variables d'instances : NomFournisseur et PrixAchatHT.
- L'appel au constructeur de la classe mère doit être la première instruction du constructeur de la méthode dérivée.
- Si l'appel au constructeur de la classe mère n'est pas explicitement cité dans le constructeur de la méthode dérivée, le compilateur recherche un constructeur par défaut (constructeur sans paramètre) dans la classe mère. Si celui-ci n'existe pas, le compilateur signale le message d'erreur suivant:
  no constructor matching Produit() found in Produit

Application: Ecrire le constructeur de la classe « CompteEpargne »:
dont la signature est fournie ci-dessous:

public CompteEpargne (int N, int TauxR)

Les paramètres N et TauxR correspondent respectivement au numéro de compte et au taux de rémunération du compte épargne. Le type du compte doit être obligatoirement « E » pour signifier qu'il s'agit d'un compte épargne. Le solde doit être initialisé à 0.

```
public CompteEpargne (int N, intTauxR)
{ super (N, 'E');
  TauxRenumeration = TauxR;
}
```

2.2) Les propriétés d'une classe dérivée

Comme il a déjà été dit dans les paragraphes précédents, un objet d'une classe dérivée hérite des propriétés de la classe mère et admet en plus des propriétés spécifiques. Il est important de souligner que l'héritage n'est pas synonyme d'accessibilité. Cela signifie que l'accès (depuis une classe dérivée) à une variable d'instance de la classe de base n'est pas immédiat: il dépend en fait de son spécificateur d'accès. Les trois possibilités sont récapitulées dans le tableau suivant:

Spécificateur d'accès d'une variable d'instance de la classe de base	Conséquences pour la référence dans la classe dérivée
public	Accès possible dans la classe dérivée comme dans toutes les classes
private	Accès impossible
protected	Accès possible dans toutes les classes dérivées de la classe mère

Dans le cas où la propriété de la classe mère est inaccessible depuis une classe dérivée, on doit recourir à des méthodes publiques de la classe mère pour la modifier ou la consulter.

Application:
Compte tenu du spécificateur d'accès défini pour les variables de la classe « Compte », indiquer si elles sont accessibles depuis la classe « CompteEpargne ».

Toutes les variables de la classe Compte sont définies en private, et donc inaccessibles depuis toute autre classe.

2.3) Les méthodes d'une classe dérivée

Avant de parler des méthodes qui seront explicitement définies dans la classe dérivée, il faut se rappeler, que toute méthode de la classe de base peut être appliquée à un objet de la classe dérivée. Outre ces traitements communs, la définition d'une classe dérivée intègre des méthodes dues à sa spécialisation. Celles-ci peuvent être réparties en 3 catégories distinctes:

- La première catégorie regroupe toutes les méthodes qui n'existent pas dans la classe mère. Ces méthodes ne peuvent s'appliquer que sur des objets de la classe dérivée.

- La seconde catégorie englobe les méthodes dites surchargées. Ces traitements ont le même nom que des méthodes de la classe mère mais avec des signatures différentes. Pour que l'exécution réussisse il faudra qu'il y ait cohérence entre la signature de la méthode et l'objet sur lequel la méthode est appliquée.

- La troisième et dernière catégorie regroupe les méthodes redéfinies. Ces méthodes qui ont la même signature que dans la classe mère sont dites polymorphes. C'est au niveau de l'exécution et en fonction de l'objet sur lequel la méthode est appliquée que la forme de la méthode est choisie. Il faut noter que la redéfinition supprime l'héritage. Cela signifie que, pour une méthode polymorphe, on ne pourra plus appliquer la forme définie au niveau de la classe mère sur 1 objet de classe dérivée.

Le modificateur "final" dans une méthode de la classe de base empêche la redéfinition de cette méthode dans les classes filles:

public final void Méthode { … }

La définition de la classe « ProduitAchete » fournie ci-dessous présente deux méthodes: Marge et Infos appartenant respectivement à la première et à la dernière catégorie.

```
class ProduitAcheté extends Produit

// Méthode qui renvoie la marge réalisée; cette méthode n'existe pas dans la classe mère
{ public float Marge()
  { return (super.PrixVenteHT() - PrixAchatHT);
  }

// Méthode polymorphe qui renvoie sous forme de chaîne de caractères les informations du produit acheté:
  public String Infos()
  { return (super.Infos() + « » + NomFournisseur + « » + PrixAchatHT);
  }
}
```

<u>Remarque</u>: la méthode Infos est construite à partir du code de la méthode « Infos » définie au niveau de la classe mère par la référence : super.Infos().Si l'on omet dans cet appel le mot-clé super, la méthode Infos devient récursive et provoque une exécution sans fin.

Application: Ecrire dans la classe « CompteEpargne » les 2 méthodes suivantes:
- AfficherInfos() afin de visualiser le message:
  compte épargne : numéro, solde, taux de rémunération
- AjouterInterets() qui ajoute au solde les intérêts annuels

```
class CompteEpargne extends Compte
{
// Méthode qui visualise les caractéristiques du compte épargne
  public void AfficherInfos()
  { System.out.println ("Compte épargne:" + super.SonNumero() +" "+
                                 super.SonSolde() +" "+ TauxRemuneration);
  }

// Méthode qui ajoute les intérêts annuels au solde
  public void AjouterInterets()
  { super.Crediter (TauxRemuneration*super.SonSolde());  }
}                                   // *this.SonSolde() ou * SonSolde()
```

III. Manipulations d'objets issus de classe mère et fille

Sur un objet de la classe mère, on peut appliquer toutes les méthodes publiques de cette classe. Sur un objet de la classe dérivée, on peut appliquer:
- Toutes les méthodes publiques de la classe mère non redéfinies,
- Toutes les méthodes publiques de la classe dérivée.

L'affectation d'un objet de la classe dérivée dans une variable objet de la classe mère est possible sans avoir à spécifier un transtypage. *Ainsi tout produit acheté étant un produit on peut affecter dans une variable de la classe "Produit" un objet de la classe "ProduitAcheté".*

```
Produit P;
ProduitAchete PA;
PA = new ProduitAchete( "R45", "Table", 250, "Dupont", 150);
P=PA;
```

Par contre l'affectation réciproque (objet de la classe mère dans une variable objet de la classe fille) nécessite un <u>transtypage</u> explicite. *Ainsi un produit n 'étant pas forcément un produit acheté, on devra spécifier un <u>cast</u> afin d'indiquer au compilateur que l'objet pointé par une variable de la classe « Produit » est effectivement un produit acheté.* Cette technique est nécessaire dès lors que l'on veut appliquer des méthodes de la classe dérivée à un objet pointé par une variable de la classe mère. *Par exemple, pour appliquer la méthode "Marge()" à un produit acheté pointé par la variable P (de la classe « Produit ») on devra écrire: ((ProduitAchete) P).Marge().*

Application: Ecrire le programme de test qui réalise les opérations suivantes:

– Création des comptes bancaires suivants:

N° de compte	Type de compte	Taux de rémunération	Variable objet
1	E	0.035	CE1
2	C		C1
3	E	0.04	CE2

– Affectation dans les variables C2 C3 des comptes d'épargnes pointés par CE1 et CE2.
– Ajout de 100 F sur les comptes pointés par Cl et C2.
– Mise à jour des intérêts du compte pointé par C2.
– Affichage des informations des comptes pointés par Cl, C2, C3.

```
class TestCompte
{ public static void main (String args[])
  { Compte C1,C2,C3;
    CompteEpargne CEI ,CE2;
    CEl = new CompteEpargne (1, 0.035f);      // sinon il le considère un double
    Cl = new Compte (2,'C');
    CE2 = new CompteEpargne (3,0.04f);        // sinon il le considère un double
    C2 = CEl ;
    C3 = CE2 ;
    Cl .Crediter(100);
    C2.Crediter(l00);
    ((CompteEpargne)C2).AjouterInterets();
    Cl.AfficherInfos();  |
    C2.Afficherlnfos();  |  // polymorphisme
    C3.Afficherlnfos();  |
  }
}
```

Auto-évaluation n°9: Révisions:

Partie n° 1: On considère la classe « Facture » définie partiellement ci-dessous:

```
class Facture
{ private static float TauxTVA=0.206f;
  private int NoFacture;
  private float MontantHT;
  …
  public int SonNumero()
  { return (NoFacture);
  }
}
```

1. Ecrire la méthode privée ValideMontant(float M) dont l'objectif est de contrôler si le montant M passé en paramètre est strictement positive. Sa logique algorithmique est présentée ci-dessous:

```
Si M <=0 Alors
    Afficher «le montant doit être strictement positif»
    Arrêt de l'exécution
FinSi
Retourner M
```

2. Un programme utilisant la classe « Facture» contient les deux instructions suivantes:

```
Facture F;
F = new Facture (1, 200);  // (N° de la facture, Montant HT de la facture)
```

Déduire de la dernière instruction la signature associé a l'un des constructeurs de la classe. Ecrire le contenu de ce constructeur en envisageant uniquement un contrôle sur le montant hors taxes qui doit être strictement positif.

3. Ecrire l'accesseur en modification « ModifMontant» chargé de modifier le montant HT d'une facture. Un exemple d'appel de cet accesseur est fourni ci-dessous et affecte la valeur 100 au montant HT de la facture pointée par F.

```
F.ModifMontant(100);
```

On devra aussi dans cet accesseur vérifier que le nouveau montant est strictement positif.

4. Ecrire la méthode «SonMontantTTCO» chargée de renvoyer le montant TTC de la facture. Indiquer l'instruction à écrire dans le programme de test pour afficher le montant TTC de la facture pointée par la variable F.

5. Définir la méthode de classe «ChangerTaux » qui permet de changer le taux de TVA en vigueur contenu dans la variable de classe « TauxTVA». Le nouveau taux est un paramètre formel de la méthode. Indiquer l'instruction à spécifier dans le programme de test pour affecter la valeur 0.204 au taux de TVA.

Partie n° 2: Pour construire la classe «ListeFactures » destinée à gérer une liste des factures on envisage les deux implantations suivantes:

<table>
<tr><th>Solution 1</th><th>Solution 2</th></tr>
<tr><td><pre>
class ListeFactures
{ private Facture [] T;
  private int Nb;
  public ListeFactures O
  { T = new Facture [50];
    Nb=0;
  }
  public void AjouterUneFacture (Facture F)
  { T[Nb] = F;
    Nb ++;
  }
</pre></td><td><pre>
importjava.util.Vector
class ListeFactures
{ private Vector V;
    public ListeFactures()
    { V = new Vector();
    }
    public void AjouterUneFacture (Facture F)
    { V.add (F);
    }
</pre></td></tr>
</table>

6. Ecrire pour chacune des deux solutions la méthode «TotalTTC » qui renvoie le total TTC des factures contenues dans la liste. On suppose que la méthode «SonMontantTTC() » est disponible dans la classe «Facture ».

Partie n^0 3: Les clients de la société émettrice de factures peuvent être répartis en deux catégories:

- les particuliers qui représentent la majorité des clients de l'entreprise et pour lesquels la tarification des frais de livraison dépend seulement du département de livraison que l'on suppose être le département du client,
- les professionnels pour lesquels la tarification de la livraison dépend du département mais aussi du chiffre d'affaires réalisé.

Pour mémoriser ces deux types de clients, on propose les classes "Client" et "ClientPro" définies ci-dessous:

- **La classe « Client »**

```
class Client
{ private String Nom;
  private String Prenom;
  protected int Departement;
  public Client (String N, String P, int D)
  { Nom=N;
    Prenom = P;
    Departement = D;
  }
  public float FraisDeLivraison()
  { float F=0;
    if (Departement != 16)
        F = 1500;
    return (F);
  }
}
```

- **La classe « ClientPro »**

```
class ClientPro extends Client
{ private float CA;
  public ClientPro (String N, String P, int D)
  { ...
  }
  public void CumulerAchat(float Montant)
  { CA+=Montant;
  }
}
```

7. Quelle est la conséquence du spécificateur protected associé à la propriété "Departement" ?

8. Définir le constructeur de la classe « ClientPro ». A l'issue de l'exécution de ce constructeur, la valeur O doit être affectée dans la variable CA.

9. Redéfinir la méthode « FraisDeLivraison » dans la classe « Clientl>ro » afm de respecter la règle de gestion suivante:

«La tarification des frais de livraison suit les mêmes règles que pour les particuliers exception faite du cas où le chiffre d'affaires du professionnel est supérieur à 10 000 F et les frais de livraison payants. Dans ce cas une réduction de 5 % est appliquée. »

10. On considère le programme de test suivant:

```
class TestClient
{ public static void main (String args[])
  { Client C1,C2,CInt;
    ClientPro CP;
    Cl = new Client("Dupont","Paul", 16);
    C2 = new Client("Durand","René", 17);
    CP = new ClientPro("Dubois", "Claude", 17);
    CP.CumulerAchat(l5 000);
    CInt=CP;
    System.out.println(CInt.FraisDeLivraison());
    System.out.prihtln (C2.FraisDeLivraison());
  }
}
```

Quel sera le résultat affiché par les deux dernières instructions?

1425 et **1500**

PROGRAMMATION ORIENTEE OBJETS :

ANNEXES: Quelques classes

EXEMPLES DE CLASSES JAVA

I. PUISSANCEDE10 67
II. CAPACITE 68
III. PUISSANCE 69
IV. INVENTAIRE 69
V. CONNEXION & LISTECONNEXION 70
VI. PRODUIT & PRODUITACHETE 72
VII. COMPTE & COMPTEEPARGNE 73
VIII. FACTURE & LISTEFACTURES 75
IX. CLIENT & CLIENTPRO 77

INFORMATIQUE – CNAM ANGOULEME 2000-2001

EXEMPLES DE CLASSES JAVA

I. PuissanceDe10

- **class PuissanceDe10**

```
{ private int Exposant;
  public PuissanceDe10 (int E)            // on ne met pas de type pour le constructeur.
  {  Exposant = E ;  }

  public void Afficher ()
  {  System.out.println ("10^" + Exposant) ;  }

  public void Multiplier (PuissanceDe10 P)
  {  Exposant += P.Exposant ;  }

  public PuissanceDe10 MultiplierBis (PuissanceDe10 P)
  {  PuissanceDe10 Result;
     Result = New PuissanceDe10 (P.Exposant + Exposant);
     Return (Result) ;  }

  public PuissanceDe10 MultiplierTer (PuissanceDe10 P)
  {  Exposant += P.Exposant ;
     return this ;                        // this retourne l'objet lui-même.
  }

  public void MultiplierQua (PuissanceDe10 P)
  {  P.Exposant += Exposant ;  }          // ici c'est P2 qui va être modifié.

  public float Valeur ()
  {  int P = 1, i ;
     for (i = 1; i <= Math.abs(Exposant); i++;) P = P*10 ;
     if (Exposant < 0) P = 1 / P ;
     return (P) ;
  }
}
```

II. Capacite

➢ **class Capacite**

```
{ private float Valeur;
  private char Unite;
  public Capacite (float V, char U)
  { if (U != 'O' && U != 'K' && U !=' M')
    { System.out.println (" capacité incorrecte");
      System.exit(1);
    }
    else
    { Valeur = V; Unite = U;  }
  }

  private long ValeurEnOctets ()
  { long L = (long) Valeur;
    if (Unite = = 'K')   L = (long) Valeur* 1024;
    else  if(Unite = = 'M')  L = (long) Valeur * 1024*1024;
    return L;
  }

  public void Ajouter (Capacite C)
  { Valeur = this.ValeurEnOctets () + C.ValeurEnOctets ();
    Unite = 'O';
  }

  public void Afficher ()
  {  System.out.println (Valeur +" "+ Unite);  }
}
```

➢ **class TestCapacite**

```
{ public static void main (String args[])
  {  Capacite C1,C2;
     C1= new Capacite (10,'O');
     C2 = new Capacite (1,'K');
     Cl.Ajouter (C2);
     Cl.Afficher ();
     C2.Afficher();
}
```

III. Puissance

➢ **class Puissance**

```
{ private static int base = 2 ;
  private int exposant ;
  public static void ChoisirBase()
  { base = Lire.i() ;  }
     exposant = Lire.i() ;  }
  public void Afficher ()
  { System.out.println (base + « ^ » + exposant) ;  }
}
```

IV. Inventaire

➢ **class Inventaire**

```
{ private Produit [] ListeProduits;
  private int NbProduits = 0;

  public Inventaire()
  { ListeProduits = new Produit [100];  }

  public void AjouterunProduit()
  { ListeProduits[NbProduits] = new Produit;
    NbProduits ++;
  }
  public Produit RechercherUnProduit (int RefProduit)
  { int i=0;
    Produit tmp=null;
    boolean Trouve = false;
    while (! Trouve && i <=NbProduits-1)
    if (RefProduit = = ListeProduits[i].SaReference())
        { Trouve = true;
          tmp = ListeProduits[i];
        }
    else  i ++;
    return tmp;
  }

  public void AfficherProduitsACommander()
  { int i;
    for (i=0; i<= NbProduits - i; i ++)
        if (ListeProduits[i].Acommander())
          System.out.print (ListeProduits[i].SaReference());
  }
}
```

- **class TestInventaire**

```
{ public static void main (String args[])
  { Inventaire INV; int R; char Rep;
    INV= new Inventaire();
    do
    { INV.AjouterunProduit();
      System.out.println ("Autre produit (O/N)?");
      Rep = Lire.c();
    } while (Rep == 'O');
    System.out.println ("Saisir une référence de produit");
    R = Lire.i();
    if (INV.RechercherunProduit(R) != null)
        System.out.println (INV.RechercherunProduit(R).SonStock());
    else
        System.out.println ("Pas de produit pour cette référence");
    System.out.println ("Liste des produits à commander");
    INV.AfficherProduitsACommander();
  }
}
```

V. Connexion & ListeConnexion

- **class Connexion**

```
{ private int Duree;
  private float VolumeEntrant;
  private float VolumeSortant;

  public Connexion(int D, float VE, float VS)    // Constructeur de la classe
  { Duree = D;
    VolumeEntrant = VE;
    VolumeSortant = VS;
  }

// Accesseurs en consultation qui renvoient la durée, le flux entrant et le flux sortant associés à la connexion
  public int SaDuree()
  { return (Duree);  }

  public float SonvolumeEntrant()
  { return (VolumeEntrant);  }

public float SonvolumeSortant()
{ return (VolumeSortant);  }
```

```
import java.util.*;
➢ class ListeConnexions
{ private Vector Vect;
  public ListeConnexions()              // Constructeur de la classe ListeConnexions
    {Vect = new Vector();  }

// Méthode qui ajoute en dernière position de la liste une nouvelle connexion fournie en paramètre dans la liste
  public void AjouterNouvelleConnexion (Connexion C)
    { Vect.add(C);  }

  private int NbTotal()             // Méthode qui affiche le nombre total de connexions
    { return Vect.size();  }

// Méthode qui affiche la durée total des connexions mémorisées dans la liste
  private int DureeTotale()
  { int i, tot=0;
    for (i=0; i <= NbTotal()- 1; i ++)
        tot+=((Connexion) Vect.elementAt(i)).SaDuree();
    return tot;
  }

  public float DureeMoyenneO       // Méthode qui renvoie la durée moyenne de connexion
  { return DureeTotale() / NbTotal();  }

  public void AfficherListe()       // Méthode qui affiche toutes les connexions de la liste
  { int i;
    for (i=0; i <= NbTotal()-1; i ++)
    { System.out.print (((Connexion) Vect.elementAt(i)).SaDuree()+"mn");
  System.out.print (((Connexion) Vect.elementAt(i)).SonVolumeEntrant() + "K");
  System.out.print (((Connexion) Vect.elementAt(i)).SonVolumeSortant() + "K");
    }
  }

// Méthode qui extrait de la liste les connexions dont la durée est supérieure à 5 minutes
// et dont les flux sortant et entrant sont inférieurs à 50 K. L'extraction n'altère pas la liste initiale.
  public ListeConnexions Extract()
  { ListeConnexions LTmp;
    LTmp = new ListeConnexions();
    int i, tot=0;
    for (i=0; i <= NbTotal()- 1; i ++)
        if (((Connexion) Vect.elementAt(i)).SaDuree()> 5 &&
          ((Connexion) Vect.elementAt(i)).SonVolumeEntrant() < 50 &&
          ((Connexion) Vect.elementAt(i)).SonVolumeSortant() < 50)
            LTmp.AjouterNouvelleConnexion((Connexion) Vect.elementAt(i));
    return (LTmp);
  }
}
```

- **class TestListeConnexions**

```
{ public static void main (String args[])
  { ListeConnexions L,LS;
    L = new ListeConnexions();
    L.AjouterNouvelleConnexion(new Connexion (1, 14.4f, 7f));
    L.AjouterNouvelleConnexion(new Connexion (8, 10f, 5f));
    L.AjouterNouvelleConnexion(new Connexion (12, 535f, 5768f));
    L.AjouterNouvelleConnexion(new Connexion (7, 2f, 2f));
    L.AfficherListe();
    System.out.println (L.DureeMoyenne());
    LS=L.Extract();
    LS.AfficherListe();
  }
}
```

VI. Produit & ProduitAcheté

- **class Produit**

```
{ private String Reference;
  private String Designation;
  private float PrixventeHT;

public Produit(String R, String D, float PV)     // Constructeur
{ Reference = R;
  Designation = D;
  PrixVenteHT = PV;
}

// Méthode qui augmente le prix de vente d'un pourcentage passé en paramètre
public void AugmenterPrixVenteHT (float TauxM)
{ PrixVenteHT = PrixVenteHT *(l+TauxM);
}

// Méthode qui renvoie sous la forme d'une chaîne de caractères toutes les caractéristiques du produit
public String InfosQ
{ retum  (Reference + « » + Designation + « » + PrixVente HT);
}

public float SonPrixVenteHT()          // Accesseur en consultation qui renvoie le prix
{ return (PrixVenteHT);
}
```

➢ **class ProduitAchete**
```
{ private Produit Prod;
  private String NomFournisseur;
  private float PrixAchatHT;

// Constructeur
public ProduitAchete (String R, String D, float PV, String F, float PA)
{ Prod = new Produit(R,D,PV);
  NomFournisseur = F;
  PrixAchatHT = PA;
}

public float Marge()                 // méthode qui calcule la marge sur un produit acheté
{ return (Prod.SonPrixVenteHT() - PrixAchatHT);
}

// Méthode qui renvoie sous la forme d'une chaîne de caractères toutes les caractéristiques du produit acheté
public String Infos()
{ return (Prod.Infos() + « » + NomFournisseur + « » + PrixAchatHT);
}
```

VII. Compte & CompteEpargne

➢ **class Compte**
```
{ private int Numero;
  private char Type;
  private float Solde;

  public Compte (int N, char T)
  { Numero = N;
    Type = T;
    Solde = 0;
  }

  public float SonSolde()
  { return (Solde);  }

  public int SonNumero()
  { return (Numero);  }

  public void Crediter(float Montant)
  { Solde += Montant;  }

  public void AfficherInfos()
  { System.out.println (« Compte » + Numero + « » + Type + « » + Solde);  }

}
```

➤ **class CompteEpargne** extends Compte

```
{ private float TauxRemuneration;

  public CompteEpargne (int N, float TauxR)
  { super (N, 'E');
    TauxRemuneration = TauxR;
  }

// Méthode qui visualise les caractéristiques du compte épargne
  public void Afficherlnfos()
  { System.out.println ("Compte épargne:" + super.SonNumero() +" "+
                                        super.SonSolde() +" "+ TauxRemuneration);
  }

// Méthode qui ajoute les intérêts annuels au solde
  public void Ajouterlnterets()
  { super.Crediter (TauxRemuneration*super.SonSolde()); }
}                                       // *this.SonSolde() ou * SonSolde()
```

➤ **class TestCompte**

```
{ public static void main (String args[])
  { Compte C1,C2,C3;
    CompteEpargne CEI ,CE2;
    CEl = new CompteEpargne (1, 0.035f);      // sinon il le considère un double
    Cl = new Compte (2,'C');
    CE2 = new CompteEpargne (3,0.04f);        // sinon il le considère un double
    C2 = CEl ;
    C3 = CE2 ;
    Cl .Crediter(100);
    C2.Crediter(l00);
    ((CompteEpargne)C2).AjouterInterets();
    Cl.AfficherInfos();   |
    C2.Afficherlnfos();   |  // polymorphisme
    C3.Afficherlnfos();   |
  }
}
```

VIII. Facture & ListeFactures

- **class Facture**

```
{ private static float TauxTVA=0.206f;
  private int NoFacture;
  private float MontantHT;

  private float ValideMontant(float M)
  { if (M<=0)
        { System.out.println("le montant doit être strictement positif');
          System.exit(1);
        }
  return (M);
  }

  public Facture(int NF, float M)
  { NoFacture = NF;
    MontantHT = ValideMontant (M);
  }

  public void ModifMontant(float NM)
  { MontantHT = ValideMontant(NM);
  }

  public float SonMontantTTC()
  { return (MontantHT*(1+TauxTVA));
  }

  public static void ChangerTaux(float NT)
  { TauxTVA=NT;
  }
}
```

- **class TestFacture**

```
{ public static void main (String args[])
  { Facture F;
    F = new Facture(56, 200);
    System.out.println (F.SonMontantTTC());
    F.ModifMontant(100);
    System.out.println (F.SonMontantTTC());
    Facture.ChangerTaux(0.204f);
    System.out.println (F.SonMontantTTC());
  }
}
```

- **class ListeFactures** **// implantée au moyen d'un tableau**

```
{ private Facture [] T;
  private int Nb;

  public ListeFactures()
  { T = new Facture[50]; Nb=0;  }

  public void AjouteruneFacture(Facture F)
  { T[Nb] = F; Nb++;  }

  public float TotalTTC()
  { int i;
    float Tot=0;
    for (i=0; i<=Nb-1; i ++)
            Tot+=T[i].SonMontantTTC();
    return Tot;
  }
}
```

- **La classe ListeFactures** **// implantée au moyen d'un vecteur**

```
import java.util.Vector;
class ListeFactures
{ private Vector V;
  public ListeFactures()
  { V = new Vector();  }

  public void AjouterUneFacture(Facture F)
  { V.add(F);  }

  public float TotalTTC()
  { int i;
    float Tot=0;
    for (i = 0; i <= V.sizeO-1; i ++)
            Tot += ((Facture)V.elementAt(i)).SonMontantTTC();
    return (Tot);
  }
}
```

- **class TestListeFactures**

```
{ public static void main (String args[])
  { Facture F;
    ListeFactures2 L;
    L = new ListeFactures2();
    F = new Facture(1,100);
    L.AjouterUneFacture (F);
    F = new Facture(2,50);
    L.AjouterUneFacture(F);
    System.out.println (L.TotalTTC());
  }
}
```

IX. Client & ClientPro

```
class ClientPro extends Client
{ private float CA;
  public ClientPro (String N, String P, int D)
  { super(N,P,D);
    CA=0;  }

  public void CumulerAchat(float Montant)
  { CA+=Montant;  }

  public float FraisDeLivraison()
  { float F=0;
    F = super.FraisDeLivraison();
    if (CA > 10 000 && F > 0)
            F *= 0.95f;
    return (F);
  }
}
```

- **class TestClient**

```
{ public static void main (String args[])
  { Client C1,C2,CInt;
    ClientPro CP;
    Cl = new Client("Dupont","Paul", 16);
    C2 = new Client("Durand","René", 17);
    CP = new ClientPro("Dubois", "Claude", 17);
    CP.CumulerAchat(l5 000);
    CInt=CP;
    System.out.println(CInt.FraisDeLivraison());
    System.out.prihtln (C2.FraisDeLivraison());
  }
}
```