Exercices en langage JAVA

H. Frezza-Buet et M. Ianotto

10 février 2003

Table des matières

Chapitre 1

Ecrire un programme, le compiler et l'exécuter

1.1 Calcul de la somme des 100 premiers entiers avec une classe définie dans un fichier source

On donne un exemple de programme qui calcule et affiche la somme des 100 premiers entiers. On définit dans la classe `Somme` la méthode `main` qui effectue tout le traitement.

```
// la classe Somme calcule et affiche la somme des 100 premiers entiers
public class Somme {
    public static void main(String[] args) {
    int Somme = 0;

    // calcule la somme des 100 premiers entiers
    for (int i = 1; i <= 100; i++)
        Somme = Somme + i;
    // affiche le résultat à l'écran
        System.out.println(Somme);
    }
}
```

Sauvegardez ce programme dans le fichier `Somme.java` puis compilez le à l'aide de la commande :

```
javac Somme.java
```

Exécuter la classe Somme à l'aide de la commande :

```
java -classpath . Somme
```

1.2 Calcul de la somme des 100 premiers entiers avec deux classes définies dans deux fichiers sources

Le but de ce deuxième exemple est faire intervenir la création d'un objet et d'appeler une méthode de cet objet. Pour cela, on définit la classe `Somme` dans le fichier `Somme.java` et la classe `TestSomme` dans le fichier `TestSomme.java`. Le fichier `Somme.java` contient les déclarations suivantes :

```
// La classe Somme calcule la somme des 100 premiers entiers
public class Somme {
    public int CalculSomme()
```

```
    {
    int Somme = 0;

    for (int i = 1; i <= 100; i++)
        Somme = Somme + i;
    return Somme;
    }
}
```

Le fichier `TestSomme.java` contient les déclarations suivantes :

```
// la classe TestSomme crée un objet Somme, appelle
// la méthode CalculSomme de cet objet et affiche
// le résultat à l'écran
public class TestSomme {
    public static void main(String[] args) {
    int Resultat;
    // creation d'un objet de type Somme
    Somme S = new Somme();

    // Calcule la somme des 100 premiers entiers
    Resultat = S.CalculSomme();
    // affiche le resultat à l'écran
    System.out.println(Resultat);
    }
}
```

Compilez les programmes à l'aide de la commande :

```
javac Somme.java TestSomme.java
```

Exécuter la classe TestSomme à l'aide de la commande :

```
java -classpath . TestSomme
```

Chapitre 2

Utiliser les types élémentaires, les expressions et les instructions de contrôle

2.1 Calcul de la factorielle d'un nombre

Question 1 Ecrire un programme qui calcule et affiche la factorielle de 5.

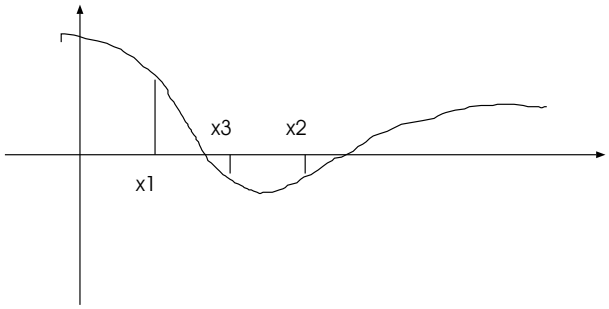
Question 2 On pourra ensuite écrire un programme qui calcule et affiche la factorielle d'un nombre lu au clavier. Pour cela, on reprendra tout ou partie du programme donné en annexe A.1. Il permet de lire un nombre entier au clavier et d'afficher à l'écran la valeur lue.
Indications : Pour tester ce programme, sauvegarder le dans le fichier `LectureEntier.java`, compilez le à l'aide de la commande `javac LectureEntier.java`, puis executez le avec la commande `java -classpath . LectureEntier`.

2.2 Calcul d'une racine d'une équation

On souhaite calculer la valeur approchée d'une racine d'une équation de la forme

$$f(x) = 0$$

On utilise pour cela une méthode de dichotomie. La méthode suppose que l'on parte de 2 valeurs x_1 et x_2 encadrant **une racine**, et telle que $f(x_1)$ et $f(x_2)$ soient de signe contraire. On calcule $x_3 = (x_1+x_2)/2$ et on remplace x_1 (respectivement x_2) par x_3 suivant que $f(x_3)$ est du même signe que $f(x_1)$ (respectivement $f(x_2)$). La largeur de l'intervalle considéré est donc divisée par 2 à chaque pas. Le calcul est répété jusqu'à ce que cette largeur soit inférieure à une valeur donnée ϵ.



Question 1 Ecrire un programme permettant d'obtenir la valeur approchée d'une racine de l'équation

$$x^2 + 3\pi(xcos(x) + sin(x)) = 0$$

pour $x_1 = 1$, $x_2 = 4$ et $\epsilon = 10^{-6}$. Les valeurs de de x_1, x_2 et ϵ seront codées dans le programme source.

Question 2 Modifier le programme afin de connaître le nombre d'itérations nécessaires pour trouver la solution.

2.3 Triangle de Pascal

Le triangle de Pascal est composé de M + 1 lignes consécutives donnant toutes les valeurs des C_n^p pour n variant de 0 à M et p variant de 0 à n. On rappelle que :

$$C_n^p = \frac{n!}{p!(n-p)!}$$

Question Ecrire un programme qui calcule et affiche les coefficients du triangle de Pascal pour une valeur M donnée. Pour M = 5, le résultat est le suivant :

```
1
1   1
1   2   1
1   3   3   1
1   4   6   4   1
1   5  10  10   5   1
```

Chapitre 3

Définir et utiliser les tableaux

3.1 Plus grande valeur des éléments d'un tableau

Ecrire un programme qui détermine et affiche la plus grande valeur des éléments d'un tableau. On complétera le programme suivant :

```
public class Valeurmax {
    public static void main(String[] args) {
    // Définition d'un tableau de 10 entiers
    int [] Tableau = {5, 3, 12, 4, 7, 9, 1, 8, 19, 2};

    }
}
```

3.2 Calcul de moyenne et d'écart type

Ecrire un programme qui calcule et affiche la valeur moyenne (M) et l'écart type (σ) d'une suite de valeurs réelles stockées dans un tableau. On rappelle que :

$$M = \frac{1}{n}\sum_{i=1}^{n} x_i \ et \ \sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2 - (\sum_{i=1}^{n}\frac{x_i}{n})^2}$$

3.3 Calcul de la valeur d'un polynôme

Ecrire un programme qui calcule la valeur en un point d'un polynôme et qui affiche le résultat de l'évaluation. Les coefficients du polynôme, à valeur réelle, sont stockés dans un tableau par puissance croissante. La valeur de calcul sera soit saisie au clavier soit codée directement dans le programme source. On utilisera, pour l'évaluation, la méthode de Horner :

$$P(x) = (...(a_n x + a_{n-1}) * x + a_{n-2}) * x + ...) * x + a_1) * x + a_0$$

3.4 Tri par sélection

Il s'agit d'écrire un programme réalisant le tri de données. La méthode de tri utilisée est basée sur la sélection du minimum. Pour chaque indice i du tableau, on recherche à partir de cet indice le plus petit élément du tableau et on permute cet élément avec celui se trouvant à l'indice i.

Question 1 Montrer le déroulement de l'algorithme sur la liste de nombres S = {8, 5, 7, 9, 2, 1, 12, 6}.

Question 2 Définir une classe `TriEntiers` permettant de trier des entiers stockés dans un tableau. L'en-tête de la méthode de tri est le suivant :

```
public void TriTableauEntiers(int [] Tableau)
```

Question 3 Tester votre programme à l'aide du programme TestTriEntiers (Annexe A.2).

Question 4 Ecrire une fonction permettant de trier des chaînes de caractères. On utilisera la méthode `compareTo` de la classe `String` pour comparer deux chaînes de caractères. L'en-tête de la fonction de tri est le suivant :

```
public void TriTableauChaines(String [] Tableau)
```

Question 5 Tester votre programme à l'aide du programme TestTriChaines (Annexe A.3).

Question 6 Calculer le nombre de comparaisons réalisées par la fonction de tri.

Chapitre 4

Programmation orientée objet : définition de classes, création d'objets, héritage, polymorphisme et classes abstraites

4.1 Calcul des impôts locaux

Dans le cadre de l'informatisation d'une mairie, on veut automatiser le calcul des impôts locaux. On distingue deux catégories d'habitation : les habitations à usage professionnel et les maisons individuelles, l'impôt se calculant différemment selon le type d'habitation. Pour cela, on définit les classes `HabitationProfessionnelle` et `HabitationIndividuelle` et les caractéristiques communes à ces deux classes sont regroupées dans la classe `Habitation`. On a donc un schéma de classes où les classes `HabitationProfessionnelle` et `HabitationIndividuelle` héritent de la classe `Habitation`.
L'objet de cet exercice est d'implémenter ce schéma d'héritage et de mettre en œuvre le mécanisme de liaison dynamique.

4.1.1 Définition de la classe Habitation

Objectif : Définir une classe avec un constructeur et créer une instance de cette classe

La classe `Habitation` comprend les attributs :

proprietaire du type chaîne de caractères et qui correspond au nom du propriétaire,

adresse du type chaîne de caractères et qui correspond à l'adresse de l'habitation,

surface du type double et qui correspond à la surface de l'habitation et qui permet de calculer le montant de l'impôt.

les méthodes :

double Impot() qui permet de calculer le montant de l'impôt que doit payer le propriétaire de l'habitation à raison de 2^F par m^2.

void Affiche() qui permet d'afficher les trois attributs de la classe Habitation.

et un constructeur à trois paramètres permettant d'initialiser une instance de la classe Habitation :
`Habitation(String P, String A, double S);`

Question 1 Définissez la classe Habitation

Question 2 Tester votre programme avec le code donné en annexe A.4.

4.1.2 Définition des classes HabitationIndividuelle et HabitationProfessionnelle

Objectif : Utiliser l'héritage pour définir de nouvelles classes, redéfinir des méthodes dans les classes héritières.

Le calcule de l'impôt d'une maison individuelle est différent de celui d'une habitation, il se calcule en fonction de la surface habitable, du nombre de pièces et de la présence ou non d'une piscine. On compte 100F/pièce et 500F supplémentaire en cas de présence d'une piscine.

Question 3 Définir la classe `HabitationIndividuelle` qui hérite de la classe Habitation en utilisant l'en-tête suivant :

```
public class HabitationIndividuelle extends Habitation {
    ...
};
```

Ajouter les attributs `NbPieces` de type entier et `Piscine` de type booléen. Redéfinir les méthodes `Impot` et `Affiche`. La méthode `Affiche` doit afficher, les attributs `proprietaire`, `adresse` et `surface` de la classe `Habitation`, et les attributs `NbPieces` et `Piscine` propres à la classe `HabitationIndividuelle`. La première ligne de la méthode `Affiche`, commencera par l'instruction `super.Affiche()`, permmettant d'appeler la méthode `Affiche` de la classe mère `Habitation`.

Question 4 Tester votre programme avec le code donné en annexe A.5.

Le calcule de l'impôt d'une habitation à usage professionnel est également différent de celui d'une habitation. Il se calcule en fonction de la surface occupée par le bâtiment et du nombre d'employés travaillant dans l'entreprise. On compte 1000F supplémentaire par tranche de 10 employés.

Question 5 Définir la classe HabitationProfessionnelle qui hérite de la classe Habitation en utilisant l'en-tête suivant :

```
public class HabitationProfessionnelle extends Habitation {
    ...
};
```

Ajouter l'attribut `NbEmployes` de type entier. Redéfinir les méthodes `Impot` et `Affiche`. La méthode `Affiche` doit afficher, en plus des attributs `proprietaire`, `adresse` et `surface`, l'attributs `NbEmployes`.

Question 6 Tester votre programme à l'aide du code donné en annexe A.6.

4.1.3 Gestion des habitations d'une commune

Objectif : Mettre en oeuvre le mécanisme de liaison dynamique.

On désire à présent calculer l'impôt local des habitations (individuelles ou professionnelles) d'une commune. Pour cela, on utilise une collection d'objets représentée par un tableau où chaque élément désigne une habitation individuelle ou professionnelle.

Question 7 Tester votre programme à l'aide du code donné en annexe A.7.

4.2 Gestion d'une collection d'objets

4.2.1 Définition de la classe Impair

Objectif : Créer une classe, et l'utiliser dans une fonction main.

On souhaite afficher les 100 premiers entiers impairs. Pour préparer les exercices suivants, on va le faire de façon un peu artificielle. On définira une classe `Impair`, qui représente les nombres impairs.

Pour disposer des nombre impairs, la classe fournit les méthodes suivantes :

Impair() : Le constructeur

int Premier() : Retourne le premier des nombres impairs (oui, oui, 1)

int Suivant() : Retourne le nombre impair suivant. La question c'est suivant quoi ?
- Si on venait d'appeler la méthode Premier(), le suivant est le deuxième nombre impair.
- Sinon, le suivant est le nombre impair qui suit celui qui a été retourné lors du dernier appel à la méthode Suivant().

boolean IlEnReste() : Dit si, vu les appels précédents aux méthodes `Premier()` et `Suivant()`, on peut encore demander un autre nombre impair. Bien que la suite des nombres impairs soit infinie, nous avons choisi de la limiter aux 100 premiers nombres impairs. Donc la fonction IlEnReste() retourne false si on dépasse cette limite, true si on est encore en mesure de donner un Suivant() inférieur à cette limite.

Question 1 Une fois la classe Impair définie, écrivez une classe Main avec une fonction main, comme on le fait usuellement pour démarrer un calcul. Cette fonction main devra :
- Allouer (new) une instance de la classe Impair, que l'on appellera nb.
- Faire une boucle while qui traduise l'algorithme suivant :
  `Tant qu'il reste des éléments à afficher dans nb, prendre l'élément suivant et l'afficher.`
  Vous utiliserez les méthodes `Premier`, `Suivant` et `IlEnReste` de l'objet nb.

4.2.2 Définition de la classe Casier

Objectif : Cet exercice fait travailler la notion de tableau, de boucle, en plus des notions de l'exercice précédent.

On souhaite définir une classe `Casier` qui permet de stocker 10 entiers, et de les récupérer en cas de besoin. Vous utiliserez un tableau. Pour stocker un entier `k` dans la case `i (0 <= i < 10 )`, la classe fournira la méthode :

```
void Range(int k,        // L'entier à ranger
           int i);       // Le numéro du tiroir !
```

Question 1 Pour lire les entiers stockés, on fournira des méthodes `Premier`, `Suivant`, `IlEnReste` qui s'utilisent comme celle de l'exercice précédent.

Question 2 Dans la fonction `main` de la classe `Main`, stockez quelques valeurs dans le casier, aux positions que vous désirez, et affichez l'ensemble du casier, de la même façon que vous aviez affiché les nombres impairs dans l'exercice précédent.

Quand ça marche, ajouter de quoi afficher la somme des entiers du casier.

Question 3 Faites apparaître proprement la taille maximale du tableau, de sorte à ne faire qu'une seule modification du code java le jour ou on veut la modifier.

4.2.3 Définition de la classe Syracuse

Objectif : Constructeur avec paramètre. Le but est aussi de vous faire constater qu'au cours des exercices, on écrit souvent la même chose (vive la fonction copier-coller des éditeurs de texte).

Une suite de Syracuse se calcule comme suit. On se donne un entier de départ arbitraire, $U(0)$. Pour un $U(n)$ donné, on calcule $U(n+1)$ comme suit :

$$Si\ U(n)\ est\ pair,\ U(n+1) = U(n)/2,\ sinon\ U(n+1) = 3 * U(n) + 1.$$

Question 1 Définissez un classe Syracuse dont le constructeur prend $U0$ en argument, et qui fonctionne à l'aide des fonctions `Premier()`, `Suivant()` et `IlEnReste()`. Cette classe fournit tour à tour les éléments de la suite de Syracuse commençant par le $U0$ passé au constructeur. On a constaté (mais personne ne sait le montrer), que les suites de Syracuse finissent par avoir les valeurs 4,2,1,4,2,1,4,2,1,..... On considérera qu'une fois qu'on a atteint la valeur 1, "Il n'en reste plus".

Question 2 Dans une fonction `main`, définissez deux suites de Syracuse différentes (elles diffèrent par leur terme U0), affichez-les, et calculez la somme des termes de la seconde en recopiant le calcul de la somme que vous aviez fait pour les Casiers.

4.2.4 Définition de la classe Entiers

Objectif : Définir une classe de base abstraite et établir une relation d'héritage avec les classes Impair, Casier et Syracuse.

Il devient lassant de toujours réécrire les mêmes boucles pour l'affichage et la somme, alors que d'un exercice à l'autre, seules les méthodes `Premier()`, `Suivant()`, et `IlEnReste()` changent.

Question 1 Définissez une classe Entiers qui impose l'existence des méthodes `public int Premier()`, `public int Suivant()` et `public boolean IlEnReste()` sans les définir (abstract), et qui définit de plus une fonction d'affichage des entiers que fournit la classe, `public void Affiche()`, ainsi que le calcul de leur somme `public int Somme()`.

Question 2 Ceci fait, réécrivez la classe Syracuse en la faisant hériter de Entiers. Ecrivez alors une fonction main, qui fait le même travail que celle de l'exercice précédent, mais cette fois-ci sans écrire de boucle.

Question 3 Faites de même pour les classes Casier et Impair... Elles savent s'afficher et calculer leur somme d'emblée !

4.2.5 Afficher les couples d'entiers

Objectif : Introduction à la notion d'itérateur.

Repartons de l'état de l'exercice précédent, et considérons Casier qui hérite maintenant de Entiers. On voudrait écrire une méthode `AfficheCouples()` de la classe Entiers qui affiche tout les couples que l'on puisse constituer à partir des entiers gérés.

Question 1 Essayez d'écrire cette fonction à l'aide de deux boucles for imbriquées et vérifiez dans un main qu'elle permet d'écrire les couples d'élément stockés dans un Casier... vous devriez avoir du mal a écrire la fonction. Si vous y parvenez, TESTEZ-LA.

Quel est le problème ?

4.2.6 Utilisation d'une classe itérateur

Objectif : Introduction à la notion d'itérateur.

Le problème posé dans l'exercice précédent est qu'il faut se remémorer le contexte du parcours des éléments, c'est-à-dire "où on en est". Stocker ce contexte comme variables membre des classes Impair, Casier et Syracuse, ce que vous avez du faire par des variables du genre `dernier_resultat_rendu`, ne marche que si on fait un parcours à la fois, mais ça exclut de faire deux parcours des éléments simultanés. Or, on a justement besoin de deux parcours simultanés quand on calcule les couples via deux boucles imbriquées.

La solution consiste à "externaliser" ces contextes, c'est-à-dire laisser les entiers à l'intérieur de la classe mais créer une autre classe afin de gérer les contextes. On appellera cette autre classe Iterateur. C'est maintenant cette classe qui va fournir les méthodes `Premier()`, `Suivant()` et `IlEnReste()`. La classe Entiers va alors uniquement se contenter de fournir (et d'allouer) des itérateur à la demande pour pouvoir imbriquer des parcours de ses éléments. On définira pour cela la méthode suivante de la classe Entiers : `Iterateur DonneIterateur()`

Les classes abstraites dont on va dériver `Impair`, `Casier`, et `Syracuse` sont au nombre de deux : `Iterateur` (annexe A.8) et `Entiers` (annexe A.9).

Question 1 Complétez la fonction AfficheCouples de la classe Entiers pour pouvoir afficher tous les couples constitués des entiers qu'elle sait fournir, puis faites hériter IterateurCasier et Casier respectivement d'Iterateur et d'Entiers. Dans le main, allouez un Casier, remplissez quelques cases, et affichez les couples. Ensuite, dans le main, calculez, à l'aide d'un itérateur le produit des éléments du casier.

Question 2 Quand cela marche, refaites les étapes précédentes pour les classes Impair et Casier.

Question 3 Enfin, quand vous avez terminé, revoyez la notion de colletion en Java...

Annexe A

Programmes de test

A.1 Programme de lecture d'un entier au clavier

```
import java.io.*;

public class LectureEntier {
  public static void main (String [] args) {
    // On commence par déclarer un objet lecteur sur le clavier
    // Le clavier est associé à la variable System.in
    // Un InputStreamReader (lecteur sur un flux) est créé dessus
    // Puis on ajoute une couche BufferedReader nécessaire pour lire des
    // informations par ligne
    BufferedReader lecteurClavier=new BufferedReader(new InputStreamReader(System.in));
    int            valeur = 0;
    String         chaineLue;

    try {
        // Lecture d'une ligne au clavier
        System.out.print("Entrer un entier : ");
        chaineLue = lecteurClavier.readLine();
        // Conversion de la chaine en entier
        valeur = Integer.parseInt(chaineLue);
    }
    catch (Exception e) {
        System.out.println("Erreur d'E/S " + e.getMessage());
    }
    System.out.println("La valeur est : " + valeur);
  }
}
```

A.2 Test du programme de tri des entiers

```
// classe permettant de tester la classe de tri des entiers
public class TestTriEntier {
    public static void main(String[] args) {
        // definition et initialisation du tableau à trier
        int [] Tableau = {12, 3, 1, 4, 9, 5, 2, 8, 3, 6};
        // Création d'un objet de type TriEntiers
        TriEntiers Tri = new TriEntiers();
```

```
        // on trie le tableau d'entiers
        Tri.TriTableauEntiers(Tableau);
        // on affiche le tableau trié
        for (int i = 0; i < Tableau.length; i++)
            System.out.print(Tableau[i] + " ");
    }
}
```

A.3 Test du programme de tri des chaînes de caractères

```
// classe permettant de tester la classe de tri des chaines de
// caractères
public class TestTriChaines {
    public static void main(String[] args) {
        // definition et initialisation du tableau à trier
        String [] Tableau = {"un", "deux", "trois", "quatre", "cinq"};
        // Création d'un objet de type TriChaines
        TriChaines Tri = new TriChaines();

        // on trie le tableau de chaines de caractères
        Tri.TriTableauChaines(Tableau);
        // on affiche le tableau trié
        for (int i = 0; i < Tableau.length; i++)
            System.out.print(Tableau[i] + " ");
    }
}
```

A.4 Test de la classe Habitation

```
// Classe TestHabitation permettant de tester la classe Habitation
public class TestHabitation{
    public static void main (String[] args){
        double I;

        // creation d'un objet de type Habitation
        Habitation H = new Habitation("Jean", "METZ", 120);
        // calcul de l'impôt
        I = H.Impot();
        // affichage des attributs de la classe Habitation
        H.Affiche();
    }
}
```

A.5 Test de la classe HabitationIndividuelle

```
// Classe TestHabitationIndividuelle pour tester la classe
// HabitationIndividuelle
public class TestHabitationIndividuelle{
    public static void main (String [] args){
        double I;

        // creation d'un objet de type HabitationIndividuelle
        HabitationIndividuelle HI = HabitationIndividuelle new("Paul", "METZ", 120, 5, False);
```

```
        // calcul de l'impôt
        I = HI.Impot();
        // affichage des attributs de la classe HabitationIndividuelle
        HI.Affiche();
    }
}
```

A.6 Test de la classe HabitationProfessionnelle

```
// Classe TestHabitationProfessionnelle permettant de tester la classe
// HabitationProfessionnelle
public class TestHabitationProfessionnelle {
    public static void main (String [] args){
        double I;

        // creation d'un objet de type HabitationProfessionnelle
        HabitationProfessionnelle HP = new HabitationProfessionnelle("ImportExport", "METZ", 2500, 130);
        // calcul de l'impôt
        I = HP.Impot();
        // affichage des attributs de la classe HabitationProfessionnelle
        HP.Affiche();
    }
}
```

A.7 Test de la classe Collection

```
// Définition de la classe TestCollection
 public class TestCollection{
    public static void main (String [] args){
        Habitation [] TableauHabitation;

        // creation d'un tableau contenant 5 habitations
        TableauHabitation = new Habitation [5];
        // Initialisation des éléments du tableau
        TableauBatiment[0] = new HabitationProfessionnelle("ImportExport", "METZ", 2500, 130);
        TableauBatiment[1] = new HabitationProfessionnelle("Export", "METZ", 250, 10);
        TableauBatiment[2] = new HabitationIndividuelle("Paul", "METZ", 100, 5, false);
        TableauBatiment[3] = new HabitationProfessionnelle("Import", "METZ", 1200, 90);
        TableauBatiment[4] = new HabitationIndividuelle("Jean", "METZ", 130, 6, true);
        // affichage des attributs de chaque élément du tableau
        for (int i = 0; i < 5; i++)
            TableauHabitation[i].Affiche();
       // calcul et affichage de l'impôt
        for (int i = 0; i < 5; i++)
            TableauHabitation[i].Impot();
    }
}
```

A.8 La classe Iterateur

```
import java.lang.*;
import java.util.*;
```

```
abstract class Iterateur {

    public Iterateur() {
    }

    public abstract int Premier();
    public abstract int Suivant();
    public abstract boolean IlEnReste();
}
```

A.9 La classe Entiers

```
import java.lang.*; import java.util.*;

abstract class Entiers {

    public Entiers() {
    }

    public abstract Iterateur DonneIterateur();

    public void Affiche()
    {
    Iterateur iter;

    iter=this.DonneIterateur();
    System.out.println(iter.Premier());
    while(iter.IlEnReste())
        System.out.println(iter.Suivant());
    }

    public int Somme()
    {
    int i,somme;
    Iterateur iter;

    iter=this.DonneIterateur();
    somme=0;
    for(i=iter.Premier();iter.IlEnReste();i=iter.Suivant())
        somme=somme+i;

    return somme;
    }

    public void AfficheCouples()
    {
        // à compléter
    }
}
```