Licence Professionnelle da2i
Université Lille 1
Sciences et technologies

DS 1 : Bases de la programmation objet

Durée : 2h
Documents autorisés – Machines interdites

Génie Logiciel
Licence SIL DA2I
2012–2013

– LISEZ ATTENTIVEMENT L'ÉNONCÉ !
– LES TROIS EXERCICES SONT INDÉPENDANTS
– ÉCRIVEZ LISIBLEMENT ET À L'ENCRE
(les réponses au crayon de bois ne feront l'objet d'aucune correction)
– MOINS IL Y A DE CODE, MIEUX C'EST !

1 Rendu de monnaie au moyen d'un itérateur *(5 points)*

On souhaite écrire un programme qui permette d'automatiser le problème du rendu de monnaie afin de contrôler un automate : étant donné un montant à payer, si le client fournit une somme supérieure à son dû, quels sont les billets et les pièces à lui rendre ? Il suffit bien sûr de connaître la somme à rendre ainsi que les espèces (billets et pièces) disponibles (dans toute la suite, on ne s'intéressera qu'à des valeurs entières, autrement dit on ignore les centimes).
En ce qui concerne l'algorithme à utiliser, on peut utiliser pour simplifier un algorithme « glouton », qui consiste à rendre **une à une** chaque espèce du montant maximal possible, jusqu'à ce qu'il ne reste plus de monnaie à rendre. Par exemple, dans le système monétaire de l'euro, pour rendre 427 €, on rendra successivement 200, 200, 20, 5 puis 2 euros.
Toutefois, dans un véritable automate, on dispose d'une **caisse** qui contient un certain nombre d'exemplaires de billets et de pièces de chaque valeur : on ne peut donc pas toujours rendre exactement la monnaie selon cette méthode.
Ainsi, si notre automate ne dispose pas de billets de 200 € et de seulement 3 billets de 100 €, le rendu sera le suivant : 427 € = 100 + 100 + 100 + 50 + 50 + 20 + 5 + 2.

On a écrit le squelette d'une classe réalisant cette opération au moyen d'un **itérateur d'entiers** :

Monnayeur.java

```
import java.util.Iterator ;

public class Monnayeur implements Iterator<Integer> {
    public static final int [] EUROS = {500, 200, 100, 50, 20, 10, 5, 2, 1} ;
    // ... autres attributs ...
    /** Crée un monnayeur pour une certaine somme, et disposant du nombre
     *  de pièces spécifié dans le tableau caisse (dans le même ordre que EUROS) */
    public Monnayeur(int somme_a_rendre, Integer[] caisse) {
        // ... à compléter ...
    }
    // ... à compléter ...
}
```

TestMonnayeur.java

```
/** Un test qui lit sur la ligne de commande : 1° le montant à rendre,
 * 2° le contenu de la caisse pour chaque valeur de billet ou de pièce
 * dans l'ordre décroissant */
public class TestMonnayeur {
    public static void main(String [] args) {
        Integer[] caisse = new Integer[args.length-1] ;
        for (int i=1; i<args.length; i++)
            caisse[i-1] = new Integer(args[i]) ;
        Monnayeur m = new Monnayeur(new Integer(args[0]), caisse) ;
        while (m.hasNext())
            System.out.print(m.next() + "\t") ;
        // en cas de problème :
        System.out.println("\nSomme non rendue : " + m.next()) ;
    }
}
```

Résultat du test

```
$ java Monnayeur 427 5 5 5 5 5 5 5 5 5
200     200     20      5       2
Somme non rendue : 0
$ java Monnayeur 427 5 0 3 5 5 5 5 5 5
100     100     100     50      50
20      5       2
Somme non rendue : 0
$ java Monnayeur 427 5 0 3 5 5 5 5 0 0
100     100     100     50      50
20      5
Somme non rendue : 2
```

Q 1. Complétez la classe `Monnayeur`, de façon à ce que votre code soit conforme aux tests ci-dessus. La méthode `hasNext` renvoie `true` lorsqu'il reste des pièces ou des billets à rendre, et `false` lorsque le rendu est terminé ou lorsqu'il n'est plus possible (par exemple parce que la caisse est vide). Chaque appel à `next` retourne une valeur de billet ou de pièce à rendre ; lorsque le rendu est terminé ou impossible, `next` doit retourner le montant non rendu. Bien évidemment la méthode `remove` ne sert pas.

---

2 Réalisation d'une file par un tableau *(7 points)*

File.java

```
public interface File<E> {
    /** Nombre maximal d'éléments pouvant être stockés dans la file */
    int TAILLE_MAX = 5 ;
    /** Donne le nombre d'éléments stockés dans la file */
    int nbElements() ;
    /** Indique si la file est vide */
    boolean vide() ;
    /** Indique si la file est pleine */
    boolean pleine() ;
    /** Ajoute un élément à la fin de la file (ne fait rien si la file
     * est pleine) */
    void ajouter(E c) ;
    /** Renvoie l'élément au début de la file (sans le retirer) ; null
     * si la file est vide */
    E premier() ;
    /** Renvoie l'élément au début de la file et le retire ; null si
     * la file est vide */
    E enlever() ;
    /** Indique si l'élément spécifié est présent dans la file */
    boolean contient(E c) ;
    /** Ordonne les éléments de la file dans l'ordre croissant */
    void trier() ;
    /** Itérateur permettant de parcourir les éléments de la file */
    java.util.Iterator<E> iterator() ;
}
```

Une *file* est un type abstrait de données qui permet de stocker séquentiellement des éléments *de même type*, et dont le mode d'accès est dit *FIFO* (*First In, First Out* : premier arrivé, premier sorti). Autrement dit, les nouveaux éléments sont placés à la fin de la file tandis que la lecture d'un élément ou son retrait de la file se font par le début de la file.

On a défini l'interface ci-contre.

On souhaite réaliser une **file de caractères** au moyen d'un tableau de taille fixe. Pour éviter de devoir décaler tous les éléments lorsqu'on en ajoute ou lorsqu'on en retire un, on décide de mémoriser un indice `debut` repérant la position du premier élément et on travaille *comme si le tableau était circulaire*, c'est-à-dire que lorsqu'on arrive au « bout » du tableau, on continue à l'indice 0 (cf. TAB 1 page 3).

On donne le début du code de la classe `FileTableau` réalisant cette solution :

Squelette de `FileTableau.java`

```
public class FileTableau implements File<Character> {
    private int nbElements ;
    private int debut ;
    private char [] elements ;

    /** Constructeur créant une file vide */
    public FileTableau() {
        nbElements = 0 ; debut = 0 ;
        elements = new char[TAILLE_MAX] ;
    }

    /** Représentation sous forme de chaîne */
    public String toString() {
        if (this.vide())
            return "File vide" ;
        StringBuffer s = new StringBuffer() ;
        int position = debut ;
        for (int i = 1; i <= nbElements; i++) {
            s.append(" " + elements[position]) ;
            position = (position + 1) % TAILLE_MAX ;
        }
        return s.toString() ;
    }
}
```

L'indice de fin peut être calculé en utilisant le nombre d'éléments. Bien entendu, il est possible qu'on soit obligé de placer certains éléments « à gauche » de l'indice `debut` s'il ne reste plus assez de place « à droite ». En voici une illustration :

TABLE 1 – Manipulation d'une file (ici avec `TAILLE_MAX = 5`). L'élément en gras est situé à l'indice `debut` et les cases vides représentent les portions du tableau qui ne sont pas utilisées (leur contenu n'est pas forcément `null`).

Opération	**Tableau** `elements`					`nbElements`	**Appel à** `toString()`	
indices :	0	1	2	3	4			
(1) État initial			**A**	Z		2	`A Z`	
(2) `ajouter(E)`			**A**	Z	E	3	`A Z E`	
(3) `ajouter(R)`	R		**A**	Z	E	4	`A Z E R`	
(4) `ajouter(T)`	R	T	**A**	Z	E	5	`A Z E R T`	*(file pleine)*
(5) `enlever()`	R	T		**Z**	E	4	`Z E R T`	
(6) `trier()`	T	Z		**E**	R	4	`E R T Z`	
(7) `enlever()`	T	Z			**R**	3	`R T Z`	
(8) `ajouter(Y)`	T	Z	Y		**R**	4	`R T Z Y`	

Q 2. Écrivez pour `FileTableau` une méthode `int indiceFin()` qui retourne *le premier indice libre à la fin de la file* (ou `debut` si la file est pleine). Par exemple, dans l'état 1 de l'exemple ci-dessus, `indiceFin()` retourne 4, puis dans l'état 2, retourne 0. Cette méthode doit-elle être publique, privée, protégée... ? **Justifiez votre réponse.** *(2 points)*
N.B. : on rappelle que l'opérateur modulo (reste de division euclidienne) se note `%` en Java.

Q 3. Écrivez le code des méthodes suivantes de `FileTableau` en respectant l'interface : *(5 points)*
– `ajouter`
– `enlever`
– `trier` (algorithme de tri de votre choix, sans utiliser `Arrays` ni `Collections`)
Vous pouvez faire appel à toutes les méthodes dont la signature est dans `File` en supposant que leur code a déjà été écrit correctement dans `FileTableau`, ainsi qu'à des méthodes supplémentaires (dont la signature n'est pas dans `File`) à condition d'en donner le code.

---

3 Structure de données répondant à des spécifications *(8 points)*

Contexte

Dans de nombreux jeux ou sports (échecs, dames, go, sumo, etc.), les adversaires en compétition dans un tournoi s'affrontent deux par deux successivement (on ne peut pas jouer contre soi-même). L'un des deux gagne, l'autre perd car on trouve toujours un moyen de départager les parties nulles. Tous les concurrents ne se rencontrent pas forcément deux à deux systématiquement (X peut ne jamais affronter Y). À la fin du tournoi, l'ordre dans lequel se sont déroulées les parties n'a pas d'importance, mais on veut être capable d'évaluer leur résultat détaillé, de façon à pouvoir dire :
– combien de parties ont été jouées, gagnées, perdues par un joueur j_1 contre un joueur j_2 ;
– combien de parties ont été jouées, gagnées, perdues en tout par un joueur j ;
– combien de parties ont été jouées, gagnées, perdues en tout.

Pour simplifier nous considérerons ici qu'un joueur est représenté par son numéro (de 0 à $(n-1)$ s'il y a n joueurs).

Par exemple, on a quatre joueurs ($0 \ldots 3$), et les parties se déroulent de la façon suivante :
– 0 gagne deux fois contre 2, puis il perd contre 1 et contre 3
– 1 gagne ensuite contre 0 et 2
– 2 gagne ensuite contre 1, mais perd contre 0
– 3 gagne ensuite contre 0 et 1, puis perd trois fois de suite contre 2.
Suite à ce tournoi, on peut dire, entre autres :
– que le joueur 3 a joué 6 parties ;
– que le joueur 2 a joué 2 parties avec le joueur 1 ;
– que le joueur 0 a gagné 3 fois contre le joueur 2, et perdu 4 fois en tout ;
– que 13 parties en tout ont été jouées.

Cahier des charges

On veut développer des applications permettant à des joueurs de s'affronter dans ce genre de tournoi. Pour uniformiser le développement, on impose aux programmeurs de respecter l'interface suivante :

Tournoi.java

```
public interface Tournoi {
    /** mémorise une victoire du joueur joueur1 sur le joueur joueur2 */
    void victoire(int joueur1, int joueur2) ;
    /** nombre de joueurs inscrits dans le tournoi */
    int nbJoueurs() ;
    /** nombre de parties gagnées par joueur1 contre joueur2 */
    int nbGagnees(int joueur1, int joueur2) ;
    /** nombre de parties perdues par joueur1 contre joueur2 */
    int nbPerdues(int joueur1, int joueur2) ;
    /** nombre de parties jouées par joueur1 contre joueur2 */
    int nbJouees(int joueur1, int joueur2) ;
    /** nombre de parties gagnées par un joueur */
    int nbGagnees(int joueur) ;
    /** nombre de parties perdues par un joueur */
    int nbPerdues(int joueur) ;
    /** nombre de parties jouées par un joueur */
    int nbJouees(int joueur) ;
    /** nombre de parties qui ont été jouées en tout */
    int nbJouees() ;
    /** numéro du joueur contre lequel la probabilité de victoire est la plus élevée
     * @return -1 si aucun adversaire connu */
    int suggererAdversaire(int joueur) ;
}
```

On voit qu'une fonctionnalité supplémentaire a été introduite : à partir de l'enregistrement des scores, on peut suggérer à un joueur j d'affronter un joueur $j_{\max}$, calculé pour être, d'après les statistiques, *celui que j a le plus de chances de battre*. Pour cela, on considèrera que la probabilité que le joueur j batte un joueur k contre lequel il a déjà joué est :

$$p_{\text{battre}(j,k)} = \frac{V_{j,k}}{P_{j,k}}$$

où $V_{j,k}$ désigne le nombre de victoires de j sur k, et $P_{j,k}$ le nombre de parties jouées entre j et k.
Si le joueur j n'a jamais joué contre k, il considérera qu'il n'a aucune chance de le battre. Si j n'a joué contre personne, la méthode `suggererAdversaire(`j`)` doit retourner -1.

Problème à résoudre

Q 4. Décrivez en quelques lignes en français la structure de données *la plus simple possible* qui permettrait de mémoriser les parties d'un tournoi, de façon à pouvoir fournir les fonctionnalités de l'interface ci-dessus. Indiquez ensuite (par un schéma par exemple) le contenu de cette structure après le tournoi donné en exemple ci-dessus. *(2 points)*

Indication : Prenez le temps de réfléchir à la structure de données la mieux adaptée au problème ! Une structure de données inadéquate imposera une implémentation très compliquée !...

Q 5. Écrivez une classe `MemoTournoi` implémentant l'interface `Tournoi` ci-dessus. Vous utiliserez les attributs *les plus simples possibles*. Par ailleurs on impose que le seul constructeur de cette classe ait exactement la signature suivante : `public MemoTournoi(int nbJoueurs)`. *(6 points)*