Maximot 1

TP du module 1 - Les bases du Java

Proposition de solution

2 - La fonction `tirerMotAleatoirement()` :

```
public static final String FICHIER_MOTS = "./dictionnaire.txt";
public static final int NB_MOTS = 22506;
public static Random r = new Random();

/**
 * Retourne un tableau de caractères contenant les lettres
 * du mot tiré au sort dans le fichier dictionnaire
 *
 * @return un tableau de caractères contenant les lettres
 *         du mot tiré au sort dans le fichier dictionnaire
 * @throws IOException
 *             s'il y a un problème d'accès au fichier dictionnaire
 */
private static char[] tirerMotAleatoirement() throws IOException {
      int numMot = r.nextInt(NB_MOTS);
      try (FileInputStream fichier = new FileInputStream(FICHIER_MOTS);
                                                Scanner s = new Scanner(fichier)) {
            for (int i = 1; i <= numMot; i++)
                  s.nextLine();
            return s.nextLine().toUpperCase().toCharArray();
      }
}
```

D'après le chemin indiqué par la constante ***FICHIER_MOTS***, le fichier dictionnaire.txt doit être positionné à la racine du projet Eclipse.

Le nombre de mots présents dans ce fichier est conservé dans une constante. Cela est un compromis :

- Cela évite un premier parcours pour connaître le nombre de mots présents dans le dictionnaire ;
- Néanmoins, si le fichier évolue avec des mots en plus ou en moins, il est nécessaire de modifier la valeur de cette constante.

La fonction est susceptible de lever des exceptions de type **`IOException`** (IO pour Input/Output, c'est-à-dire des erreurs d'entrées/sorties). Cela est dû à l'accès au fichier contenant les mots du dictionnaire. Il est difficile de traiter ces erreurs au sein de cette méthode. S'il est impossible d'accéder aux mots du dictionnaire, comment peut-on procéder ? Il aurait été possible, dans ce cas, de retourner un tirage de secours, mais ce n'est pas ce choix qui a été effectué. Ces erreurs n'ayant pas été traitées au sein de cette méthode, elles sont donc propagées.

Pour lire le fichier, la méthode d'accès au fichier utilisée est celle avec gestion automatique des ressources. Cela permet d'une part d'avoir un code plus concis et d'autre part d'être sûr de ne pas oublier de libérer les ressources nécessaires à la lecture du fichier.

3 - La fonction `melanger()` :

```
/**
 * Créer un nouveau tableau de caractères contenant les mêmes caractères
 * que ceux présents dans le tableau passé en paramètre, mais ceux-ci sont
 * positionnés dans un ordre aléatoire
 *
 * @param mot
 *            les caractères du mot à mélanger
 * @return un nouveau tableau dont les caractères
 *         ont été mélangés aléatoirement
 */
private static char[] melanger(char[] mot) {
	// clonage du tableau
	char[] mel = new char[mot.length];
	for (int i = 0; i < mel.length; i++) {
		mel[i] = mot[i];
	}
	// échanges de position de caractères
	for (int i = 0; i < mel.length * 4; i++) {
		int p1 = r.nextInt(mel.length);
		int p2 = r.nextInt(mel.length);
		char tmp = mel[p1];
		mel[p1] = mel[p2];
		mel[p2] = tmp;
	}
	return mel;
}
```

Il faut bien avoir en tête qu'un tableau est un type référence. Si les modifications sont effectuées sur le tableau passé en paramètre, cela sera réalisé sur l'original du tableau et non sur une copie. La première étape consiste donc à dupliquer le tableau.

La seconde étape consiste à mélanger les caractères présents dans ce nouveau tableau. Pour cela, deux indices du tableau sont tirés aléatoirement. Les caractères présents à ces deux positions sont alors échangés. Cette opération doit être réalisée un nombre suffisant de fois pour que la probabilité que les caractères restent à la même position soit très faible. Le choix a été fait d'effectuer quatre fois plus de permutations que le nombre de caractères présents dans ce tableau.

4 - La procédure `afficher()` :

```
/**
 * Affiche les caractères présents dans le tableau les uns
 * à la suite des autres puis retourne à la ligne
 *
 * @param mot
 *            le mot à afficher
 */
private static void afficher(char[] mot) {
	for (int i = 0; i < mot.length; i++) {
		System.out.print(mot[i]);
	}
```

```
		System.out.println();
	}
```

Normalement, pas de difficulté notable pour cette fonction.

5 - La fonction `bonnesLettres()` :

```
	public static final char VIDE = ' ';

	/**
	 * Vérifie si tous les caractères présents dans le tableau de caractères
	 * {@code prop} sont présents dans le tableau de caractères {@code tirage}.
	 * Un caractère présent plusieurs fois dans {@code prop} doit être présent
	 * au moins autant de fois dans {@code tirage}.
	 *
	 * @param prop
	 *            le tableau avec les caractères à vérifier
	 * @param tirage
	 *            le tableau avec les caractères à utiliser
	 * @return vrai si uniquement les caractères présents dans {@code tirage}
	 *         sont utilisés dans {@code prop}
	 */
	private static boolean bonnesLettres(char[] prop, char[] tirage) {
		// clonage du tableau des lettres à utiliser
		char[] copie = new char[tirage.length];
		for (int i = 0; i < copie.length; i++) {
			copie[i] = tirage[i];
		}
		// vérification de chaque lettre de la proposition
		int j = 0;
		boolean ok = true;
		while (ok && j < prop.length) {
			int k = 0;
			while (k < copie.length && prop[j] != copie[k]) {
				k++;
			}
			if (k == copie.length)
				ok = false;
			else {
				copie[k] = VIDE;
				j++;
			}
		}
		return ok;
	}
```

Une copie du tableau contenant les lettres à utiliser est effectuée afin de pouvoir retirer de la copie les lettres au fur et à mesure de leur utilisation. Ces manipulations sont effectuées sur une copie pour conserver le tableau contenant le tirage dans son état initial (cf. correction de la méthode `melanger()`).

Chaque caractère du tableau des lettres à vérifier est recherché dans la copie du tableau des lettres à utiliser et retiré de celui-ci. Dès qu'une lettre n'est pas présente, la méthode retourne faux. Si toutes les lettres ont été vérifiées et trouvées, alors la méthode retourne vrai.

6 - La fonction `dansLeDico()` :

```
/**
 * Indique si le mot {@code prop} passé en paramètre
 * est présent dans le dictionnaire
 *
 * @param prop
 *            le mot à vérifier
 * @return vrai si le mot est présent dans le dictionnaire
 * @throws IOException
 *             s'il y a un problème d'accès au fichier dictionnaire
 */
private static boolean dansLeDico(char[] prop) throws IOException {
	boolean trouve = false;
	try (FileInputStream fichier = new FileInputStream(FICHIER_MOTS);
                                                Scanner s = new Scanner(fichier)) {
		char[] motDico;
		while (s.hasNext() && !trouve) {
			motDico = s.nextLine().toUpperCase().toCharArray();
			trouve = Maximot.sontIdentiques(prop, motDico);
		}
	}
	return trouve;
}

/**
 * Compare deux tableaux de caractères et retourne vrai si les deux tableaux
 * de caractères sont identiques (même nombre de valeurs et mêmes valeurs)
 *
 * @param mot1
 *            le premier mot à comparer
 * @param mot2
 *            le second mot à comparer
 * @return vrai si les deux tableaux de caractères sont identiques
 */
private static boolean sontIdentiques(char[] mot1, char[] mot2) {
	boolean ok = mot1.length == mot2.length;
	if (ok) {
		int i = 0;
		while (ok && i < mot1.length) {
			ok = mot1[i] == mot2[i];
			i++;
		}
	}
	return ok;
}
```

Les erreurs d'accès au fichier dictionnaire sont propagées de la même manière que pour la fonction `tirerMotAleatoirement()` et la lecture du fichier est également gérée avec gestion automatique des ressources (cf. correction de la méthode `tirerMotAleatoirement()`).

Le choix a été fait de créer une fonction `sontIdentiques()` permettant de comparer deux tableaux de caractères, mais il est également possible d'intégrer ces instructions au sein de la fonction `dansLeDico()`.

La fonction `sontIdentiques()` commence par comparer le nombre d'éléments des deux tableaux. Si ce nombre est différent, il n'est pas nécessaire de comparer les caractères présents dans les tableaux : les mots sont forcément différents.

7 - La procédure principale :

```
/**
 * Ce programme permet de jouer au Maximot. Le joueur doit trouver un mot
 * le plus long possible à partir d'un tirage de lettres. Ce mot doit exister
 * dans le dictionnaire. Plus le mot est long, plus il rapporte de points.
 *
 * @param args non nécessaire
 */
public static void main(String[] args) {
	try {
		char[] mot = tirerMotAleatoirement();
		char[] tirage = melanger(mot);
		System.out.println("Voici le tirage :");
		afficher(tirage);
		Scanner console = new Scanner(System.in);
		System.out.println("Quel est le mot caché dans ce tirage ?");
		char[] prop = console.nextLine().toUpperCase().toCharArray();
		if (!bonnesLettres(prop, tirage)) {
			System.out.println("Lettre incorrecte !");
		} else {
			if (!dansLeDico(prop)) {
				System.out.println("Mot non présent dans le dico");
			} else {
				System.out.println("Bravo, vous marquez " +
					prop.length + " points");
			}
		}
		afficher(mot);
		console.close();
	} catch (IOException e) {
		System.err.println("Problème de lecture du dictionnaire");
	}
}
```

La procédure principale fait appel aux procédures et fonctions précédemment créées et en particulier à deux fonctions levant des exceptions. Tout le code est donc entouré d'un bloc `try … catch` afin de traiter ces erreurs. Le traitement est assez rustique, puisque seul un message d'erreur est affiché à l'utilisateur.