# Séquence 2

# Utiliser un scanner

## 1 Quel est le problème ?

En Java, comme dans beaucoup d'autres langages il y a une entrée standard et une sortie standard. Par défaut, c'est la console Java qui est utilisée à la fois comme entrée standard et sortie standard.

Ce sont deux objets qui s'appellent respectivement `System.in` et `System.out`. Mais alors que `System.out` fournit plusieurs méthode `print` et `println` qui permettent d'afficher les valeurs des types primitifs et des chaînes de caractères, `System.in` ne permet pas de lire directement au clavier des entiers, des booléens ou des chaînes de caractères. Les seules choses que l'ont peut lire au clavier sont des suites d'octets.

Pour lire au clavier les données qui nous intéressent il faut créer et utiliser un objet spécial en instanciant la classe `java.util.Scanner`. Cet objet va nous fournir les méthodes qui manquaient pour lire des données.

## 2 Création d'un scanner

Comme pour d'autres classes (java.util.ArrayList par exemple), pour pouvoir l'utiliser, il faut commencer par l'importer, sinon Java ne la trouve pas.

En début de fichier (avant `public class`), il faut la clause suivante :

```
import java.util.Scanner;
```

L'instanciation qui permet de créer l'objet scanner qui lit au clavier est la suivante :
`new Scanner(System.in);` Cela crée un objet. Pour pouvoir l'utiliser, il est judicieux de mémoriser cet objet dans une variable. Cela peut être fait comme suit :

```
Scanner scan = new Scanner(System.in);
```

## 3 Méthodes de lecture des données

Le scanner dispose de méthodes différentes pour lire des données de types différents :

— `nextInt` va renvoyer un entier lu au clavier
— `nextDouble` va renvoyer un nombre à virgule lu au clavier
— `nextBoolean` va renvoyer un booléen lu au clavier
— `nextLine` va renvoyer une chaîne de caractère lue au clavier

À noter : il n'existe pas de méthode pour lire un caractère, pas de `getChar`. Pour lire un caractère, il faut lire une chaîne et en extraire le premier caractère.

Si ce qui est tapé ne correspond pas au type exigé par la méthode employée, une exception `InputMismatchException` ets levée. Par exemple, si la méthode en cours d'exécution est `nextInt` et que l'on tape des lettres, c'est ce qui arrive.

```
import java.util.Scanner;
public class UseScanner1 {

	public static void main(String[] args) {
		Scanner scan = new Scanner(System.in);
		int nombre;
		System.out.print("Entrez␣un␣nombre␣entier:␣");
		nombre = scan.nextInt();
		System.out.println(nombre);
		scan.close();
	}
}
```

Et voyons ensuite deux exécutions du programme :

```
> java UseScanner1
Entrez un nombre entier: 47
47
> java UseScanner1
Entrez un nombre entier: abc
Exception in thread "main" java.util.InputMismatchException
at java.util.Scanner.throwFor(Scanner.java:864)
at java.util.Scanner.next(Scanner.java:1485)
at java.util.Scanner.nextInt(Scanner.java:2117)
at java.util.Scanner.nextInt(Scanner.java:2076)
at UseScanner1.main(UseScanner1.java:8)
```

# 4 Méthode de test du contenu

Pour éviter une exception`InputMismatchException`, on peut tester le type de ce qui est entré au clavier avec une méthode. Pour les entiers, cette méthode se nomme `hasNextInt` et elle renvoie un booléen.

Voyons un exemple de l'utilisation de cette méthode.

```
import java.util.Scanner;
public class UseScanner2 {
	public static void main(String[] args) {
		Scanner scan = new Scanner(System.in);
		int nombre;
		System.out.print("Entrez␣un␣nombre␣entier:␣");
		if (scan.hasNextInt()) {
			nombre = scan.nextInt();
			System.out.println("Nombre␣lu:␣"+nombre);
		}else {
			System.out.println("Vous␣avez␣tapé␣autre␣chose␣qu'un␣entier.");
```

```
                }

                scan.close();
        }
}
```

Et l'exécution :

```
> java UseScanner2
Entrez un nombre entier: abc
Vous avez tapé autre chose qu'un entier.
```

Notez que les deux méthodes `nextInt` et `hasNextInt` sont blocante : tant qu'on n'a pas tapé une ligne au clavier, le programme qui les exécute est stoppé et mis en attente.

## 5 Lecture du scanner et consommation des symboles

Ce qui est tappé au clavier n'est lu par le scanner qu'après un passage à la ligne. Mais une lecture du scanner ne *consomme* pas nécessairement la ligne entière. Voyons cela avec une lecture en boucle de nombres à virgule.

```
Entrez cinq nombres à virgule: 1,2 5,6
Nombre lu: 1.2
Nombre lu: 5.6
7,8 8,9
Nombre lu: 7.8
Nombre lu: 8.9
5,6
Nombre lu: 5.6
```

On voit sur cet exécution que les deux nombres `1,2` et `5,6` tapés sur la même ligne sont lus à deux tours de boucles différents par deux `nextDouble` successifs.

Le scanner a une notion de symbole (token en anglais) qui correspond à un élément unitaire. Cela peut être un nombre entier, un nombre à virgule, un boolean, une suite de lettres, une suite de caractères de ponctuation, un une suite comportant des lettres des chiffres et des caractères de ponctuation mélangés.

Plus précisément, pour le scanner, un token est toute suite de caractères terminée par un espace au sens large (qui peut être un espace, un retour à la ligne ou une tabulation).

Le scanner, lors de l'appel d'une méthode `nextXXX`, peut *consommer* un retour à la ligne situé avant le token lu, il ne consomme pas le retour à la ligne qui suit sauf dans le cas de la méthode `nextLine`.

## 6 Le piège du retour à la ligne

Cette non consommation du retour à la ligne qui suit un nombre produit un piège que nous allons illustrer avec le programme suivant. Le but de ce programme est de lire au clavier `n` noms, ce nombre `n` étant lu au clavier avant les noms en question.

```
import java.util.Scanner;
public class PiegeScanner {
```

```
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        int nb;
        String[] tab;
        System.out.print("Combien␣de␣noms?␣");
        nb = scan.nextInt();
        tab = new String[nb];
        for (int i=0; i<nb; i++) {
            System.out.print("Entrez␣le␣nom␣numéro␣"+i+":␣");
            tab[i] = scan.nextLine();
        }
        for (int i=0; i<nb; i++) {
            System.out.println("Nom␣numéro␣"+i+":␣" + tab[i]);
        }
        scan.close();
    }

}
```

Et voici une exécution :

```
> java PiegeScanner
Combien de noms? 3
Entrez le nom numéro 0: Entrez le nom numéro 1: riri
Entrez le nom numéro 2: fifi
Nom numéro 0:
Nom numéro 1: riri
Nom numéro 2: fifi
```

Vous voyez qu'au premier tour de boucle, le programme n'attend pas que l'on tape le premier nom, il continue. Au deuxième tour de boucle, il s'arrête et lit un nom et de même au troisième tour de boucle.

L'explication est qu'au premier tour de boucle, il reste un retour à la ligne qui a été tapé après le 3 et pas consommé par le scanner. Celui-ci est lu lors de l'appel à `nextLine` lors du premier tour de boucle. Cette méthode en l'occurrence lit une ligne vide et cela se traduit lors de l'affichage du nom numéro 0.

Plusieurs remèdes sont possibles pour éviter le problème : par exemple, lire la ligne vide avant d'entrer dans la boucle avec un `nextLine` ou tester que la ligne lue n'est pas vide et recommencer la lecture le cas échéant.

# 7 Unicité du scanner

Techniquement, il est possible de créer plusieurs objets scanner dans un programme, mais cela se traduit par un gaspillage de ressources. Il faut donc autant que possible s'assurer qu'on ne crée qu'un scanner par exécution du programme et réutiliser toujours le même scanner pour toutes les lectures au clavier.

La méthode `close` pour fermer un scanner lorsqu'on n'en a plus besoin est conseillée. Elle libère une ressource système. On ne peut plus rien lire une fois le scanner fermé.