IPO – TP 6 : interfaces

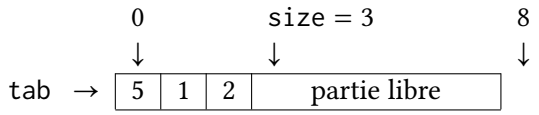
http://www.lri.fr/~blsk/IPO/

Ensembles On veut manipuler des ensembles d'entiers. L'objectif de cet exercice est de définir une interface pour une telle structure, et deux concrétisations différentes de cette interface.

1. Définir une interface `Set` pour une structure de données représentant un ensemble d'entiers (type `Integer`). On veut au minimum des méthodes correspondant aux descriptions suivantes :
   - une méthode `contains` indiquant si un élément donné appartient ou non à l'ensemble,
   - une méthode `cardinal` donnant le nombre d'éléments d'un ensemble,
   - une méthode `toString` qui renvoie une chaîne de caractères décrivant l'ensemble des éléments de l'ensemble (dans un ordre arbitraire),
   - une méthode `add` qui modifie un ensemble en lui ajoutant un élément, et une méthode `remove` qui retire un élément,
   - une méthode `clone` qui renvoie une copie d'un ensemble, de sorte que les modifications de l'une des copies n'ait pas d'influence sur les autres copies ni sur l'objet d'origine,
   - une méthode `union` renvoyant un nouvel ensemble obtenu par union de l'ensemble courant et d'un autre donné en paramètre, sans modifier aucun de ces ensembles d'origine, et une méthode `intersection` similaire.
2. Écrire dans une classe principale une méthode qui teste quelques opérations de cette structure (vous ne pourrez pas l'exécuter tout de suite, à défaut d'une concrétisation des ensembles).

On propose une première réalisation utilisant un tableau `tab` sous-jacent de taille fixe, et un attribut `size` donnant le nombre de cases de `tab` effectivement utilisées (en supposant, comme dans l'exemple de tableau associatif vu en cours, que les éléments utilisés sont dans les `size` premières cases).

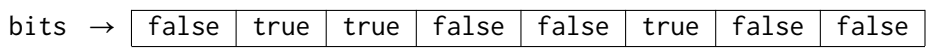
Une représentation possible de l'ensemble $\{1, 2, 5\}$ avec un tableau sous-jacent de taille 8 :



3. Définir une classe `TabSet` concrétisant l'interface `Set`, avec les caractéristiques suivantes :
   - utilisation d'un tableau primitif `tab` et d'un attribut `size`,
   - la taille (fixe) de `tab` est donnée en paramètre au constructeur,
   - un entier donné ne peut apparaître qu'une seule fois dans l'ensemble,
   - on s'autorise à échouer si un utilisateur tente d'ajouter des éléments à un `TabSet` dont le tableau sous-jacent est déjà plein.

   Indication : la méthode `union` doit avoir la signature `public Set union(Set s)`. Elle ne peut donc utiliser que les éléments de l'interface, et aucune connaissance spécifique à la classe `TabSet`.
4. Exécuter les tests définis précédemment sur l'interface `Set` en utilisant cette concrétisation.

On propose une deuxième réalisation utilisant un tableau `bits` de booléens, dans lequel `bits[k]` vaut **`true`** si et seulement l'ensemble représenté contient l'entier `k`. La représentation de l'ensemble $\{1, 2, 5\}$ par un tel tableau de taille 8 serait :



5. Définir une classe `BitSet` concrétisant l'interface `Set`, avec les caractéristiques suivantes :
   - utilisation d'un tableau primitif de booléens `bits`,
   - la taille (fixe) de `bits` est donnée en paramètre au constructeur.
6. Exécuter à nouveau les tests de `Set` avec cette nouvelle version.

Question supplémentaire, à garder pour quand vous aurez terminé la page suivante.

7. Définir les méthodes `equals` des classes `TabSet` et `BitSet`.

Itérateurs On s'intéresse maintenant aux deux interfaces `Iterable` et `Iterator` fournies par le langage Java (`java.util.Iterable` et `java.util.Iterator`), qui permettent d'itérer sur une collection à l'aide d'une boucle *for each*.

```
for (Integer k : set) {
  ...
}
```

Une telle boucle énumère tous les éléments d'un ensemble `set` d'entiers. On peut l'écrire en Java dès lors que l'objet `set` appartient à une classe `C` qui concrétise l'interface `Iterable<Integer>` des ensembles énumérables d'entiers.

Pour concrétiser l'interface `Iterable<Integer>`, une classe comme `TabSet` ou `BitSet` doit définir une méthode

```
public Iterator<Integer> iterator() { ... }
```

Cette méthode construit et renvoie un *itérateur*, c'est-à-dire un objet chargé d'énumérer tous les éléments de l'ensemble. L'itérateur est un objet d'une classe concrétisant l'interface `Iterator<Integer>`. Une telle classe doit fournir une méthode

```
public boolean hasNext() { ... }
```

qui renvoie **`true`** si et seulement il reste au moins un élément à énumérer, et une méthode

```
public Integer next() { ... }
```

qui renvoie le prochain entier de l'énumération. La méthode `next` est telle que chaque nouvel appel renvoie un nouvel élément de l'ensemble à énumérer. Elle fait donc évoluer l'état de l'itérateur. Ces méthodes, utilisées ensembles, permettraient d'écrire le code suivant, qui énumère tous les éléments d'un ensemble `set` d'entiers.

```
Iterator<Integer> it = set.iterator();
while (it.hasNext()) {
  Integer k = it.next();
  ...
}
```

La boucle *for each* est précisément une syntaxe simplifiée pour les lignes précédentes.

On propose d'énumérer les éléments d'un `TabSet` dans l'ordre donné par le tableau `tab` sous-jacent. Il suffit pour cela de se donner un indice `i` désignant le prochain élément à renvoyer, et d'incrémenter cet indice à chaque appel à `next`.

1. Compléter la ligne **`public class`** `TabSet` **`implements`** `Set` en

   ```
   public class TabSet implements Set, Iterable<Integer> {
   ```

   Cela permet de déclarer que la classe `TabSet` concrétise simultanément les deux interfaces `Set` et `Iterable<Integer>`. Ajouter à la classe `TabSet` la définition suivante,

   ```
   public Iterator<Integer> iterator() {
       return new TabSetIterator(tab, size);
   }
   ```

2. Définir la classe `TabSetIterator` concrétisant l'interface `Iterator<Integer>`. Il faut donc définir, en plus du constructeur, les méthodes `next` et `hasNext`.
3. Ajouter dans la fonction principale un test énumérant tous les éléments d'un `TabSet` pour les afficher, à l'aide d'une boucle *for each*.

On propose d'énumérer les éléments d'un `BitSet` dans l'ordre croissant.

4. Créer une classe `BitSetIterator`, qui concrétise l'interface `Iterator<Integer>`. À vous de préciser quels éléments devra contenir cette classe pour permettre l'itération.
5. Compléter la classe `BitSet` afin qu'elle concrétise l'interface `Iterable<Integer>`.