Ensemble ordonné
Corrigé

Exercice 1 : Définition d'un ensemble

L'objectif de cet exercice est de définir des ensembles d'entiers qui pourront être utilisés dans différents contextes.

Nous prendrons comme exemple d'utilisation de l'ensemble, le calcul des nombres premiers par une variante de l'algorithme dit du « crible d'Ératosthène ». Ce n'est cependant qu'un exemple.

Pour trouver tous les nombres premiers de 2 à MAX, le principe du crible d'Ératosthène est le suivant :

1. Construire l'ensemble contenant tous les entiers de 2 à MAX ;
2. Extraire et afficher le plus petit élément de l'ensemble car c'est un nombre premier ;
3. Enlever de l'ensemble tous les multiples de ce nombre premier ;
4. Continuer en 2 jusqu'à ce que l'ensemble soit vide.

1.1. Spécifier en UML puis écrire en Java `Ensemble` qui décrit un ensemble d'entiers.

Pour simplifier, on définira seulement les opérations sur les ensembles qui sont utiles pour le crible d'Ératosthène, c'est-à-dire ajouter un élément, supprimer un élément, savoir si un élément est présent ou non, savoir si l'ensemble est vide ou non, connaître la cardinalité de l'ensemble, donner le plus petit élément de l'ensemble. On ne considérera donc pas les opérations sur les ensembles telles que l'union, l'intersection, l'inclusion, etc.

Solution : Spécifier en UML, c'est faire un diagramme d'analyse sur lequel on ne s'intéresse qu'aux méthodes publiques classées en requêtes et commandes. C'est la vue utilisateur. Plus précisément, on doit :

- identifier ces requêtes et commandes
- donner leur signature : nom et type des paramètre, type de retour pour les requêtes. UML permet aussi d'indiquer une direction (**in**, **out** et **in out**)
- donner la documentation, une description en langage naturelle de l'objectif de la méthode, de ses paramètres, de ses conditions d'utilisation, de ses effets, etc.
- une documentation en langage naturelle est informelle, donc souvent ambiguë. On peut donc la formaliser en utilisant la programmation par contrat. On identifie alors :
  - des invariants qui portent sur les requêtes, l'état visible de l'objet. Les invariants doivent toujours être vrais pour tout objet du type considéré, ici Ensemble.
  - des préconditions et postconditions qui formalisent les conditions d'utilisation et les effets d'une méthode.

Voici le diagramme d'analyse sans faire apparaître sa documentation et sa formalisation.

Ensemble
requêtes
cardinal : int estVide : boolean contient(in x : int) : boolean min : int
commandes
ajouter(in x : int) supprimer(in x : int)

Quelques remarques :

— Nous devrions utiliser le terme EnsembleOrdonné et non Ensemble puisqu'il fournit aussi le min, opération que l'on n'a pas sur un ensemble.

— Une commande correspond généralement à un verbe à l'infinitif : on demande à un objet de faire quelque chose.

— Le nom d'une requête caractérise l'information fournie par la requête, souvent un nom.
Dans le cas d'une requête booléenne, on utilise souvent la forme `estXxx` comme par exemple `estVide` pour savoir si un ensemble est vide ou non.
Nous avons choisi `cardinal()` et `min()` plutôt que `getCardinal()` et `getMin()`. Ceci ne semble pas suivre les convention [1] Java qui consistent à définir des accesseurs et modifieurs en utilisant les prefixes get et set. Par exemple, getNom et et setNom pour accéder à la valeur du nom ou changer cette valeur. Si c'est un moyen simple et automatique de nommer les accesseurs et modifieurs, ce n'est pas forcément très lisible et pas systématique, même dans les bibliothèques Java. Par exemple la taille d'une liste en Java se note `size()` et non `getSize()`.
Enfin, quand on nomme une requête, il faut penser à la manière dont elle sera utilisée. Par exemple, la requête qui dit si un élément est présent pourrait s'appeler `estPresent` (ou `appartient`) mais est-ce que c'est lisible d'écrire : `monEnemble.estPresent(3)` (ou `monEnsemble.appartient(3)`) ? Non, on lit dans le mauvais sens. Un meilleur nom pour cette requête est donc `contient` (`monEnsemble.contient(3)`).

Nous n'allons pas écrire la documentation, elle apparaîtra dans le code Java. Nous allons toutefois la formaliser en utilisant JML.
Commençons par les invariants.

```
public invariant estVide() <==> cardinal() == 0;
public invariant 0 <= cardinal();

public invariant ! estVide() ==> contient(min());
public invariant ! estVide() ==>
      (\forall int x; contient(x); x >= min());
```

1. C'est une convention qui vient des Java Beans, composants réutilisables que l'on veut pouvoir assembler dynamiquement même sans connaître la classe de ces composants. On s'appuie sur l'introspection et des conventions sur les noms des méthodes. Ainsi une propriété d'un *bean* (qui pourrait être stockée dans un attribut) est reconnu grâce aux méthodes getXxx (on peut donc accèder à la valeur de la propriété xxx) et setXxx (on peut donc modifier la valeur de la propriété xxx).

La dernière peut être réécrite en utilisant l'opérateur `\min` :

```
public invariant ! estVide() ==>
      min() == (\min int x; contient(x); x);
```

Notons la garde (`! estVide()`) utilisée pour les deux derniers invariants. Elle est nécessaire car le minimum n'est pas défini si l'ensemble est vide.
Intéressons nous maintenant aux préconditions et postconditions des méthodes. Commençons par les requêtes.

1. estVide : pas de précondition (on peut toujours demander à tout ensemble s'il est vide ou non) pas de postcondition (les deux valeurs retournées vrai et faux sont possibles). estVide ne modifie pas l'ensemble, mais c'est ce que signifie une requête. Cette propriété n'existe pas en Java. On peut le dire en JML en la qualifiant de **`pure`**.
2. Contient : Pas de précondition, pas de postcondition (les deux résultats vrai et faux sont possible).

   Il est parfois proposé de définir une précondition : `! estVide()`. Si on ajoute cette précondition, on complique l'utilisation de cette méthode. En effet, l'utilisateur devra vérifier que l'ensemble n'est pas vide avant d'utiliser `contient`.

   En fait, si l'ensemble est vide, on sait que le résultat sera faux. Ceci pourrait être exprimé par une postcondition : `estVide() ==> ! \result`
3. cardinal() : On pourrait définir une postcondition qui exprime qu'elle doit être positive ou nulle. Cette propriété a été déjà capturée par un invariant. Il est donc inutile de la rappeler en postcondition.

   Si on instrumente les contrats, utiliser un invariant signifie que la propriété sera vérifiée avant et après chaque exécution d'une méthode publique de l'objet. Une postcondition n'est vérifiée qu'après l'exécution de la méthode portant la postcondition.
4. min() : il n'a de sens que si l'ensemble est non vide. D'où une précondition. Les postconditions ont déjà été exprimées au moyen d'invariants.

On a souvent des réponses différentes concernant les préconditions et postconditions des deux commandes `ajouter` et `supprimer`. Ceci démontre qu'il est donc essentiel d'exprimer les contrats pour être sûr de la sémantique de chaque méthode.

Certains disent qu'il faut que l'élément ne soit pas présent pour pouvoir l'ajouter, d'autre qu'on peut toujours ajouter, etc. On retrouve les mêmes variantes pour supprimer : l'élément doit être présent ou pas en précondition, voire l'ensemble ne doit pas être vide. Pour la postcondition, tout le monde est d'accord que `ajouter` ajoute l'élément et que `supprimer` le supprime. On peut aussi donner les conditions sur l'évolution de le cardinal qui seront plus ou moins faciles à écrire suivant la précondition choisie. Enfin, il faudrait aussi exprimer le fait que tous les autres éléments de l'ensemble restent inchangés.

Les postconditions ci-dessus sont dites *postconditions naturelles* car elles expriment l'objectif principal de la méthode. Si on utilise les contrats pour instrumenter le code, elles sont suffisantes pour détecter les erreurs classiques de programmation. Si en revanche on veut pouvoir raisonner sur le programme indépendamment de toute exécution particulière, il est nécessaire de les compléter. Par exemple, pour ajouter, il faudrait dire que :

— tous les éléments qui étaient avant dans l'ensemble sont toujours dans l'ensemble
— que le cardinal après est le cardinal avant si l'élément ajouté était déjà présent dans l'ensemble ou le cardinal avant plus 1 si l'élément n'était pas présent.

Au lieu de la deuxième propriété, on pourrait dire que tous les éléments de l'ensemble après ajout sont soit l'élément ajouté, soit un élément qui était présent dans l'ensemble avant.

Si des contrats différents sont proposés pour `ajouter` et `supprimer` c'est que les deux méthodes ne sont pas bien comprises. Il est donc essentiel de formaliser la description pour lever les ambigüités et être sûr que ceux qui écriront le crible (question 1.2) et ceux qui réaliseront l'ensemble (question 1.3) pourront mettre leur code ensemble pour obtenir un programme fonctionnel.

Si on ne lève pas les ambigüités dans la phase de spécification, c'est au moment de programmer le crible ou l'implantation de l'ensemble que le choix sera fait, en général en retenant le choix qui permet d'écrire plus simplement le code. Si tout le monde se simplifie la vie, il y a de bonnes chances que l'application complète ne fonctionne pas.

On peut se demander qu'elles sont les « bonnes » préconditions. La réponse est qu'il n'y en a pas. Le tout est de faire un choix et de le dire explicitement grâce aux préconditions et postconditions.

On peut toutefois noter que choisir une précondition plus forte facilitera la vie de celui qui devra implanter la méthode et compliquera celle de celui qui devra l'utiliser. C'est ce qu'on a vue avec `contient` ci-dessus.

Remarque : Il serait plus logique de faciliter la vie des utilisateurs que de ceux qui implantent la méthode car ils sont plus nombreux ! Aussi, en général, on préfère des préconditions plus faibles.

Pour la suite du TD, on fera les choix suivants :

```
/*@
   requires ! contient(x)
   ensures contient(x)  // élémnet ajouté
   ensures cardinal() == \Old(cardinal()) + 1
@*/
void ajouter(int x);

/*@
   requires contient(x)
   ensures ! contient(x) // élément supprimé
   ensures cardinal() == \Old(cardinal()) - 1
@*/
void ajouter(int x);
```

La deuxième partie de la question demande de traduire en Java cette notion d'ensemble. Le piège est de penser *classe* et donc de se poser la question des attributs ! Pourquoi faire ce choix maintenant ? En particulier, la question 1.3 montre bien que ce choix n'est pas encore à faire !

La traduction en Java d'une spécification, c'est une interface. On pourrait écrire systématiquement une interface avant d'écrire les classes qui la réalisent, même si ce n'est pas ce qui se fait généralement.

```
1 /** Définition d'un ensemble d'entier. */
2 public interface Ensemble {
```

```
3       //@ public invariant estVide() <==> cardinal() == 0;
4       //@ public invariant 0 <= cardinal();
5
6       // Caractérisation du min (quand il existe !)
7       // -------------------------------------------
8       /*@ public invariant ! estVide() ==>
9               contient(min());              // un élément de l'ensemble
10          public invariant ! estVide() ==>
11              min() == (\min int x; contient(x); x);    // le plus petit
12      @*/
13
14      /** Obtenir le nombre d'éléments dans l'ensemble.
15       * @return nombre d'éléments dans l'ensemble.  */
16      /*@ pure helper @*/ int cardinal();
17
18      /** Savoir si l'ensemble est vide.
19       * @return Est-ce que l'ensemble est vide ? */
20      /*@ pure helper @*/ boolean estVide();
21
22      /** Savoir si un élément est présent dans l'ensemble.
23       * @param x l'élément cherché
24       * @return x est dans l'ensemble */
25      /*@ pure helper @*/ boolean contient(int x);
26
27      /** Ajouter un élément dans l'ensemble.
28       * @param x l'élément à ajouter */
29      //@ requires ! contient(x);     // l'élément n'y est pas déjà
30      //@ ensures contient(x);        // élément ajouté
31      //@ ensures ! \old(contient(x)) ==> cardinal() == \old(cardinal()) + 1;    // car
32      //@ ensures \old(contient(x)) ==> cardinal() == \old(cardinal());        // ou ca
33      void ajouter(int x);
34
35      /** Enlever un élément de l'ensemble.
36        * @param x l'élément à supprimer */
37      //@ requires contient(x);       // l'élément est l'ensemble
38      //@ ensures ! contient(x);      // élément supprimé
39      //@ ensures \old(contient(x)) ==> cardinal() == \old(cardinal()) - 1;    // cardi
40      //@ ensures ! \old(contient(x)) ==> cardinal() == \old(cardinal());       // ou
41      void supprimer(int x);
42
43      /**  Obtenir le minimum de l'ensemble.
44       * @return le plus petit élément de l'ensemble  */
45      //@ requires !estVide();
46      /*@ pure helper @*/ int min();
47
48  }
```

1.2. Écrire l'algorithme qui calcule les nombres premiers en utilisant le principe du crible d'Ératosthène (et l'ensemble défini, bien sûr !) ainsi qu'un programme qui l'utilise.

Solution :

Dans une approche objet, tout est (devrait être) objet !
On a donc un objet pour le crible, de la classe Crible.
Un crible sert à.... cribler (afficher les nombres premiers).

Quels sont les paramètres ? Normalement, seulement max : on veut afficher les nombres premiers de 2 à max. Mais, pour réaliser ce traitement, on a besoin d'un ensemble. Ce pourrait être une variable locale mais pour l'instant on ne sait pas créer un ensemble (pas encore de réalisation e l'ensemble). On a alors deux possibilités :

1. le recevoir en paramètre de la méthode `cribler`. Mais ce n'est pas très logique, l'utilisateur de cribler ne devrait fournir que `max` comme identifié ci-dessus.
2. en attribut de la classe. Il sera initialisé grâce au constructeur. C'est ici que l'on va définir l'ensemble qui sera utilisé dans la méthode cribler.

   Remarque : L'interface Ensemble spécifie les opérations attendus sur l'objet qui servira à stocker les entiers. Dans une approche à composant, on l'appellera « interface requise ». Pour que le « composant » Crible fonctionne, il faudra l'assembler avec un « composant » qui réalise son interface requise `Ensemble`.

Remarque : On constate sur cet exemple que l'on peut utiliser la méthode des raffinages (voir les commentaires dans le code).

Il faut bien sûr s'assurer que l'on a le droit d'écrire chaque ligne de code. On doit donc vérifier ses conditions d'applications, en particulier si c'est un appel à une méthode, les préconditions de cette méthodes.

Que penser de la la classe Crible suivante ?

```
1  /** Le crible d'Ératosthène.  Il calcule les nombres premiers suivant
2   * l'algorithme dit du crible d'Ératosthène.  Les entiers sont
3   * stockés dans un ensemble.
4   * @author Xavier Crégut &lt;prenom.nom@enseeiht.fr&gt;
5   */
6  public class Crible {
7
8      //@ private invariant nombres.estVide();
9
10     /** L'ensemble utilisé par le crible */
11     private Ensemble nombres;
12
13     /** Fournir l'ensemble ens à utiliser
14      * @param ens l'ensemble à utiliser
15      */
16     //@ requires ens.estVide();
17     public Crible(Ensemble ens) {
18         this.nombres = ens;
19     }
20
21     /** Cribler les entiers de 2 à max par l'algorithme d'Ératosthène.
22      * Les nombres premiers sont affichés à l'écran.
23      * L'ensemble nombres est utilisé pour conserver les entiers.
24      * @param max borne supérieure de l'intervalle
```

```
25          */
26         //@ requires max >= 2;
27         public void cribler(int max) {
28             // construire l'ensemble des nombres entiers de 1 à max
29             for(int entier = 2; entier <= max; entier++) {
30                 this.nombres.ajouter(entier);
31             }
32 
33             // afficher les nombres premiers
34             System.out.print("Nombres premiers : ");
35             int nb = 0;    // nombre de nombres premiers affichés
36             while (!this.nombres.estVide()) {
37                 // déterminer le nombre premier suivant
38                 int premier = this.nombres.min();
39 
40                 // afficher le nombre premier
41                 if (nb > 0) {
42                     System.out.print(", ");
43                 }
44                 System.out.print(premier);
45                 nb++;
46                 System.out.flush();    // forcer l'affichage
47 
48                 // supprimer les multiples de premier
49                 for(int multiple = premier; multiple <= max; multiple += premier) {
50                     this.nombres.supprimer(multiple);
51                 }
52             }
53             System.out.println();
54         }
55 
56     }
```

Est-ce que le compilateur va accepter cette classe `Crible` ? Pourquoi ?

Le compilateur l'accepte et produira le fichier `Crible.class` correspondant car il a pu vérifier que l'appel de chaque méthode de l'ensemble était conforme à la signature donnée dans l'interface `Ensemble`. C'est la **laison statique** !

Quel sera le code exécuté lors des appels aux méthodes de l'ensemble : `ajouter`, `min` ou `supprimer` ?

On ne le sait pas... et le compilateur non plus ! On n'a pas encore écrit l'implantation de ces méthodes. Ni le compilateur, ni nous, ne savons où on pourrait le trouver !

Ce n'est pas un problème ! À l'exécution, l'ensemble concret sera connu (celui passé en paramètre du constructeur de `Crible`) et ce sont ses méthodes qui seront appelées. C'est la **liaison statique**.

Le code de `Crible.cribler` ci-dessus est le code généralement écrit par les étudiants. Est-il correct ? Pourquoi ?

Il est incorrect car on n'a pas respecté la spécification de l'ensemble. Plus précisément, quand on supprime les multiples on n'est pas sûr que l'entier est encore présent dans l'ensemble. Par exemple, lors de la suppression des multiples de 3, on va supprimer 6 qui a déjà été supprimé comme multiple de 2 !

Il faut donc rajouter un condition avant la suppression.

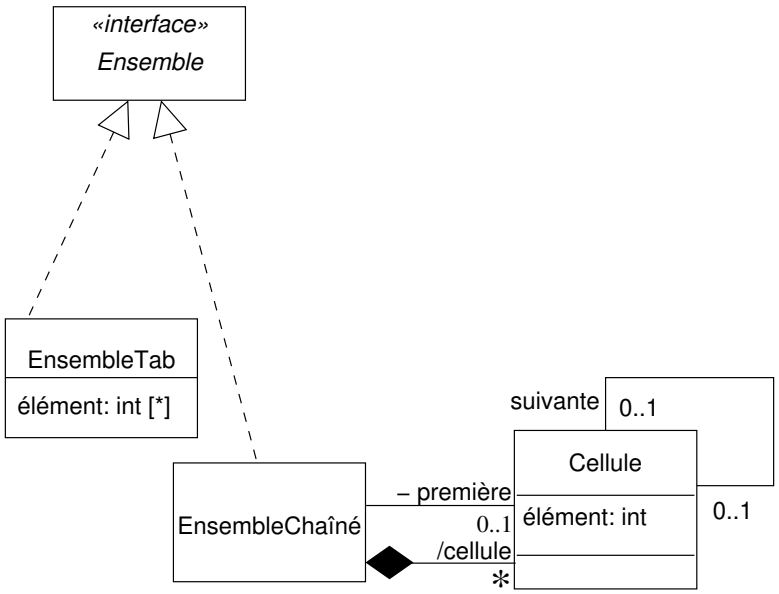
```
if (nombres.contient(multiple)) {
   nombres.supprimer(nombre);
}
```

On constate bien qu'on a rendu la vie moins facile pour les utilisateurs en définissant une précondition sur `supprimer` ! On constate aussi que le programmeur est enclin à se faciliter la vie en considérant que `supprimer` ne devrait pas avoir de précondition !

1.3. On hésite sur la manière d'implanter ces ensembles. On envisage d'utiliser un tableau pour stocker les éléments ou de les chaîner entre eux.

1.3.1. Dessiner le diagramme de classe qui fait apparaître ces deux implantations des ensembles.

Solution :



Notons la relation de composition entre `EnsembleChaîne` et `Cellule`. Le rôle est préfixé « / » pour indiquer que cette relation peut être déduite des autres (en l'occurrence de « première » et « suivante »). La relation de composition montre que c'est la liste qui est responsable de la durée de vie des cellules. On n'a pas mis une relation de composition pour « suivante » car une cellule n'est pas responsable de la durée de vie de sa suivante.

Pour `EnsembleTab`, notez la définition de l'attribut « élément » avec une multiplicité « * » notée entre crochets. On a dit ainsi qu'il pouvait y avoir plusieurs éléments pour un `EnsembleTab`. C'est une notation équivalente à celle d'une relation avec le rôle au singulier et une multiplicité.

1.3.2. Expliquer comment on dit que la classe `EnsembleTab` est une réalisation de Ensemble.
Solution : On utiliser le mot-clé **`implements`** :

```
public class EnsembleTab implements Ensemble {
    ...
}
```

1.3.3. Expliquer ce que fera la méthode qui ajoute un élément dans un ensemble pour les deux implantations de l'ensemble. Le code n'est pas demandé.
Solution : Notre précondition spécifie que l'élément ajouté ne doit pas être déjà présent dans l'ensemble. Il suffit donc de l'ajouter à la fin du tableau ou en première cellule de l'ensemble chaîné.

Si nous avions autorisé à ajouter dans l'ensemble un élément déjà présent, alors nous aurions dû ajouter une condition pour vérifier que l'élément n'est pas présent dans l'ensemble avant de l'ajouter.

Dans le cas de la version avec tableau, se pose la question de savoir quoi faire si le tableau est plein. Ici encore, c'est la précondition qui nous permet de savoir quoi faire. Si nous avons mis une précondition imposant que le cardinal soit strictement inférieure à la capacité du tableau (et donc si cette notion de capacité a été identifiée dans l'interface Ensemble), ce cas ne peut pas se produire (ou l'appelant a commis une erreur en ne s'assurant pas que la précondition est vraie). Ici la précondition ne mentionne rien concernant une éventuelle capacité, c'est qu'on doit toujours pouvoir ajouter un nouvel élément. Dans ce cas, il nous faut allouer un tableau plus grand et copier les éléments de l'ancien tableau dans le nouveau. On peut alors ajouter le nouvel élément.

1.3.4. Écrire le code de la méthode qui donne le plus petit élément d'un ensemble pour les deux implantations de l'ensemble.
Solution : Voici le code de la méthode min() de EnsembleTab :

```
    @Override public int min() {
        int resultat = elements[0];
        for (int i = 1; i < nb ; i++) {
            if (elements[i] < resultat) {
                resultat = elements[i];
            }
        }
        return resultat;
    }
```

Voici le code de la méthode min() de EnsembleChainee :

```
    @Override public int min() {
        int min = this.premiere.element;
        Cellule curseur = this.premiere.suivante;
        while (curseur != null) {
            if (curseur.element < min) {
                min = curseur.element;
            }
```

```
8                curseur = curseur.suivante;
9            }
10           return min;
11       }
```

1.3.5. Est-ce que les implantations envisagées sont efficaces ?
Solution : Non, elles ne sont pas efficaces. Par exemple, pour savoir si un élément est présent ou non dans un ensemble, il faut, dans le pire des cas, accéder à tous les éléments (cas où l'élément n'est pas présent). De la même manière la suppression ne sera pas efficace car elle nécessite un parcours de tous les éléments (si l'élément n'est pas présent) ou presque (s'il est en dernière position).

Quelles implantations seraient plus efficaces ?
Solution : On pourrait prendre un tableau caractéristique : l'indice est l'élément, la valeur est sa présence ou non. Est-ce réaliste sur des entiers quelconques ?

On pourrait prendre une table de hachage.

Si on a une relation d'ordre sur les éléments (ce qui est le cas ici puisque l'on s'intéresse au minimum des éléments de l'ensemble) on pourrait s'appuyer sur un arbre, par exemple un arbre binaire de recherche.

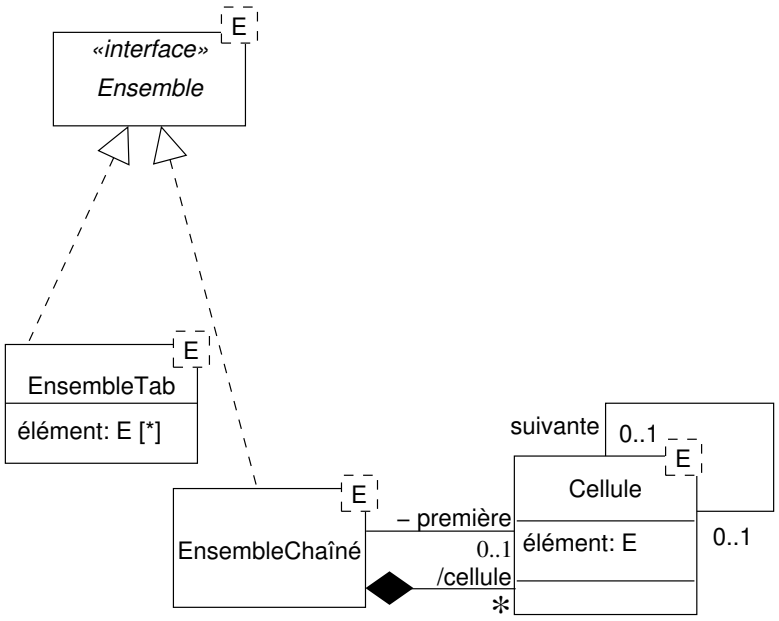
Exercice 2 : Généraliser les ensembles du crible d'Ératosthène
Dans l'exercice 1, nous avons eu le souci de développer une classe Ensemble réutilisable. Cependant, nous nous sommes limités à des ensembles d'entiers. Il serait intéressant de disposer d'ensemble de points, de nombres réels, etc.

Proposer une solution pour atteindre cet objectif.
Solution : On remarque qu'il suffit de remplacer la plupart des int par le nouveau type pour avoir un tableau pour un nouveau type. Pour éviter d'avoir à faire du copier/coler, puis du remplacement de texte, on a deux possibilités :

1. Utiliser le polymorphisme paramètrique (la généricité) ;
2. Utiliser le polymorphisme de sous-typage.

La généricité a pour avantage de préserver le contrôle de type à l'insertion et à l'extraction (pas de transtypage à faire). La généricité a été ajouté en Java dans sa version 5, donc il y a bien longtemps. Il est donc fortement conseillé de l'utiliser pour bénéficier du contrôle de type réalisé par le compilateur.

Polymorphisme paramétrique (ou généricité). Les interface et les classes sont alors paramétrées par le type des éléments qui peuvent être mis dans l'ensemble. En UML, on note les paramètres de généricité dans un rectangle en traits interrompus, en haut à droite de la classe ou interface.



Comme nous souhaitons obtenir le minimum d'un ensemble, il nous faut donc être capable de comparer entre eux les éléments de l'ensemble. E ne peut donc pas être n'importe quel type. Il faut qu'il soit équipé d'une relation d'ordre (de comparaison). Le moyen de l'imposer en Java est de dire que E doit être un sous-type du type qui définit cette opération. En Java, l'interface Comparable permet de comparer un élément (**`this`**) avec un autre dont le type est le paramètre de généricité de `Comparable`. Cette méthode s'appelle `compareTo`. Pour dire que E est un sous-type de `Comparable`, on n'utilise pas le mot-clé **`implements`** mais **`extends`** (utilisé pour l'héritage et qui induit aussi une relation de sous-typage).

Remarquons que le paramètre de généricité E pourra être instancié avec `Integer` ou `Double` qui réalisent `Comparable` et définissent donc sa méthode `compareTo`. En revanche, notre classe `Point` ne pourra pas être utilisée sauf à lui faire réaliser `Comparable` et définir sa méthode `compareTo`.

```
1  /** Définition d'un ensemble d'entier. */
2  public interface Ensemble<E extends Comparable<E>> {
3      //@ public invariant estVide() <==> cardinal() == 0;
4      //@ public invariant 0 <= cardinal();
5
6      // Caractérisation du min (quand il existe !)
7      // -------------------------------------------
8      /*@ public invariant ! estVide() ==>
9              contient(min());               // un élément de l'ensemble
10         public invariant ! estVide() ==>
11             min() == (\min int x; contient(x); x);    // le plus petit
12     @*/
13
14     /** Obtenir le nombre d'éléments dans l'ensemble.
15      * @return nombre d'éléments dans l'ensemble.  */
16     /*@ pure helper @*/ int cardinal();
```

```
17
18      /** Savoir si l'ensemble est vide.
19       * @return Est-ce que l'ensemble est vide ? */
20      /*@ pure helper @*/ boolean estVide();
21
22      /** Savoir si un élément est présent dans l'ensemble.
23       * @param x l'élément cherché
24       * @return x est dans l'ensemble */
25      /*@ pure helper @*/ boolean contient(E x);
26
27      /** Ajouter un élément dans l'ensemble.
28       * @param x l'élément à ajouter */
29      //@ requires ! contient(x);     // l'élément n'y est pas déjà
30      //@ ensures contient(x);        // élément ajouté
31      //@ ensures ! \old(contient(x)) ==> cardinal() == \old(cardinal()) + 1;    // car
32      //@ ensures \old(contient(x)) ==> cardinal() == \old(cardinal());        // ou ca
33      void ajouter(E x);
34
35      /** Enlever un élément de l'ensemble.
36        * @param x l'élément à supprimer */
37      //@ requires contient(x);       // l'élément est l'ensemble
38      //@ ensures ! contient(x);      // élément supprimé
39      //@ ensures \old(contient(x)) ==> cardinal() == \old(cardinal()) - 1;    // cardi
40      //@ ensures ! \old(contient(x)) ==> cardinal() == \old(cardinal());        // ou
41      void supprimer(E x);
42
43      /**  Obtenir le minimum de l'ensemble.
44       * @return le plus petit élément de l'ensemble  */
45      //@ requires !estVide();
46      /*@ pure helper @*/ E min();
47
48  }
```

Pour la réalisation `EnsembleTab`, on écrira :

```
1  public class EnsembleTab<E extends Comparable<E>> implements Ensemble<E> {
2      ...
3  }
```

```
1  /** Le crible d'Ératosthène.  Il calcule les nombres premiers suivant
2   * l'algorithme dit du crible d'Ératosthène.  Les entiers sont
3   * stockés dans un ensemble.
4   * @author Xavier Crégut &lt;prenom.nom@enseeiht.fr&gt;
5   */
6  public class Crible {
7
8      //@ private invariant nombres.estVide();
9
10     /** L'ensemble utilisé par le crible */
11     private Ensemble<Integer> nombres;
```

```java

    /** Fournir l'ensemble ens à utiliser
     * @param ens l'ensemble à utiliser
     */
    //@ requires ens.estVide();
    public Crible(Ensemble<Integer> ens) {
        this.nombres = ens;
    }

    /** Cribler les entiers de 2 à max par l'algorithme d'Ératosthène.
     * Les nombres premiers sont affichés à l'écran.
     * L'ensemble nombres est utilisé pour conserver les entiers.
     * @param max borne supérieure de l'intervalle
     */
    //@ requires max >= 2;
    public void cribler(int max) {
        // construire l'ensemble des nombres entiers de 1 à max
        for(int entier = 2; entier <= max; entier++) {
            this.nombres.ajouter(entier);
        }

        // afficher les nombres premiers
        System.out.print("Nombres premiers : ");
        int nb = 0;    // nombre de nombres premiers affichés
        while (!this.nombres.estVide()) {
            // déterminer le nombre premier suivant
            int premier = this.nombres.min();

            // afficher le nombre premier
            if (nb > 0) {
                System.out.print(", ");
            }
            System.out.print(premier);
            nb++;
            System.out.flush();    // forcer l'affichage

            // supprimer les multiples de premier
            for(int multiple = premier; multiple <= max; multiple += premier) {
                if (this.nombres.contient(multiple)) {
                    this.nombres.supprimer(multiple);
                }
            }
        }
        System.out.println();
    }

}
```

Remarque : On ne peut pas avoir un ensemble dont les éléments sont des types primitifs. Il faut dans ce cas passer par les classes enveloppes. Ainsi on aura un ensemble d'`Integer` et non de

`int` !

Polymorphisme de sous-typage. Il s'agit de remplacer le type int (ou E de la version générique) par un type (une classe ou interface) suffisamment général pour pouvoir y mettre tous les éléments souhaités, par exemple `Object`. Dans le cas présent, c'est `Comparable` qu'il faut prendre puisque nous devons pouvoir comparer les éléments pour calculer le minimum de l'ensemble.

L'inconvénient de cette approche, c'est que l'on perd le contrôle de type. Dans un ensemble de comparable, on pourra mettre des entiers, des réels ou des chaînes de caractères qui sont tous des sous-types de Comparable.

Quand on demandera le minimum de l'ensemble, on n'obtiendra un `Comparable` qu'il faudra transtyper pour retrouver son vrai type et pouvoir en utiliser toutes les opérations. C'est à ce moment que l'on aura une erreur si on a ajouté dans l'ensemble un élément du mauvais type (par exemple une chaîne de caractères dans un ensemble d'entiers).

Objectifs :

- Spécification
- Interface
- Réalisation
- Sous-typage, Principe de substitution, Liaison dynamique
- Généricité
- Généricité contrainte