Fiche 3 : définition de classes, bis

IPO – Introduction à la programmation objet

Thibaut Balabonski @ Université Paris-Saclay
`https://www.lri.fr/~blsk/IPO/`

V1, automne 2022

Environnement de développement Un *environnement de développement* (IDE) regroupe un ensemble d'outils utiles pour écrire, compiler, tester, débugger (et encore d'autres choses...) un projet logiciel. Pour ce cours, on suggère *IntelliJ*. Depuis les salles du bâtiment 336, vous pouvez le lancer en exécutant la commande suivante dans un terminal.

```
idea.sh &
```

Pour commencer à programmer :

1. cliquer sur « New Project »,
2. donner un nom au projet, puis valider avec les valeurs par défaut,
3. créer (au moins) un fichier source avec un clic droit sur le répertoire `src`, (New > Java Class),
4. c'est prêt !

L'outil réagit en direct à tout ce qui est écrit. En particulier :

— lorsque l'on commence à écrire un mot, il montre les différentes manières possibles de le compléter, en se basant sur les noms de classes qui existent, ou sur les noms de méthodes et d'attributs qui peuvent être utilisés à cet endroit (les flèches permettent de sélectionner l'une des possibilités, et la touche entrée de valider),
— il signale en rouge les erreurs de compilation,
— il peut émettre des suggestions sur la manière de corriger certaines erreurs ou d'améliorer certaines lignes de code (à utiliser avec précaution !)

Une fois que votre programme possède une méthode `main`, vous pouvez cliquer sur le triangle vert pour le compiler et l'exécuter. S'affiche alors en bas de la fenêtre un terminal dans lequel on pourra observer le bon déroulement du programme, ou les erreurs survenant à l'exécution.

Exemple de code : associations On définit une classe pour représenter des paires « clé, valeur », formées par une chaîne de caractères (la *clé*) et une valeur entière.

```
public class Association {
  private String key;
  private int value;
```

Rappel : un constructeur doit initialiser les attributs. Il possède donc, en gros, une ligne par attribut.

```
  public Association(String k, int v) {
    this.key = k;
    this.value = v;
  }
```

Les attributs sont privés, ce qui garantit qu'ils ne seront pas modifiés par un utilisateur maladroit. Pour donner accès à un attribut en *lecture*, on peut créer un *getter*, c'est-à-dire une méthode publique dont le seul objectif est de transmettre la valeur d'un attribut.

```
  public String getKey () {
    return this.key;
  }
```

Note : cette méthode ne permet que de consulter l'attribut, pas de le modifier. Pour la modification, on introduit *parfois* une autre méthode appelée *setter*. Par défaut, ne pas le faire !

```
  public void setValue(int v) {
    this.value = v;
  }
```

Dans le cas où l'on définit un *setter*, attention : une telle méthode doit s'assurer que les modifications préservent les éventuels invariants de la classe.

Pour compléter notre classe, on peut donner une fonction définissant l'affichage d'une association.

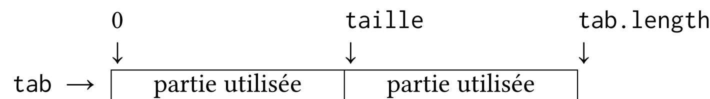
```
  public String toString () {
    return this.key + " -> " + this.value;
  }
}
```

Exemple de code : tableau redimensionnable On définit une classe pour représenter un tableau dans lequel il est possible d'ajouter dynamiquement des éléments. Notez que ceci n'est pas permis par les tableaux primitifs de Java, dont la taille est définitivement fixée lors de leur création.

On donne deux attributs à un tableau redimensionnable : un tableau primitif Java destiné à contenir les éléments, et un entier indiquant quelle partie du tableau concret est effectivement utilisée.

```
public class TableauRedimensionnable {
  private Association[] tab;
  private int taille;
```

On peut se figurer la situation ainsi :



À la création, le tableau sous-jacent `tab` a une longueur par défaut, définie arbitrairement, et la taille du tableau redimensionnable est fixée à zéro (il n'y a pas encore d'éléments).

```
public TableauRedimensionnable() {
  this.taille = 0;
  this.tab = new Association[32];
```

Pour accéder à un élément du tableau redimensionnable identifié par un indice `i`, on vérifie que l'indice désigne bien une position de la partie utilisée du tableau sous-jacent. Dans le cas contraire, la ligne commençant par **`throw`** déclenche la même erreur que celle produite par Java lors de l'accès à un indice invalide d'un tableau primitif.

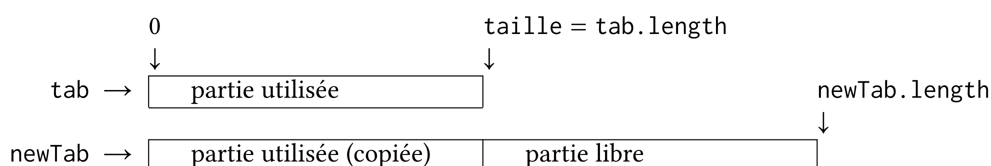
```
  public Association get(int i) {
    if (i < this.taille) {
      return this.tab[i];
    } else {
      throw new ArrayIndexOutOfBoundsException(i);
    }
  }

  public void set(int i, Association a) {
    if (i < this.taille) {
      this.tab[i] = a;
    } else {
      throw new ArrayIndexOutOfBoundsException(i);
    }
  }
```

Note : en cas d'indice négatif, c'est l'accès au tableau sous-jacent lui-même qui déclenche l'erreur.

Pour l'ajout d'un nouvel élément, on a deux cas à traiter. S'il reste effectivement de la place libre dans le tableau tab sous-jacent, il suffit d'utiliser la prochaine case, et d'incrémenter la taille. S'il ne reste plus de place libre en revanche, il faut au préalable étendre le tableau sous-jacent, ce qu'on délègue à une méthode extend détaillée ci-dessous.

```
  public void add(Association a) {
    if (this.taille >= this.tab.length) {
      this.extend();
    }
    this.tab[this.taille++] = a;
  }
```

Comme il n'est pas possible de modifier la taille d'un tableau primitif de Java, on va créer un nouveau tableau plus grand, qui va remplacer le tableau d'origine. Pour ne rien perdre, on copie intégralement le contenu du tableau d'origine dans le début du nouveau tableau.



```
    private void extend () {
      Association[] newTab = new Association[2*this.taille];
      for (int i=0; i < this.taille; i++)
        newTab[i] = this.tab[i];
      this.tab = newTab;
    }
```

Note : ce code repose effectivement sur le fait que **`this`**`.taille` est la taille réelle `tab.length` du tableau sous-jacent.

Du point de vue de **`this`**, c'est-à-dire du tableau redimensionnable lui-même, la méthode `extend` ne fait rien d'autre que modifier un attribut.

this (avant) →	TableauRedimensionnable	taille	tab
this (après) →	TableauRedimensionnable	taille	newTab

Pour finir, on ajoute une méthode qui cherche dans un tel tableau redimensionnable une association portant une clé `k` donnée en paramètre. La méthode renvoie l'indice de l'association trouvée, ou `-1` à défaut.

```
    public int findKey(String k) {
      for (int i=0; i < this.taille; i++) {
        String kk = this.tab[i].getKey();
        if (kk.equals(k)) {
          return i;
        }
      }
      return -1;
    }
  }
```

Note : cette méthode fait appel à la méthode `getKey()` de la classe `Association`. En revanche, dans tout ce qui avait été fait avant cela, on ne parlait que du tableau et le type des éléments du tableau n'importait pas.

La classe `TableauRedimensionnable` définie ici reprend les principes de la classe `ArrayList` de Java, spécialisée pour contenir des associations (ce que l'on pourrait noter `ArrayList<Association>`).

Exemple de code : table d'association Pour finir, on définit une classe pour représenter un ensemble d'associations entre des clés (ici, des chaînes de caractères) et des valeurs (ici, des entiers). On veut pouvoir ajouter de nouvelles associations à volonté, et faire en sorte de ne pas avoir deux associations pour une même clé. Autrement dit, ajouter une nouvelle association pour une clé déjà présente doit remplacer l'association d'origine.

On donne pour seul attribut à une table associative un tableau redimensionnable contenant des associations. Ce sont les méthodes qui devront s'assurer de ne jamais créer deux associations pour une même clé.

```
  public class TableAssociative {
    private TableauRedimensionnable tab;
    public TableAssociative() {
      this.tab = new TableauRedimensionnable();
    }
```

Pour ajouter une nouvelle association (k, v), on commence donc par chercher une éventuelle association déjà présente pour la même clé k. Si l'on en trouve une, il suffit de la modifier. Dans le cas contraire, on crée une nouvelle association et on l'ajoute au tableau redimensionnable.

```
    public void put(String k, int v) {
      int i = this.findKey(k);
      if (i >= 0) { // remplacer
        this.tab.get(i).setValue(v);
      } else { // ajouter □a la fin
        this.tab.add(new Association(k, v));
      }
    }
  }
```

Cette classe `TableAssociative` mime, du point de vue de l'utilisateur, le comportement de la classe `HashMap` de Java, en fixant les types des clés et des valeurs. Notre `TableAssociative` correspond donc à une `HashMap<String, Integer>`. De même, la classe `Association` correspond à `Map.Entry<String, Integer>` en Java. Il y a cependant une différence de poids à noter : notre version est inefficace (lente), alors que la vraie `HashMap` est basée sur un algorithme *très* efficace.