Fiche 5 : interfaces

IPO – Introduction à la programmation objet

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

V2, automne 2024

Retour sur l'encapsulation Dans la construction d'un programme on cherche, pour chaque élément, à distinguer la manière dont on l'utilise (point de vue extérieur) et la manière dont il fonctionne (point de vue interne). À un utilisateur extérieur, on présente un mode d'emploi qui dit comment utiliser l'élément, et quels résultats il faut attendre. Ce mode d'emploi extérieur est appelé une *interface*, ou une *abstraction*. Ce point de vue en revanche ne dit pas exactement la manière dont l'élément est organisé en interne.

En java, ceci se manifeste notamment avec la notion d'encapsulation et avec l'habitude de déclarer les attributs « privés », c'est-à-dire inaccessibles à un utilisateur extérieur, et les méthodes « publiques », c'est-à-dire utilisables par un utilisateur extérieur.

Cette séparation a plusieurs vertus.

— Un utilisateur qui n'a pas directement accès aux attributs d'une structure ne pourra pas, par mégarde, les modifier d'une manière qui ne respecterait pas les conventions de cette structure.

— Si l'utilisateur n'a besoin de connaître que le mode d'emploi, et pas tous les détails internes des structures qu'il utilise, alors il a moins de choses à connaître ou à maîtriser : il peut se concentrer sur sa partie.

— En limitant la manière dont les utilisateurs peuvent utiliser une structure, on évite que chaque modification du fonctionnement de cette structure doive être suivie d'une adaptation du code de l'utilisateur. Tant que l'on préserve l'interface, on peut corriger le code de la structure, lui ajouter une nouvelle fonctionnalité ou modifier ses caractéristiques internes sans que cela invalide les programmes utilisant cette structure.

Interface Une *interface* en Java résume les caractéristiques « extérieures » d'une structure. Une interface est essentiellement constituée de déclarations de méthodes, placées dans un bloc ressemblant à une définition de classe. On peut ainsi placer la définition

```
public interface Pile {
  void push(int n);  // ajouter un élément au sommet
  int  pop();        // retirer (et renvoyer) l'élémént du sommet
  int  peek();       // renvoyer l'élément du sommet
  boolean isEmpty(); // la pile est-elle vide ?
}
```

dans un fichier `Pile.java` pour définir l'*interface* d'une structure de données de *pile* (*LIFO : Last In, First Out*).

Chaque méthode est ici résumée à une déclaration : on fournit le nom de la méthode, les types de ses paramètres et le type de son éventuel résultat, mais on ne donne pas de code. Remarquez aussi que, par essence, une interface est une liste d'éléments *publics*. On se passe donc du mot-clé **`public`**, qui serait superflu. Dans l'interface, on ne donne en revanche pas d'attributs (on pourrait, mais cela aurait un sens particulier que nous verrons plus tard), ni de constructeurs (l'interface n'est qu'une vue abstraite, et ne permet pas à elle seule de créer un objet).

L'interface donne les éléments permettant d'utiliser une certaine structure. Notez que l'on peut donc écrire un programme utilisant une pile, en s'appuyant sur cette interface, avant même d'avoir des définitions concrètes pour les différents éléments de la pile. Voici par exemple un programme utilisant une pile `p` et ses méthodes `push` et `pop` pour déterminer la parenthèse ouvrante associée à chaque parenthèse fermante dans une chaîne de caractères. La seule partie laissée en suspens est la création de la pile, puisque nous n'avons à ce stade pas de constructeur.

```
public static int[] matching(String s) {
  int[] m = new int[s.length()];
  Pile p = new ???;
  for (int i=0; i<s.length(); i++) {
    if (s.charAt(i) == '(') { p.push(i); }
    else if (s.charAt(i) == ')') { m[i] = p.pop(); }
  }
  return m;
}
```

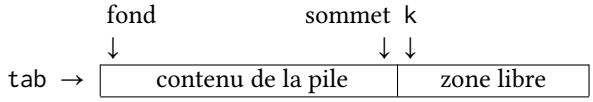
De la même manière que pour les classes, une interface définit un *type*. Ici, l'occurrence de `Pile` dans la déclaration `Pile p` désigne le type associé à la variable `p`.

Concrétisation d'une interface Une interface `I` étant fixée, on peut déclarer qu'une classe concrète `C` *réalise* cette interface en ajoutant une mention **`implements`** `I` dans la définition de `C`.

```
public class TabPile implements Pile {
  ...
}
```

Pour que cela soit admis, il *faut* que la classe concrète `TabPile` fournisse des définitions pour chacune des méthodes déclarées dans l'interface que l'on prétend réaliser. La classe concrète doit également contenir comme d'habitude des attributs, qui définissent sa structure, et au moins un constructeur, qui permet de créer des objets concrets.

On propose ici de représenter une pile d'entiers à l'aide d'un tableau primitif `tab`, de taille arbitraire mais fixe. Un attribut supplémentaire `k` donne l'indice de la première case libre du tableau, où sera ajouté le prochain élément introduit par la méthode `push`.



On peut alors simplement définir une telle classe `TabPile`, en déclarant ces deux attributs, en fournissant un constructeur qui initialise ces deux attributs, et en donnant une définition pour chacune des quatre méthodes mentionnées dans l'interface.

```
public class TabPile implements Pile {
  private int[] tab; // tableau sous-jacent
  private int k;     // première case libre

  public TabPile(int n) {
    this.tab = new int[n];
    this.k = 0;
  }

  public void    push(int n) { tab[k++] = n;    }
  public int     pop()       { return tab[--k]; }
  public int     peek()      { return tab[k-1]; }
  public boolean isEmpty()   { return k == 0;   }
}
```

En l'occurrence ici, les méthodes sont très simples et chacune tient en une ligne, mais on peut bien sûr avoir un code de complexité arbitraire. Note à propos de ce code : dans les méthodes, on accède aux attributs de l'objet courant sans utiliser le mot-clé **`this`**. Celui-ci est en effet facultatif lorsqu'il n'y a pas d'ambiguïté sur l'élément `tab` ou `k` désigné.

Utilisation conjointe d'une interface et d'une classe concrète associée Une interface ne fait que définir un *type*, et les méthodes auxquelles peut faire appel un utilisateur de ce type. Une classe concrète indique en revanche comment construire des objets concrets réalisant le type précédent. On peut donc compléter la ligne incomplète de notre fonction `matching` de la manière suivante.

```
Pile p = new TabPile(512);
```

On crée donc un objet de la classe (concrète) `TabPile`, que l'on associe à une variable `p` dont le type est donné par l'interface `Pile`. C'est possible car, du fait de la mention **`implements`** `Pile` dans la définition de la classe `TabPile`, tout objet de la classe `TabPile` est *à la fois* du type `TabPile` et du type `Pile`. En effet, tout objet `p` de la classe `TabPile` possède bien des méthodes `push`, `pop`, `peek` et `isEmpty`, et peut donc être utilisé à tout endroit où l'on attendait un élément de type `Pile`.

Notez qu'en revanche, la classe concrète `TabPile` peut définir des choses *en plus* de ce qui est déclaré dans l'interface. On pourrait ainsi par exemple ajouter une méthode publique

```
public boolean isFull() { return k == tab.length; }
```

dans la classe `TabPile`, qui fait référence à une caractéristique spécifique des piles concrètes `TabPile` mais n'existe pas dans l'interface générale des piles.

Cette possibilité de spécialisation de la classe concrète a deux conséquences.

1. Tout objet de type `Pile` ne peut pas nécessairement être considéré comme étant également du type `TabPile`. En effet, une pile concrète quelconque ne définit pas nécessairement la méthode `isFull` qu'un utilisateur de `TabPile` pourrait attendre.
2. Une même classe peut concrétiser simultanément *plusieurs* interfaces. On note par exemple

   ```
   public class TabPile implements Pile, Iterable<Integer> { ... }
   ```

   pour déclarer que la classe `TabPile` concrétise également l'interface `Iterable<Integer>` des structures permettant d'énumérer un ensemble d'entiers (voir `java.util.Iterable`).

Digression : surcharge statique et constructeurs multiples Dans une classe Java, on peut définir plusieurs méthodes portant le même nom, à conditions que ces méthodes puissent être distinguées par les types de leurs paramètres. On parle de *surcharge statique* de ces méthodes. Lorsque l'on utilise une telle méthode dans le code, le compilateur Java analyse les types des arguments fournis et choisit, parmi les différentes méthodes du même nom, celle dont les types des arguments attendus correspondent.

On utilise couramment ce système pour affecter des valeurs par défaut à certains paramètres qui ne seraient pas fournis, et cela s'applique aussi bien aux méthodes qu'aux constructeurs. On peut ainsi fournir deux constructeurs à la classe `TabPile` : celui déjà vu prenant en paramètre la taille à donner au tableau sous-jacent, et un autre ne prenant pas de paramètre et utilisant à la place une taille par défaut.

```
  public TabPile(int n) {
    this.tab = new int[n];
    this.k = 0;
  }
  public TabPile() {
    this.tab = new int[42];
    this.k = 0;
  }
```

Une création `new TabPile(5)` utilise alors le premier constructeur, et une création `new TabPile()` le deuxième. Remarque supplémentaire ici : la deuxième version correspond précisément à appeler la première version avec le paramètre par défaut `42`. Plutôt que de recopier le code, on peut donc explicitement faire appel à l'*autre* constructeur en lui fournissant cette valeur. On utilise le mot-clé **`this`** pour faire ainsi référence à un constructeur de la classe courante.

```
  public TabPile() { this(42); }
```

Note : utilisé ainsi, **`this`** fait référence à *un* constructeur de `TabPile`. Le choix parmi les différents constructeurs de cette classe est fait en regardant le type des paramètres fournis. On ne peut utiliser cette notation que dans un constructeur, et uniquement comme première instruction.

Concrétisations alternatives On peut proposer de multiples manières de concrétiser une interface donnée, les différents classes obtenues pouvant avoir des caractéristiques très différentes. Voici une concrétisation de l'interface `Pile` bâtie sur une structure de liste de chaînée, dotée d'une méthode publique supplémentaire `size`.

```
  class Cellule {
    private int k;
    private Cellule next;
    public Cellule(int k, Cellule n) {
      this.k = k;
      this.next = n;
    }
    public int hd() { return k; }
    public Cellule tl() { return next; }
  }

  public class ListePile implements Pile {
    private Cellule liste;
    private int size;
    public ListePile() {
      this.liste = null;
      this.size = 0;
    }
    public void push(int n) {
      this.liste = new Cellule(n, this.liste);
      size++;
    }
    public int pop() {
      size--;
      int e = liste.hd();
      liste = liste.tl();
      return e;
    }
    public int peek() { return liste.hd(); }
    public boolean isEmpty() { return this.liste == null; }
    public int size() { return this.size; }
}
```

Tests automatisés avec JUnit Il est utile d'accompagner tout projet d'un ensemble de tests de ses fonctions principales, afin de se donner les meilleures chances de détecter les erreurs. On exécute ces tests d'une part pendant le développement initial (pour s'assurer que la version initiale est correcte), puis à nouveau après chaque modification ou mise à jour du code (pour s'assurer qu'on n'introduit pas d'erreurs a cette occasion). Le *test unitaire* consiste à tester chaque méthode indépendamment des autres, en l'exécutant dans différentes conditions et en vérifiant ses résultats et ses effets. L'outil JUnit permet d'inclure des tests unitaires dans un projet Java.

On organise les tests JUnit dans un répertoire séparé. Pour chaque classe `C` à tester du projet, on crée une classe de test `CTest` (dans IntelliJ, cette classe peut être créée à partir d'un clic droit sur la classe `C` elle-même). Dans la classe `CTest`, on définit des méthodes précédées de l'annotation `@Test`, qui décrivent les tests à effectuer. Voici quelques exemples reliés à la classe `TabPile`.

```
public class TabPileTest {
    @Test
    public void popTest() { ... }
    @Test
    public void sizeTest() { ... }
    @Test
    public void isEmptyTest() { ... }
}
```

Chaque test contient un ou plusieurs appels de la méthode à tester, avec des paramètres variés, puis des vérifications des résultats à l'aide d'*assertions*. JUnit propose pour cela une famille de méthodes dont le nom commence par `assert`.

```
    @Test
    public void isEmptyTest() {
        TabPile p = new TabPile(64);
        assertTrue(p.isEmpty); // vérifie que le résultat est 'true'
        p.push(27);
        assertFalse(p.isEmpty); // vérifie que le résultat est 'false'
    }
    @Test
    public void sizeTest() {
        TabPile p = new TabPile(64);
        assertEquals(0, p.size()); // vérifie que le résultat est '1'
        for (int i=1; i <= 100; i++) {
            p.push(i*i);
            assertEquals(i, p.size()); // on peut tester à grande échelle
        }
    }
```

On peut également tester la présence d'exceptions dans les cas où ce comportement est attendu. Cela peut arriver par exemple dans le cas d'un appel à pop avec une pile vide. Dans le code ci-dessous, l'appel à `assertThrows` prend en deuxième paramètre une lambda-expression, un élément de Java que nous reverrons plus tard.

```
    @Test
    void pop() {
        TabPile p = new TabPile(27);
        p.push(42);
        assertEquals(42, p.pop());
        assertThrows(Exception.class, () -> { p.pop(); });
    }
```

Remarque : on aurait également pu définir ces tests unitaires relativement à l'interface `Pile` plutôt que relativement à la classe concrète `TabPile`, pour pouvoir les utiliser à la fois pour `TabPile` et pour `ListePile`. Cela demande toutefois une petite manipulation que nous verrons la semaine prochaine.