Interfaces et généricité

DU-ISN

Université Grenoble Alpes

Plan

Interfaces

Généricité

Notion d'interface

Contrat passé par une classe qui implémente l'interface

- ensemble de méthodes que la classe doit définir
- type de référence associé permettant de manipuler les objets de la classe

Exemple

```
interface Pile {
    void empiler(int element);
    int depiler();
    boolean est_pile_vide();
}
```

pas de limite au nombre d'interfaces qu'une classe implémente

Exemple d'implémentation avec une liste

```
class PileListe implements Pile {
    Maillon sommet;

    // Les méthodes implémentant l'interface
    // doivent être publiques
    public void empiler(int element) {
        Maillon m = new Maillon(element, sommet);
        sommet = m;
    }
    public int depiler() {
        // Exception si sommet == null (pile vide)
        int resultat = sommet.element;
        sommet = sommet.suivant;
        return resultat;
    }
    public boolean est_pile_vide() {
        return sommet == null;
    }
}
```

Exemple d'implémentation avec un tableau

```
class PileTableau implements Pile {
    int [] elements; int sommet;

    PileTableau() {
        elements = new int[10];
        sommet = 0;
    }
    public void empiler(int element) {
        // Exception si sommet >= elements.length
        elements[sommet++] = element;
    }
    public int depiler() {
        // Exception si sommet <= 0
        return elements[--sommet];
    }
    public boolean est_pile_vide() {
        return sommet == 0;
    }
}
```

Utilisation (programme principal)

```
class EssaiPile {
    public static void main(String [] args) {
        Pile p;

        p = new PileListe();
        // ou alors
        p = new PileTableau();
        p.empiler(362);
        p.empiler(42);
        System.out.println(p.depiler());
    }
}
```

Par rapport à l'héritage

On aimerait parfois qu'une classe hérite de plusieurs autres

- elle combine plusieurs comportements (un carré est une forme dessinable, pouvant subir une rotation, . . .)
- sa nature est multiple (un carré est à la fois un rectangle et un losange particuliers)

Les interfaces compensent l'absence d'héritage multiple en java

- type et comportement implémentable par une classe
- pas de limite au nombre d'interfaces implémentées

Reste différent de l'héritage multiple

- pas d'attributs communs dans les interfaces
- pas de code partagé (java <8) ou code partagé limité

Plan

Interfaces

Généricité

Généricité

Classe ou interface paramétrée par un type

- ce type n'est pas connu à la compilation
- rend le code indépendant du type des objets manipulés

Par exemple, une liste chaînée peut contenir

- des entiers
- des nombres à virgule flottante
- des chaînes de caractères
- ...

Lors de la déclaration d'une référence générique ou à la création d'un objet générique

- il faut préciser quel est le type attendu en paramètre
- limité aux types associés aux références (classe ou interface)

Exemple

Déclaration, paramètre formel ajouté au nom de la classe

```
class Couple<Type> {
    Type a, b;

    Couple(Type va, Type vb) {
        a = va;
        b = vb;
    }
}
```

Utilisation, paramètre effectif ajouté au type

```
class Exemple {
    public static void main(String [] args) {
        Couple<Integer> c =
            new Couple<Integer>(new Integer(42),
                                new Integer(1));
        ...
    }
}
```

Exemple

Déclaration, paramètre formel ajouté au nom de la classe

```
class Couple<Type> {
    Type a, b;

    Couple(Type va, Type vb) {
        a = va;
        b = vb;
    }
}
```

Utilisation, paramètre effectif ajouté au type

```
class Exemple {
    public static void main(String [] args) {
        Couple<Double> c =
            new Couple<Double>(new Double(42.1),
                               new Double(1.42));
        ...
    }
}
```

Exemple

Déclaration, paramètre formel ajouté au nom de la classe

```
class Couple<Type> {
    Type a, b;

    Couple(Type va, Type vb) {
        a = va;
        b = vb;
    }
}
```

Utilisation, paramètre effectif ajouté au type

```
class Exemple {
    public static void main(String [] args) {
        Couple<String> c =
            new Couple<>("titi", "tutu");

        ...
    }
}
```

Différence avec le polymorphisme

La généricité est un type de polymorphisme

- appelé polymorphisme paramétré
- comparable à l'utilisation du type `Object`

Avantages par rapport au type `Object`

- pas de conversion explicites (*casts*)
- le compilateur vérifie le typage (en général)

Sur la pile (interface Pile)

On remarque que les méthodes peuvent renvoyer des références du type donné en paramètre

```
interface PileGenerique<E> {
    void empiler(E element);
    E depiler();
    boolean est_pile_vide();
}
```

Sur la pile (Maillon)

Un maillon dont les éléments sont d'un type E encore inconnu

```
// classe Maillon paramétrée par E
class MaillonGenerique<E> {
    E element;
    // Utilisation de E (paramètre)
    // comme type effectif pour le suivant
    MaillonGenerique<E> suivant;

    MaillonGenerique(E e, MaillonGenerique<E> s) {
        element = e;
        suivant = s;
    }
}
```

Sur la pile (PileListe)

```
class PileListeGenerique<E>
                implements PileGenerique<E> {
    MaillonGenerique<E> sommet;

    public void empiler(E element) {
        MaillonGenerique<E> m;
        m = new MaillonGenerique<>(element, sommet);
        sommet = m;
    }
    public E depiler() {
        E resultat;
        resultat = sommet.element;
        sommet = sommet.suivant;
        return resultat;
    }
    public boolean est_pile_vide() {
        return sommet == null;
    }
}
```

Problème avec les tableau

Pas de tableaux génériques en java

- limite du langage
- due à la représentation interne des types

Contournement

- polymorphisme classique : tableau d'`Object`
- *casts* dans le type générique aux bon endroits

Sur la pile (PileTableau)

```
class PileTableauGenerique<E>
                implements PileGenerique<E> {
    // Tableaux génériques interdits par java
    Object [] elements; int sommet;

    PileTableauGenerique() {
        elements = new Object[10]; sommet = 0;
    }
    public void empiler(E element) {
        elements[sommet++] = element;
    }
    public E depiler() {
        // Object uniquement dans la classe
        return (E) elements[--sommet];
    }
    public boolean est_pile_vide() {
        return sommet == 0;
    }
}
```

Sur la pile (programme principal)

```
class EssaiPile {
    public static void main(String [] args) {
        PileGenerique<Integer> p;

        p = new PileListeGenerique<>();
        // ou alors
        p = new PileTableauGenerique<>();
        p.empiler(362);
        p.empiler(42);
        System.out.println(p.depiler());
    }
}
```

Contraintes sur les types génériques

Si on veut qu'un type générique

- soit ordonné
- dispose d'une distance entre objets de ce type
- supporte certaines opérations

Il faut qu'il soit plus spécifique qu'`Object`

- on le contraint lors de sa déclaration
- on précise qu'il doit étendre une classe ou interface particulière

Exemple :

```
class FAP<E extends Comparable<E>> {
...
```

Pour aller plus loin

Les méthodes aussi peuvent être génériques

```
<E> void afficher(Pile<E> p) {
...
```

... sachant que tout objet est affichable !