Programmation Objet
Java – ADTs

Michail Lampis
michail.lampis@dauphine.fr

Abstract Data Types

- Type de données abstrait :
  - Une spécification mathématique d'un ensemble de données
  - Des opérations qu'on peut effectuer sur ces données
- Exemples :
  - Conteneur (Container)
  - Pile, File
  - Dictionnaire (Tableaux Associatif)
  - …
- ADT == Classe !
  - On va utiliser les principes de l'OOP pour implémenter quelques ADTs

Set

- Type de données abstrait : Ensemble
- Représente un ensemble dans le sens mathèmatique
  - Opérations souhaitées :
    - Membership : $x \in S$?
    - Addition : $S = S \cup \{x\}$
    - Union
    - Intersection
    - Est-vide ?
    - …
- Quel type de données va-t-il contenir ?
- Comment l'implémenter ?

Classe Set

- Signature (partielle) possible...

```
class Set {
        public Object [] toArray();
        public String toString();
        public Set Copy() ;
        public void add(Object o);
        public boolean isMember(Object o);
        public Set Union(Set s);
        public boolean isEmpty();
}
```

Implémentation

- Que nous manque-t-il ?
  - On a défini quelles sont les opérations de cette classe, mais sans aucun détail d'implémentation
    - On peut utiliser un tableau, une liste chaînée, un dictionnaire,… ?
- Héritage : On peut définir une classe qui ne contient que la signature des opérations souhaitées
- On laisse les définitions pour les sous-classes
  - Avantage : la super-classe peut être utilisé partout où on a besoin de cet ADT
  - Avantage : Plusieurs versions des sous-classes peuvent être plus efficaces dans des contextes différents.

Classes Abstraites

- Une classe abstraite (abstract class) est une classe qui contient des méthodes abstraites, donc des méthodes pas implémentées.
- On utilise une classe abstraite pour définir une classe qui n’est pas instantiable (pas de constructeur)
- Le but est d’hériter de cette super-classe
  - On obtient une hiérarchie des classes qui partagent leur fonctionnalité, même si elles utilisent des moyens différentes pour l’implémenter.

Méthodes Abstraites

- Méthode abstraite : une méthode qu'on est obligé de redéfinir dans une sous-classe (sinon→ sous-classe abstraite)
- Pas obligatoire que toutes les méthodes d'une classe soient abstraites !

```
abstract class Set {
        public Set Union(Set s){
                Set s1 = Copy(); //this.Copy() !!
                Object [] objs = s.toArray();
                for(Object item:objs) s1.add(item);
                return s1;
        }
}
```

Classe Set

- Classe abstraite

```
abstract class Set {
        abstract public Object [] toArray();
        abstract public String toString();
        abstract public Set Copy() ;
        abstract public void add(Object o);
        abstract public boolean isMember(Object o);
        public Set Union(Set s);
        abstract public boolean isEmpty();
}
```

ArraySet

- On peut essayer de définir cette structure de données en utilisant un tableau (qui va contenir les éléments du Set)

```
class ArraySet extends Set {
        Object [] data; //=null
        public Object [] toArray() { return data; }
        public boolean isMember(Object o){
                if(data==null) return false;
                for(Object item:data){
                        if(item.equals(o)) return true;
                }
                return false;
        }
...
```

La méthode equals

- Pourquoi écrit-on **item.equals(o)** et pas **item==o** ?

  Ex :

  Integer i1 = new Integer(5);

  Integer i2 = new Integer(5);

  System.out.println(i1==i2);

- == fait comparaison des références (adresses) pour des types non-primitifs. i1==i2 seulement s'il s'agit de la même adresse.
  - La méthode **equals**, définie pour la classe **Object,** donc pour tous les types non-primitifs, nous permet de faire des comparaisons plus logiques.

    i1.equals(i2) == true

La classe Integer

- On vient d’utiliser la classe Integer
  - Rappel : 8 types primitfs : **int, long, floar, double,…**
- Java nous offre pour chacun une version « wrapper »
  - Rappel : les 8 types primitifs ne sont pas des objets
    - **int x ; x.equals(5) ; // ??**
    - **void f(Object o) ; f(x) ;// ??**
  - Avec les wrapper types on peut utiliser ces 8 types où on peut utiliser des objets, comme des objets.
  - → Sémantiques des objets. Variable=référence,…
- Les types Integer, Double, Short, Long, … sont des sous-types de la class **Number**.

ArraySet

- Continuation...

```
class ArraySet extends Set {
        Object [] data; //=null
        …
        public void add(Object o) {
                if(isMember(o)) return ; //Pourquoi ?
                if(data==null){ data = new Object[1]; data[0]=o; return;}
                Object [] data2 = new Object[data.length+1];
                for(int i=0;i<data.length;i++) data2[i]=data[i];
                data2[data.length]=o;
                data = data2;
        }
…
```

ArraySet

- Continuation...

```
class ArraySet extends Set {
        Object [] data; //=null
        …
        public Set Copy(){
        ArraySet copy = new ArraySet();
        for(Object item:data) copy.add(item);
                return copy;
        }

        public boolean isEmpty() { return data==null; }
```

ArraySet

- Continuation...

```
class ArraySet extends Set {
        Object [] data; //=null
        …
        public ArraySet Copy(){
        ArraySet copy = new ArraySet();
        for(Object item:data) copy.add(item);
                return copy;
        }

        public boolean isEmpty() { return data==null; }
```

ArraySet

- Continuation...

```
class ArraySet extends Set {
        Object [] data; //=null
        …
        public String toString() {
            String s="";
            for(Object obj:data)
                  s=s+obj+",";
            return s;
      }
```

ArraySet Exemple

- Pour utiliser une telle classe :

  Set s = new ArraySet(); // Pourquoi pas ArraySet s ??

  s.add(5);

  s.add(3);

  s.add(5);

  s.add(4);

  System.out.println(s); //5,3,4

  Set s2 = new ArraySet();

  s2.add(6); s2.add(5);

  System.out.println(s.Union(s2)); //5,3,4,6

ArraySet Exemple

- Pour utiliser une telle classe :

  Set s = new ArraySet();

  s.add(new Point(2.2,3.3));

  s.add(new Point(2.2,3.3));

  System.out.println(s);

- Afficher le même Point deux fois !
  - Attn : on n'a pas redéfini la méthode **equals** pour la classe **Point**
  - Sans redéfinition, **equals équivaut** à ==

ArraySet Exemple

- Pour utiliser une telle classe :

```
Set s = new ArraySet();
s.add(new Point(2.2,2.2));
s.add(new Point(1.1,1.1));
Set s2 = s.Copy();
((Point)s.toArray()[0]).x=5.5; // ??
System.out.println(s);
System.out.println(s2);
//Affiche : Coords:5.5,2.2,Coords:1.1,1.1,
//              Coords:5.5,2.2,Coords:1.1,1.1,
```

Shallow Copy

- Attn : Dans notre implémentation on a fait une copie naïve :
  - On a utilisé simplement l'opération =
  - Pour les types références ça ne copie que les références
    - → on a fait une copie de l'adresse de chaque élément
    - → les deux copies de l'ensemble sont liées
- Ce n'est pas un problème avec les **Integer**
  - La classe **Integer** produit des objets **immuable**
  - → plus sûr de partager des références.
- La classe **Point** ne produit pas des objets immuables
  - Il faudrait utiliser le mot-clé **final**

ListSet

- Le but d'utiliser une classe abstraite est de nous permettre de définir plusieurs implémentations du même ADT
  - Pourquoi ? Questions d'efficacité. Il n'y a pas une solution qui est « la meilleur partout »
- Exemple : Implémentation List Chainée (à compléter pendant le TD)

```
class ListSet extends Set {

        ListSet next;

        Object o;

        int taille;

        public Object [] toArray();

        public void add(Object o);

        public boolean isMember(Object o);

        public Set Copy();

        public String toString();

        public boolean isEmpty();

}
```

ADT : Stack

- Une pile est un ADT qui nous donne (au moins) les opérations suivantes
  - **push** (empiler) : ajoute un élément dans la pile, au sommet
  - **pop** (dépiler) : retire l'élément qui se trouve au sommet
  - **top** : retourne l'élément qui se trouve au sommet, sans changer l'état de la pile.
- Pile == stockage LIFO (Last-In-First-Out)
- Tâches :
  - Donner une définition abstraite de l'ADT
  - Donner une implémentation avec des tableaux

Classe Abstraite Stack

```
public abstract class Stack {
        abstract public void push(Object o);
        abstract public Object pop();
        abstract public String toString();
        abstract public Object top() ;
}
```

- **Conseil :** quand on peut définir une méthode de manière raisonnable même dans la classe abstraite, c’est une bonne idée de le faire.

Classe Abstraite Stack

```
public abstract class Stack {
        abstract public void push(Object o);
        abstract public Object pop();
        abstract public String toString();
        public Object top(){
                Object o = pop();
                push(o);
                return o;
        }
        abstract public boolean isEmpty();
}
```

Références

- **Attn :** Objets == variables de référence (==pointeurs)
  - La méthode pop retourne une référence pour l'objet qui se trouve au sommet de la pile
  - Une fonction qui manipule cette référence va modifier l'objet qui se trouve dans la pile !
- Exemple : on empile un objet de la classe Point…
  - Solution : on n'empile que des classes immuables
  - Solution : l'interface Cloneable (à voir plus tard).

Classe Stack:Utilisation

```
public static void main(String [] args){
        Stack s = new ArrayStack();
        //Stack s = new Stack(); ??
        //ArrayStack s = new ArrayStack(); ??
        for(int i=1;i<10;i++){
            s.push(i);
            System.out.println(s);
        }
```

- Qu'affiche le programme ?
- Peut-on le compiler ?
- Peut-on l'exécuter ?

Classe Stack:Utilisation

```
public static void main(String [] args){
        Stack s = new ArrayStack();
        for(int i=1;i<10;i++){
                s.push(i);
                System.out.println(s);
        }
        for(int i=1;i<5;i++){
                System.out.println(s.pop());
        }
```

- Qu'affiche le programme ?
- Compilation : OK
- Exécution : Il nous faut une implémentation concréte !!

Emballage

- Détail : on a appelé **push(i)**
  - **push** prend comme paramètre un **Object**
  - **i** est une variable **int** (→ primitive)
  - **int** n'est pas une sous-classe de **Object**
- Cependant, le compilateur accepte :
  - Emballage automatique == la conversion automatique d'un type primitif vers sa version emballée
    - int → Integer
    - double → Double
    - ...
  - Le compilateur tente une conversion automatique (sous quelques conditions) pour faire marcher le programme...

ArrayStack

- Implémentation 1 : utiliser un tableau
  - Rappel: tableau == taille fixée

```
public class ArrayStack extends Stack{
        private Object [] data;
```

- Idée : on va mettre les éléments au tableau
  - Dernière position == Sommet de la pile
  - **data** == null → pile vide

ArrayStack

```
public class ArrayStack extends Stack{
        private Object [] data;
        public boolean isEmpty() { return data==null; }
        public void push(Object o){
                if(data==null){ data = new Object[1];
                        data[0] = o; return;   }
                Object [] data2 = new Object[data.length+1];
                for(int i=0;i<data.length;i++)
                        data2[i]=data[i];
                data2[data.length] = o;
                data = data2;
        }
```

ArrayStack

```
public class ArrayStack extends Stack{
        private Object [] data;
        public Object pop(){
          if(data==null) return null;
          Object o = data[data.length-1];
          Object [] data2 = new Object[data.length-1];
          for(int i=0;i<data2.length;i++)
                data2[i]=data[i];
          data = data2;
          return o;
        }
```

Resize Operation...

ArrayStack

```
public class ArrayStack extends Stack{
        private Object [] data;
        public String toString(){
                String s="";
                for(Object i:data)
                        s = s+i.toString();
                return s;
        }
```

Efficacité

- Questions à poser :
  - Notre implémentation, est-elle correcte ?
    - A-t-on respecté le contrat ?
  - Est-elle efficace ?
    - Mémoire ?
    - Temps d’exécution ?
      - Par opération push/pop/top
  - **Attn :** chaque opération push/pop fait une copie de **data** !

Over-ride

- On a dit que c’est une bonne idée de donner une implémentation abstraite si possible
  - Rappel :

```
public Object top(){
        Object o = pop();
        push(o);
        return o;
}
```

- Coût de cette opération : O(n)

Over-ride

- La classe concrète peut redéfinir les méthodes de la sous-classe
  - Avantage : on a plus d'information sur l'implémentation !

```
public Object top(){
        if(data==null) return null;
        return data[data.length-1];
}
```

- Coût : O(1)

Over-ride

- Tester les deux implémentations avec le programme :

```
for(int i=1;i<10000;i++){
        s.push(i);
}
System.out.println(s);
for(int i=1;i<500000;i++){
        s.top();
}
```

- Leçon à retenir
  - Avoir une implémentation par défaut, c’est bien
  - Avoir aussi une implémentation precisée, c’est mieux !

Efficacité

- Peut-on utiliser des tableaux pour implémenter une pile de manière efficace ?
  - Oui, si on évite les changements de taille fréquents.
  - Ça va coûter un peu en termes de mémoire…
  - Un petit sacrifice de mémoire →

    un grand bénéfice de temps d’exécution.
- Idée, quand il faut plus de mémoire, on double la taille du tableau.

ArrayStack2

```
public class ArrayStack2 extends Stack{
        private Object [] data;
        int size=0;
        public boolean isEmpty() { return size==0; }
        private void doubleSize(){
                Object [] data2 = new Object[data.length*2];
                for(int i=0;i<data.length;i++)
                        data2[i]=data[i];
                data = data2;
        }
```

ArrayStack2

```
public class ArrayStack2 extends Stack{
        private Object [] data;
        int size=0;
        public void push(Object o){
                if(data==null){ data = new Object[1];
                        size = 1;  data[0] = o;  return;  }
                if(size == data.length) doubleSize();
                data[size++]=o;
        }
```

ArrayStack2

```
public class ArrayStack2 extends Stack{
        public Object pop(){
                if(isEmpty()) return null;
                return data[size--];
        }
        public String toString(){
                String s="";
                for(int i=0; i<size; i++)
                        s = s+data[i].toString();
                return s;
        }
```

Réflexions

- Utilisation de mémoire :
  - Max mémoire utilisé ≤ 2*max mémoire nécessaire
    - Mais ! La mémoire utilisée ne diminue jamais !
  - Temps d’exécution :
    - Pop : O(1). :-)
    - Top : O(1), même avec l’implémentation naïve. :-)
    - Push : O(n) ?
      - En fait, O(1) par opération (coût moyen)
      - La plupart des opérations en O(1)
      - 1 opération sur n prend O(n)
  - Coût de n push = 1+2+1+4+1+1+1+8+1+1+1+..+16+..

ListStack

- Utiliser une liste chaînée pour implémenter une pile
  - Avantage : pas besoin de changer la classe abstraite
  - Tous les programmes qui utilisent la classe fonctionnent
- Efficacité :
  - Opérations en O(1) toujours (pas en moyenne)
  - Utilisation de mémoire optimale