Cours 7

Classes locales

Clonage

Divers: tableaux

Classe locale

- Au lieu de créer d'implémenter Iterator on pourrait aussi créer une méthode qui retourne un iterateur.

Exemple parcourir

```
public static Iterator<Object> parcourir(final Object[] data){
        class Iter implements Iterator<Object>{
            private int pos=0;
            public boolean hasNext(){
                return (pos <data.length);
            }
            public Object next() throws NoSuchElementException{
                if (pos >= data.length)
                    throw new NoSuchElementException();
                return data[pos++];
            }
            public void remove(){
                throw new  UnsupportedOperationException();
            }
        }
        return new Iter();
    }
```

et l'appel

```
Integer[] tab=new Integer[12];
//…
afficher(parcourir(tab));
```

Remarques

- `parcourir()` retourne un itérateur pour le tableau passé en paramètre.
- l'itérateur implémente `Iterator`
  - mais dans une classe locale à la méthode `parcourir`
  - la méthode `parcourir` retourne un objet de cette classe.
- `data[]` est déclaré final:
  - même si tous les objets locaux sont dans la portée de la classe locale, la classe locale ne peut accéder aux variables locales que si elles sont déclarées final.

Anonymat…

- mais était-il utile de donner un nom à cette classe qui ne sert qu'à créer un objet Iter?

Classe anonyme

```
public static Iterator<Object> parcourir1( final Object[] data){
            return new Iterator<Object>(){
            private int pos=0;
            public boolean hasNext(){
                return (pos <data.length);
            }
            public Object next() throws NoSuchElementException{
                if (pos >= data.length)
                    throw new NoSuchElementException();
                return data[pos++];
            }
            public void remove(){
                throw new  UnsupportedOperationException();
            }
        };
    }
```

Exemple interface graphique:

```
jButton1.addActionListener(new ActionListener(){
   public void actionPerformed(ActionEvent evt){
          jButton1ActionPerformed(evt);
   }
});
```

Principe…

- ActionListener est une interface qui contient une seule méthode
  - void **actionPerformed**(ActionEvent e)
  - cette méthode définit le comportement voulu si on presse le bouton
- Il faut que le Button `jButton1` associe l'événement correspondant au fait que le bouton est pressé l'ActionListener voulu: `addActionListener`

Dans l'exemple

1. `jButton1ActionPerformed` est la méthode qui doit être activée
2. Création d'un objet de type ActionListener:
   1. (Re)définition de `ActionPerformed` dans l'interface ActionListener: appel de `jButton1ActionPerformed`
   2. classe anonyme pour ActionListener
   3. operateur new
3. ajout de cet ActionListener comme écouteur des événements de ce bouton `jButton1.addActionListener`

Chapitre IV

1. Interfaces
2. Classes imbriquées
3. Objets, clonage

Le clonage

- les variables sont des références sur des objets -> l'affectation ne modifie pas l'objet
- la méthode `clone` retourne un nouvel objet dont la valeur initiale est une copie de l'objet

Points techniques

- Par défaut la méthode `clone` de `Object` duplique les champs de l'objet (et dépend donc de la classe de l'objet)
- L'interface `Cloneable` doit être implémentée pour pouvoir utiliser la méthode `clone` de `Object`
  - Sinon la méthode `clone` de `Object` lance une exception `CloneNotSupportedException`
- De plus, la méthode `clone` est `protected` -> elle ne peut être utilisée quand dans les méthodes définies dans la classe ou ses descendantes (ou dans le même package).

En conséquence

- en implémentant `Cloneable`, `Object.clone()` est possible pour la classe et les classes descendantes
  - Si `CloneNotSupportedException` est captée, le clonage est possible pour la classe et les descendants
  - Si on laisse passer `CloneNotSupportedException`, le clonage peut être possible pour la classe (et les descendants) (exemple dans une collection le clonage sera possible si les éléments de la collection le sont)
- en n'implémentant pas `Cloneable`, `Object.clone()` lance uniquement l'exception et en définissant une méthode clone qui lance une `CloneNotSupportedException`, le clonage n'est plus possible

Exemple

```
class A implements Cloneable{
    int i,j;
    A(int i,int j){
        this.i=i; this.j=j;
    }
    public String toString(){
        return "(i="+i+",j="+j+")";
    }
    protected Object clone()
       throws CloneNotSupportedException{
        return super.clone();
    }
}
```

Suite

```
A a1=new A(1,2);
A a2=null;
try {// nécessaire!
    a2 =(A) a1.clone();
} catch (CloneNotSupportedException ex) {
    ex.printStackTrace();
}
```

donnera:

a1=(i=1,j=2) a2=(i=1,j=2)

Suite

```
class D extends A{
    int k;
    D(int i,int j){
        super(i,j);
        k=0;
    }
     public String toString(){
        return ("(k="+k+")"+super.toString());
     }
}
//…
         D d1=new D(1,2);
        D d2=null;
        try { //nécessaire
            d2=(D) d1.clone();
        } catch (CloneNotSupportedException ex) {
            ex.printStackTrace();
        }
        System.out.println("d1="+d1+" d2="+d2);
    }
```

Remarques

- `super.clone();` dans `A` est nécessaire il duplique *tous* les champs d'un objet de D
- Pour faire un clone d'un objet D il faut capter l'exception.

Suite

```
class B implements Cloneable{
    int i,j;
    B(int i,int j){
        this.i=i; this.j=j;
    }
    public String toString(){
        return "(i="+i+",j="+j+")";
    }

    protected Object clone(){
        try {
            return super.clone();
        } catch (CloneNotSupportedException ex) {
            ex.printStackTrace();
            return null;
        }
    }
}
```

Suite

```
class C extends B{
    int k;
    C(int i,int j){
        super(i,j);
        k=0;
    }
    public String toString(){
        return ("(k="+k+")"+super.toString());
    }
}//...
 B b1=new B(1,2);
 B b2 =(B) b1.clone();
 C c1=new C(1,2);
 C c2 =(C) c1.clone();
```

Pourquoi le clonage?

- Partager ou copier?
- Copie profonde ou superficielle?
  - par défaut la copie est superficielle:

Exemple

```
class IntegerStack implements Cloneable{
    private int[] buffer;
    private int sommet;
    public IntegerStack(int max){
        buffer=new int[max];
        sommet=-1;
    }
    public void empiler(int v){
        buffer[++sommet]=v;
    }
    public int dépiler(){
        return buffer[sommet--];
    }
    public IntegerStack clone(){
        try{
            return (IntegerStack)super.clone();
        }catch(CloneNotSupportedException e){
            throw new InternalError(e.toString());
        }
    }
}
```

Problème:

```
IntegerStack un=new IntegerStack(10);
un.emplier(3);
un.empiler(9)
InetegerStack deux=un.clone();
```

Les deux piles partagent les mêmes données…

Solution…

```
public IntegerStack clone(){
  try{
     IntegerStack nObj = (IntegerStack)super.clone();
     nObj.buffer=buffer.clone();
     return nObj;
  }catch(CloneNotSupportedException e){
       //impossible
       throw new InternalError(e.toString());
   }
 }
```

Copie profonde

```
public class CopieProfonde implements Cloneable{
    int val;
    CopieProfonde n=null;
    public CopieProfonde(int i) {
        val=i;
    }
    public CopieProfonde(int i, CopieProfonde n){
        this.val=i;
        this.n=n;
    }
    public Object clone(){
        CopieProfonde tmp=null;
        try{
            tmp=(CopieProfonde)super.clone();
            if(tmp.n!=null)
                tmp.n=(CopieProfonde)(tmp.n).clone();
        }catch(CloneNotSupportedException ex){}
        return tmp;
    }
}
```

Suite

```
class essai{
    static void affiche(CopieProfonde l){
        while(l!=null){
            System.out.println(l.val+" ");
            l=l.n;
        }
    }
    public static void main(String[] st){
        CopieProfonde l=new CopieProfonde(0);
        CopieProfonde tmp;
        for(int i=0;i<10;i++){
            tmp=new CopieProfonde(i,l);
            l=tmp;
        }
        affiche(l);
        CopieProfonde n=(CopieProfonde)l.clone();
        affiche(n);
    }
}
```

Chapitre V

Enumeration, tableaux, conversion de types, noms

Types énumérés

- Exemple:
  - `enum Couleur {PIQUE, CŒUR, CARREAU, TREFLE,}`
  - définit des constantes énumérées (champs static de la classe)
  - on peut définir des méthodes dans un enum
  - des méthodes
    - public static E[] values() retourne les constantes dans l'ordre de leur énumeration
    - public static E valueOf(String nom) la constante associé au nom
  - un type enum étend implicitement java.lang.Enum (aucune classe ne peut étendre cette classe)

Tableaux

- collection ordonnée d'éléments,
- les tableaux sont des Object
- les composants peuvent être de types primitifs, des références à des objets (y compris des références à des tableaux),

Tableaux

- int [] tab= new int t[3];
  - déclaration d'un tableau d'int
  - initialisé à un tableau de 3 int
- indices commencent à 0
- contrôle de dépassement
  - ArrayIndexOutOfBoundException
- length donne la taille du tableau

Tableaux

- un tableau final: la référence ne peut être changée (mais le tableau référencé peut l'être)
- tableaux de tableaux:
- exemple:

```
public static int[][] duplique(int[][] mat){
        int[][] res= new int[mat.length][];
        for(int i=0;i<mat.length;i++){
            res[i]=new int[mat[i].length];
            for (int j=0;j<mat[i].length;j++)
                res[i][j]=mat[i][j];
        }
        return res;
    }
```

Tableaux

- exemple:

```
public static void affiche(int [][] tab){
        for(int[] d:tab){
            System.out.println();
            for(int v:d)
                System.out.print(v+" ");
        }
    }
```