

Classes locales

- classes définies à l'intérieur d'un bloc de code,
- analogue à des variables locales: une classe interne locale n'est pas membre de la classe et donc pas d'accès,
- usage: créer des instances qui peuvent être passées en paramètres
- usage: créer des objets d'une extension d'une classe qui n'a de sens que localement (en particulier dans les interfaces graphiques)



Exemple

- classes Collections (ou Containers):
classes correspondant à des structures de données.
 - exemples: List, Set, Queue, Map.
- L'interface Iterator permet de parcourir tous les éléments composant une structure de données.



Iterator

```
public interface Iterator<E>{  
    boolean hasNext();  
    E next() throws NoSuchElementException;  
    void remove()throws  
        UnsupportedOperationException,  
        IllegalStateException;  
}
```

Exemple: MaCollection

```
class MaCollection implements Iterator<Object>{
    Object[] data;
    MaCollection(int i){
        data=new Object[i];
    }
    MaCollection(Object ... l){
        data=new Object[l.length];
        for(int i=0;i<l.length;i++)
            data[i]=l[i];
    }
    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();
    }
}
```

Et une iteration:

```
public class Main {
    public static void afficher(Iterator it){
        while(it.hasNext()){
            System.out.println(it.next());
        }
    }
    public static void main(String[] args) {
        MaCollection m=new MaCollection(1,2,3,5,6,7);
        afficher(m);
    }
}
```



Classe locale

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

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. opérateur `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.empiler(3);  
un.empiler(9)  
IntegerStack 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 énumération
 - `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
 - `ArrayIndexOutOfBoundsException`
- `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+" ");  
    }  
}
```



Initialisations

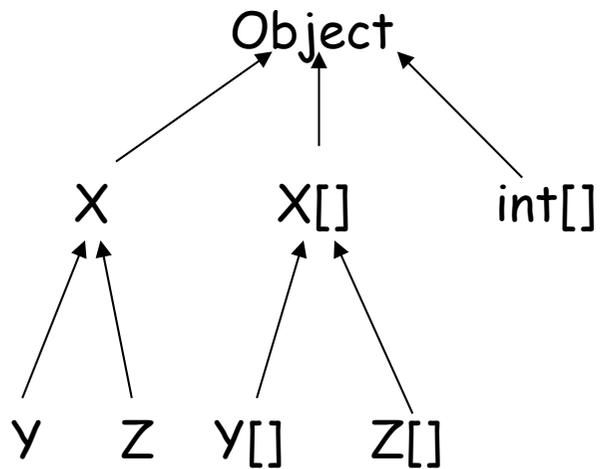
```
static int[][] pascal={  
    {1},  
    {1,1},  
    {1,2,1},  
    {1,3,3,1},  
};
```

```
String[] nombre= {"un", "deux", "trois", "quatre"};
```

Exemple

```
for(int i=1;i<p.length;i++){
    p[i]=new int[i+1];
    p[i][0]=1;
    for(int j=1; j<i;j++){
        p[i][j]=p[i-1][j-1]+p[i-1][j];
    }
    p[i][i]=1;
}
```

Tableau et héritage



- `Y[] yA=new Y[3];`
- `X[] xA=yA; //ok`
- `xA[0]=new Y();`
- `xA[1]=new X(); //non`
- `xA[1]=new Z(); //non`



Noms

- il ya 6 espaces de noms
 - package
 - type
 - champs
 - méthode
 - variable locale
 - étiquette

Noms!?

```
package divers;
class divers{
    divers divers(divers divers){
        divers:
        for(;;){
            if (divers.divers(divers)==divers)
                break divers;
        }
        return divers;
    }
}
```

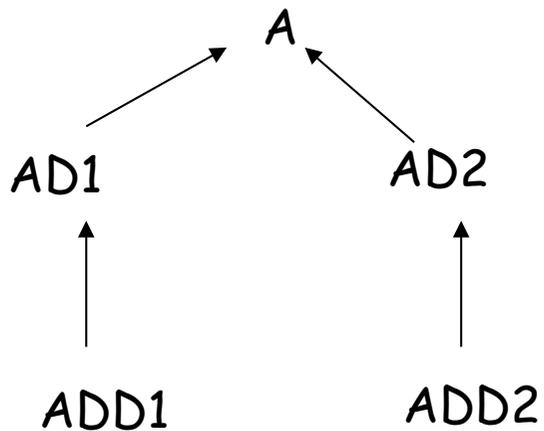
Trouver la bonne méthode

- Il faut pouvoir déterminer une seule méthode à partir d'une invocation avec des paramètres
- problèmes: héritage et surcharge
 - (en plus des problèmes liés à la généricité)
- principe trouver la méthode "la plus spécifique"

Règles

1. *déterminer dans quelle classe chercher la méthode (uniquement par le nom)*
2. trouver les méthodes dans la classe qui peuvent s'appliquer
 1. sans "boxing" sans nombre variable d'arguments
 2. avec boxing
 3. avec un nombre variable d'arguments
3. si une méthode a des types de paramètres qui peuvent être affectés à une autre des méthodes de l'ensemble -> la supprimer
4. s'il ne reste qu'une méthode c'est elle (sinon ambiguïté sauf s'il s'agit de méthodes abstraites)

Exemple



```
void f(A a,AD2 ad2)//un  
void f(AD1 ad1,A a)//deux  
void f(ADD1 add1, AD2 s)//trois  
void f(A ... a)//quatre
```

```
f(Aref, AD2ref);//a  
f(ADD1ref, Aref);//b  
f(ADD1ref, ADD2ref);//c  
f(AD1ref, AD2ref);//d  
f(AD2ref, AD1ref);//e
```