

# Chapitre IV

---

Interfaces, classes imbriquées, Object

# Chapitre IV

---

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

# classes abstraites

---

```
abstract class Benchmark{
    abstract void benchmark();
    public final long repeat(int c){
        long start =System.nanoTime();
        for(int i=0;i<c;i++)
            benchmark();
        return (System.nanoTime() -start);
    }
}
class MonBenchmark extends Benchmark{
    void benchmark(){}
    public static long mesurer(int i){
        return new MonBenchmark().repeat(i);
    }
}
public static void main(String[] st){
    System.out.println("temps="+
        MonBenchmark.mesurer(1000000));
}
```

Résultat:

temps=6981893

# Interfaces

- une interface est une « classe » sans données membres qui ne contient que des déclarations de méthodes d'instances (=méthodes abstraites)
- peut contenir des constantes, des déclarations de méthodes, des méthodes par défaut, des méthodes statiques (et des types imbriqués) (java 8)
- un interface ne peut pas être instanciée (mais peut être implémentée)

# Example:

---

```
interface Comparable<T>{
    int compareTo(T obj);
}
class Couple implements Comparable<Couple>{
    int x,y;
    //
    public int compareTo(Couple c){
        if(x<c.x)return 1;
        else if (c.x==x)
            if (c.y==y)return 0;
        return -1;
    }
}
```

# Quelques interfaces...

---

- Cloneable: est une interface vide(!) un objet qui l'implémente peut redéfinir la méthode clone
- Comparable: est une interface qui permet de comparer les éléments (méthode compareTo)
- Runnable: permet de définir des "threads"
- Serializable: un objet qui l'implémente peut être "sérialisé" = converti en une suite d'octets pour être sauvegarder.

# Déclarations

---

- une interface peut déclarer:
  - des constantes (toutes les variables déclarées sont par défaut `static public` et `final`)
  - des méthodes (elles sont implicitement `abstract`)
  - des classes internes et des interfaces

# Extension

---

les interfaces peuvent être étendues avec  
extends:

- Exemple:

```
public interface SerializableRunnable  
    extends Serializable, Runnable;
```

(ainsi une interface peut étendre de plusieurs façons  
une même interface, mais comme il n'y a pas  
d'implémentation de méthodes et uniquement des  
constantes ce n'est pas un problème)



# Exemple

---

```
interface X{
    int val=0;
}
interface Y extends X{
    int val=1;
    int somme=val+X.val;
}
class Z implements Y{}

public class InterfaceHeritage {
    public static void main(String[] st){
        System.out.println("Z.val="+Z.val+" Z.somme="+Z.somme);
        Z z=new Z();
        System.out.println("z.val="+z.val+
            " ((Y)z).val="+((Y)z).val+
            " ((X)z).val="+((X)z).val);
    }
}
-----
Z.val=1 Z.somme=1
z.val=1 ((Y)z).val=1 ((X)z).val=0
```

# Redéfinition, surcharge

---

```
interface A{
    void f();
    void g();
}
interface B{
    void f();
    void f(int i);
    void h();
}
interface C extends A,B{}
```

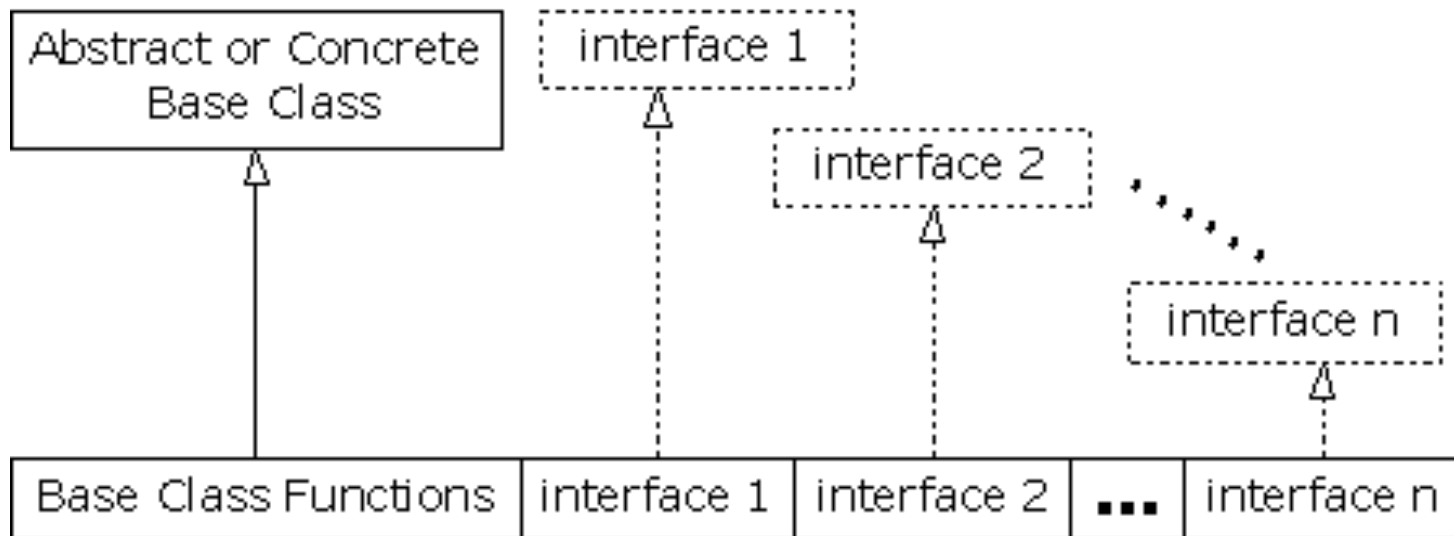
Rien n'indique que les deux méthodes `void f()` ont la même "sémantique". Comment remplir le double contrat?

# Interfaces et héritage multiple

---

- Il n'y a pas d'héritage multiple en Java: une classe ne peut être l'extension que d'une seule classe
- Par contre une classe peut implémenter plusieurs interfaces (et être l'extension d'une seule classe)
- Une interface ne contient (essentiellement) que des déclarations de méthodes
- (Une interface est un peu comme une classe sans données membres et dont toutes les méthodes seraient abstraites)

# Héritage "multiple" en java



# Remarques...

---

- Pourquoi, a priori, l'héritage multiple est plus difficile à implémenter que l'héritage simple?
- Pourquoi, a priori, implémenter plusieurs interfaces ne pose pas (trop) de problèmes?
- (Comment ferait-on dans un langage comme le C?)

# Compléments java 8

---

- Possibilité de définir des méthodes par défaut, des constantes et des méthodes statiques.
- méthode par défaut (`default`) peut être redéfinie ou rendue abstraite.
- les variables sont statiques et `final`

# Compléments java 8...

---

```
interface I{
    int val=100;
    int f(int j);
    default int calcul(int i){return mul(i)*f(i);}
    static public int mul(int i){return val*i;}
}
class B implements I{
    public int f(int i){return i+i;}
    public int mul(int i){return f(i)*f(i);}
}
class C implements I{
    public int f(int i){return i*i;}
}
```

---

```
int i=(new B()).calcul(5);// 5000
i=(new C()).calcul(3); // 2700
```

# Chapitre IV

---

1. Interfaces
2. Classes internes et imbriquées
3. Object, clonage



# Classes imbriquées (nested classes)

---

- Classes membres statiques (static nested classes)
  - membres statiques d'une autre classe
- Classes membres ou classes internes (inner classes)
  - membres d'une classe englobante
- Classes locales
  - classes définies dans un bloc de code
- Classes anonymes
  - classes locales sans nom

# Classe imbriquée statique

---

- membre statique d'une autre classe
  - classe ou interface
  - mot clé `static`
  - similaire aux champs ou méthodes statiques: n'est pas associée à une instance et accès uniquement aux champs statiques

# Exemple

---

```
class PileChaine{
    public static interface Chainable{
        public Chainable getSuivant();
        public void setSuivant(Chainable noeud);
    }
    Chainable tete;
    public void empiler(Chainable n){
        n.setSuivant(tete);
        tete=n;
    }
    public Object depiler(){
        Chainable tmp;
        if (!estVide()){
            tmp=tete;
            tete=tete.getSuivant();
            return tmp;
        }
        else return null;
    }
    public boolean estVide(){
        return tete==null;
    }
}
```

# exemple (suite)

---

```
class EntierChainable implements PileChainee.Chainable{
    int i;
    public EntierChainable(int i){this.i=i;}
    PileChainee.Chainable next;
    public PileChainee.Chainable getSuivant(){
        return next;
    }
    public void setSuivant(PileChainee.Chainable n){
        next=n;
    }
    public int val(){return i;}
}
public static void main(String[] args) {
    PileChainee p;
    EntierChainable n;
    p=new PileChainee();
    for(int i=0; i<12;i++){
        n=new EntierChainable(i);
        p.empiler(n);
    }
    while (!p.estVide()){
        System.out.println(((EntierChainable)p.depiler()).val());
    }
}
```

# Remarques

---

- Noter l'usage du nom hiérarchique avec  
'.'  
.
- On peut utiliser un import:
  - `import PileChaine.Chainable;`
  - `import PileChaine;`

(Exercice: réécrire le programme précédent sans utiliser de classes membres statiques)

# Classe interne membre

---

- membre non statique d'une classe englobante
- peut accéder aux champs et méthodes de l'instance
- une classe interne ne peut pas avoir de membres statiques
- un objet d'une classe interne est une partie d'un objet de la classe englobante

# Exemple

---

```
class CompteBanquaire{
    private long numero;
    private long balance;
    private Action der;
    public class Action{
        private String act;
        private long montant;
        Action(String act, long montant){
            this.act=act;
            this.montant= montant;
        }
        public String toString(){
            return numero+": "+act+" "+montant;
        }
    }
    public void depot(long montant){
        balance += montant;
        der=new Action("depot",montant);
    }
    public void retrait(long montant){
        balance -= montant;
        der=new Action("retrait",montant);
    }
}
```

# Remarques

---

- numero dans toString
- this:
  - der=this.new Action(...);
  - CompteBancaire.this.numero



# Classe interne et héritage

---

```
class Externe{
    class Interne{}
}
class ExterneEtendue extends Externe{
    class InterneEtendue extends Interne{}
    Interne r=new InterneEtendue();
}
class Autre extends Externe.Interne{
    Autre(Externe r){
        r.super();
    }
}
```

(un objet Interne (ou d'une de ses extensions) n'a de sens qu'à l'intérieur d'un objet Externe)

# Quelques petits problèmes

---

```
class X{  
    int i;  
    class H extends Y{  
        void incremente(){i++;}  
    }  
}
```

Si *i* est une donnée membre de *Y*... c'est ce *i* qui est incrémenté

*X.this.i* et *this.i* lèvent cette ambiguïté.

# Suite

---

```
class H{
    void print(){}
    void print(int i){}
    class I{
        void print(){};
        void show(){
            print();
            H.this.print();
            // print(1); tous les print sont occultés
        }
    }
}
```

# Suite

---

```
public class Shadow {
    public int x = 0;
    class FirstLevel {
        public int x = 1;
        void methodInFirstLevel(int x) {
            System.out.println("x = " + x);
            System.out.println("this.x = " + this.x);
            System.out.println("Shadow.this.x = " + shadow.this.x);
        }
    }
    public static void main(String... args) {
        Shadow st = new Shadow();
        Shadow.FirstLevel fl = st.new FirstLevel();
        fl.methodInFirstLevel(23);
    }
}
---
x = 23
this.x = 1
Shadow.this.x = 0
```

# Classes locales

---

- classes définies à l'intérieur d'un bloc de code,
- une classe locale a accès aux membres de la classe englobante, et n'a accès qu'aux variables locales déclarées comme final (capture)
- comme les classes internes, elles ne peuvent pas définir des membres statiques (sauf des constantes) ou interfaces
- java 8 introduit quelques changements...
- (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;
}
```

(en java 8 il y a en plus une méthode par défaut  
forEachRemaining(Consumer<? super E>action)  
qui applique action à tous les éléments:  
while (hasNext())  
 action.accept(next());)

# 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();
}
```

---

```
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`:
  - Les objets locaux (entre autres les paramètres d'une méthode) ne sont accessibles pour une classe locale que s'ils sont constants (`final` ou java 8 effectively `final`)
  - (une classe locale a accès aux membres de la classe englobante)

# 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();
        }
    };
}
```

# Classe anonyme (java 8)

---

```
class G{int f(int i){return i;}}  
interface I{  
    int f(int j);  
    default int calcul(int i){return f(i);}  
}
```

```
——  
i=(new I(){  
    public int f(int i){return i*i;}  
    public int calcul(int i){  
        return f(i)*f(i);  
    }  
}).calcul(2); //16  
i=(new G(){int f(int i)  
    {return i*i;}}).f(3); //9
```

# 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` lance une exception `CloneNotSupportedException`
- De plus, la méthode `clone` est `protected` -> elle ne peut être utilisée que 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,  
en définissant une méthode `clone` qui lance une `CloneNotSupportedException`, le clonage n'est plus possible

# Example

---

```
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);
}
-----
d1=(k=0)(i=1,j=2) d2=(k=0)(i=1,j=2)
```

# 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.

# 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)  
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);
    }
}
```