Cours du 5 novembre

C) Méthodes: Redéfinition

- Un classe hérite des méthodes des classes ancêtres
- Elle peut ajouter de nouvelles méthodes
- Elle peut surcharger des méthodes
- Elle peut aussi redéfinir des méthodes des ancêtres.

Exemple

```
class Mere{
    void f(int i){
        System.out.println("f("+i+") de Mere");
    }
    void f(String st){
        System.out.println("f("+st+") de Mere");
    }
}
```

Exemple (suite)

```
class Fille extends Mere{
    void f(){ //surcharge
        System.out.println("f() de Fille");
    }
//  char f(int i){
// même signature mais type de retour différent
//   }
    void g(){ //nouvelle méthode
        System.out.println("g() de Fille");
        f();
        f(3);
        f("bonjour");
    }
    void f(int i){ // redéfinition
        System.out.println("f("+i+") de Fille");
    }
}
```

Exemple

```
public static void main(String[] args) {

        Mere m=new Mere();
        Fille f=new Fille();
        m.f(3);
        f.f(4);
        m=f;
        m.f(5);
        //m.g();
        ((Fille)m).g();
        f.g();
    }
```

Résultat

f(3) de Mere
f(4) de Fille
f(5) de Fille
g() de Fille
f() de Fille
f(3) de Fille
f(bonjour) de Mere
g() de Fille
f() de Fille
f(3) de Fille
f(bonjour) de Mere

D) Conséquences

- Et les variables?
  - Un principe:
    - Une méthode (re)définie dans une classe A ne peut être évaluée que dans le contexte des variables définies dans la classe A.
      - Pourquoi?

Exemple

```
class A{
    public int i=4;
    public void f(){
        System.out.println("f() de A, i="+i);
    }
    public void g(){
        System.out.println("g() de A, i="+i);
    }
}
class B extends A{
    public int i=3;
    public void f(){
        System.out.println("f() de B, i="+i);
        g();
    }
}
```

Exemple suite:

```
A a=new B();
a.f();
System.out.println("a.i="+a.i);
System.out.println("((B) a).i="+((B)a).i);
```

Donnera:

- f() de B, i=3
- g() de A, i=4
- a.i=4
- ((B) a).i=3

Remarques:

- La variable i de A est **occultée** par la variable i de B
- La variable i de A est toujours présente dans tout objet de B
- Le méthode g de A a accès à toutes les variables définies dans A (et uniquement à celles-là)
- La méthode f de B **redéfinit** f. f() redéfinie a accès à toutes les variables définies dans B

E) Divers

- super
  - Le mot clé `super` permet d'accéder aux méthodes de la super classe
    - En particulier super permet dans une méthode redéfinie d'appeler la méthode d'origine

    (exemple: super.finalize() appelé dans une méthode qui redéfinit le finalize permet d'appeler le finalize de la classe de base)

Exemple

```
class Base{
    protected String nom(){
        return "Base";
    }
}
class Derive extends Base{
    protected String nom(){
        return "Derive";
    }
    protected void print(){
        Base maref = (Base) this;
        System.out.println("this.name():"+this.nom());
        System.out.println("maref.name():"+maref.nom());
        System.out.println("super.name():"+super.nom());
    }
-------------
this.name():Derive
maref.name():Derive
super.name():Base
```

Contrôle d'accès

- `protected`: accès dans les classes dérivées
- Le contrôle d'accès ne concerne pas la signature
- Une méthode redéfinie peut changer le contrôle d'accès mais uniquement pour élargir l'accès (de `protected` à `public`)
- Le contrôle d'accès est vérifié à la compilation

Interdire la redéfinition

- Le modificateur `final` interdit la redéfinition pour une méthode
- (Bien sûr une méthode de classe ne peut pas être redéfinie! Mais, elle peut être surchargée)
- Une variable avec modificateur `final` peut être occultée

E) Constructeurs et héritage

- Le constructeurs ne sont pas des méthodes comme les autres:
  - le redéfinition n'a pas de sens.
- Appeler un constructeur dans un constructeur:
  - super() appelle le constructeur de la super classe
  - this() appelle le constructeur de la classe elle-même
  - Ces appels doivent se faire au début du code du constructeur

Constructeurs

- Principes:
  - Quand une méthode d'instance est appelée l'objet est déjà créé.
  - Création de l'objet (récursivement)
    1. Invocation du constructeur de la super classe
    2. Initialisations des champs par les initialisateurs et les blocs d'initialisation
    3. Une fois toutes ces initialisations faites, appel du corps du constructeur (super() et this() ne font pas partie du corps)

Exemple

```
class X{
    protected int xMask=0x00ff;
    protected int fullMask;
    public X(){
        fullMask = xMask;
    }
    public int mask(int orig){
    return (orig & fullMask);
    }
}
class Y extends X{
    protected int yMask = 0xff00;
    public Y(){
        fullMask |= yMask;
    }
}
```

Résultat

	xMask	yMask	fullMask
Val. par défaut des champs	0	0	0
Appel Constructeur pour Y	0	0	0
Appel Constructeur pour X	0	0	0
Initialisation champ X	0x00ff	0	0
Constructeur X	0x00FF	0	0x00FF
Initialisation champs de Y	0x00FF	0xFF00	0x00FF
Constructeur Y	0x00FF	0xFF00	0xFFFF

La classe Objet

- Toutes les classes héritent de la classe Object
- méthodes:
  - public final Class<? extends Object> **getClass**()
  - public int **hashCode**()
  - public boolean **equals**(Object obj)
  - protected Object **clone**() throws CloneNotSupportedException
  - public String **toString**()
  - protected void **finalize**() throws Throwable
  - (wait, notify,notfyall)

Exemple

```
class A{
    int i;
    int j;
    A(int i,int j){
        this.i=i;this.j=j;}
}
class D <T>{
    T i;
    D(T i){
        this.i=i;
    }
}
```

Suite

```
public static void main(String[] args) {
    A a=new A(1,2);
    A b=new A(1,2);
    A c=a;
    if (a==b)
        System.out.println("a==b");
    else
        System.out.println("a!=b");
    if (a.equals(b))
        System.out.println("a equals b");
    else
         System.out.println("a not equals b");
    System.out.println("Objet a: "+a.toString()+" classe "+a.getClass());
    System.out.println("a.hashCode()"+a.hashCode());
    System.out.println("b.hashCode()"+b.hashCode());
    System.out.println("c.hashCode()"+c.hashCode());
    D <Integer> x=new D<Integer>(10);
    System.out.println("Objet x: "+x.toString()+" classe "+x.getClass());
}
```

Résultat:

- a!=b
- a not equals b
- Objet a: A@18d107f classe class A
- a.hashCode()26022015
- b.hashCode()3541984
- c.hashCode()26022015
- Objet x: D@ad3ba4 classe class D

En redéfinissant equals

```
class B{
    int i;
    int j;
    B(int i,int j){
        this.i=i;this.j=j;
    }
    public boolean equals(Object o){
        if (o instanceof B)
            return i==((B)o).i && j==((B)o).j;
        else return false;
    }
}
```

Suite

```
B d=new B(1,2);
B e=new B(1,2);
B f=e;
if (d==e)
    System.out.println("e==d");
else
    System.out.println("d!=e");
if (d.equals(e))
    System.out.println("d equals e");
else
    System.out.println("a not equals b");
System.out.println("Objet d: "+d.toString());
System.out.println("Objet e: "+e.toString());
System.out.println("d.hashCode()"+d.hashCode());
System.out.println("e.hashCode()"+e.hashCode());
```

Résultat:

- d!=e
- d equals e
- Objet d: B@182f0db
- Objet e: B@192d342
- d.hashCode()25358555
- e.hashCode()26399554

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);

    }
}
```

suite

```
public static void main(String[] st){
   System.out.println("temps="+
            MonBenchmark.mesurer(1000000));
    }
```

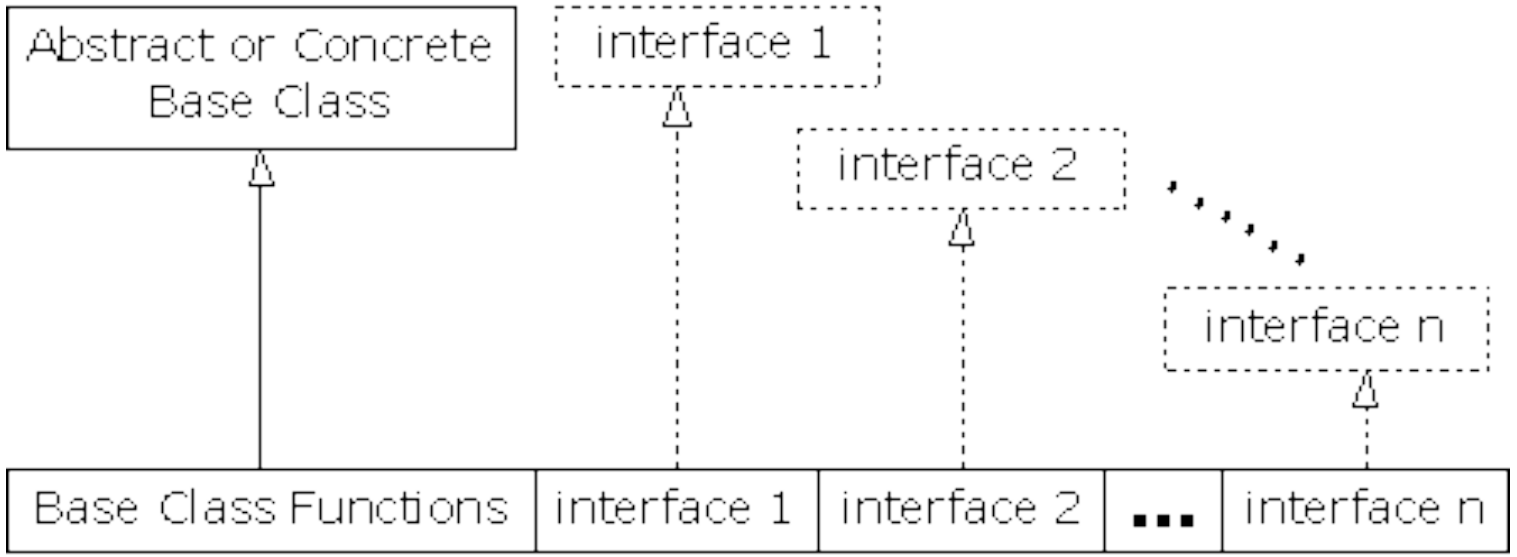
Résultat:

temps=6981893

Interfaces

- 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



Exemple:

```
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;
    }
}
```

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?)

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 `static` `public` et `final`)
  - des méthodes (elles sont implicitement `abstract`)
  - des classes internes et des interfaces