Cours 4

Héritage (suite)

B) Méthodes: Surcharge

- Méthodes et signature:
  - Signature: le nom et les arguments avec leur type (mais pas le type de la valeur retournée)
  - Seule la signature compte:
    - int f(int i)
    - char f(int i)
    - Les deux méthodes ont la même signature: c'est interdit
  - Surcharge possible:
    - Des signatures différentes pour un même nom

      int f(int i)

      int f(double f)
    - Le *compilateur* détermine par le type des arguments quelle fonction est utilisée (on verra les règles…)

Surcharge

- Un même nom de fonction pour plusieurs fonctions qui sont distinguées par leur signature

(Java, C++, Ada permettent la surcharge)

En C ' / ' est surchargé

3/2 division entière -> 1

3.0/2 division réelle -> 1,5

Surcharge

```
 public int f(int i){
        return i;
    }
//    public double f(int i){
//        return Math.sqrt( i);
//    }
    public int f(double i){
        return (int) Math.sqrt( i);
    }
    public int f(char c){
        return c;
    }
```

Remarques

- La résolution de la surcharge a lieu à la compilation
- La signature doit permettre cette résolution
- (quelques complications du fait du transtypage:
  - Exemple: un char est converti en int
  - Exemple: upcasting
)

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 d'appeler dans une méthode redéfinie 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)
- Le modificateur `final` pour une variable définit des « constantes »
  - 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