




L'héritage en Java

Les règles

- Opérateur instanceof
- Masquage des variables - Redéfinition des méthodes
- Visibilité des membres, modificateur d'accès, encapsulation

Xavier HER 1



L'héritage : qq règles

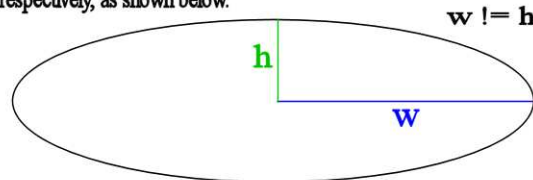
- Une classe ne peut hériter que d'une seule classe (héritage simple vs héritage multiple)
- Mot réservé extends en java
- par défaut, les classes dérivent directement ou indirectement de `java.lang.Object`
- L'opérateur *instanceof* permet de tester un objet pour connaître ce dont il hérite
 - Voir aussi l'API d'introspection (reflection)
 - `o.getClass()` fournit une référence sur l'objet correspondant à la classe dans la MV ...
- Les méthodes avec le modificateur « *final* » ne peuvent pas être redéfinies dans les sous-classes
- Une classe avec le modificateur « final » ne peut pas être dérivée

XH 2

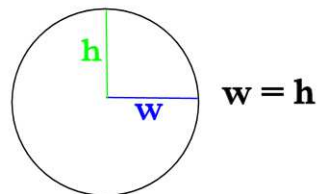


Des ellipses et des cercles

An ellipse which is situated at the origin may be uniquely specified by a tuple (w, h) , where w and h represent the half-width and half-height respectively, as shown below.



If we have the condition that $h = w$, then the ellipse is a circle.

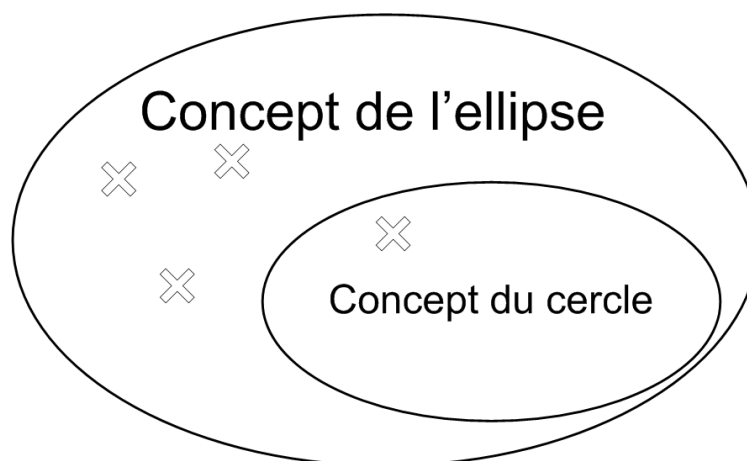


XH

3



Abstraction



XH

4

```

class Ellipse {
    public double r1, r2;
    public Ellipse(double r1, double r2) { this.r1 = r1; this.r2 = r2; }
    public double surface() { return Math.PI * r1 * r2; }
}

class Cercle extends Ellipse {
    public Cercle(double r) { super(r, r); }
    public double getRadius() { return r1; }
}

public class Heritagel {
    public static void main(String[] args) {
        Ellipse e1 = new Ellipse(2.0, 4.0);
        Cercle c1 = new Cercle(2.0);
        System.out.println("Aire de e:" + e1.surface() + ", Aire de c:" +
            c1.surface());
        System.out.println((e1 instanceof Cercle)); // false
        System.out.println((e1 instanceof Ellipse)); // true
        System.out.println((c1 instanceof Cercle)); // true
        System.out.println((c1 instanceof Ellipse)); // true !!!

        e1 = c1; // up-casting Barbara Liskov
        System.out.println((e1 instanceof Cercle)); // true
        System.out.println((e1 instanceof Ellipse)); // true
        //int r = e1.getRadius(); // Error: method getRadius not found

        //c1 = e1; // Error: Incompatible type for =. Explicit cast
        //needed.
        c1 = (Cercle)e1; //down-casting
    }
}

```

Opérateur instanceof

5

Résumé

- Opérateur instanceof
 - marche aussi avec les interfaces implémentées
- Transtypage ascendant
 - Référence d'un type remontant dans la hiérarchie d'héritage
 - Affectation sans avoir besoin de cast
 - Pas d'accès aux méthodes définies plus bas
 - Voir pour un point d'accès aux méthodes définies dans les classes filles, même abstract
 - getRadius ?
- Transtypage descendant la hiérarchie d'héritage
 - Cast explicite
 - risque de ClassCastException (point « chaud » du code)

XH
6

```

class A {
    int x;
    void m() {int A=0,B=0; A=B;}

class B extends A{
    int x;
    void m() {int A=0,B=0; A=B;}

class C extends B {
    int x, a;
    void m() {int A=0,B=0; A=B;}
    void test(){
        a = super.x; // a reçoit la valeur de la variable x de la
        // classe B
        a = ((B)this).x; // a reçoit la valeur de la variable x
        // de la classe B
        //a = super.super.x; // Syntax error
        a = ((A)this).x; // a reçoit la valeur de la variable x
        // de la classe A
        super.m(); // Appel à la méthode m de la classe B
        //super.super.m(); // Syntax error
        ((B)this).m(); // Appel à la méthode m de la classe C
        !!! (et non B)
    }
    public static void main ( String [] args) {
        C obj = new C();obj.test();
    }
}

```

Masquage de variable - Redéfinition de méthode

```

classDiagram
    class A {
        x: Integer
        m()
    }
    class B {
        x: Integer
        m()
    }
    class C {
        x: Integer
        a: Integer
        m()
        test()
        $main()
    }
    A <|-- B
    B <|-- C

```

XH
7

■ FAIRE UN DESSIN DE L'OBJET

XH
8



Masquage de variable et redéfinition de méthode

- Masquage de variable
 - une classe peut définir des variables portant le même nom que celles de ses classes ancêtres
 - Une classe peut accéder aux attributs redéfinis de sa classe mère en utilisant « *super.* » ou par *cast*
- Redéfinition de méthode
 - Une classe peut accéder aux méthodes redéfinies de sa classe mère en utilisant « *super.* »
 - Autrement, les méthodes sont virtuelles en Java

XH

9



Masquage de variable vs redéfinition de méthode

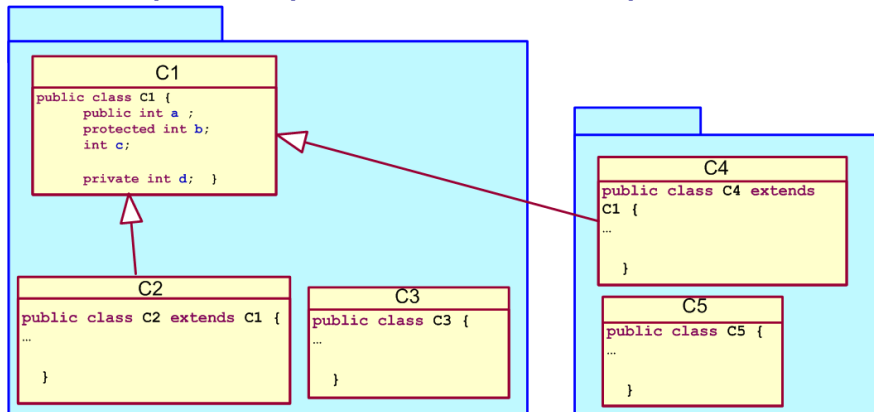
- En Java, les méthodes sont toujours virtuelles
 - Modificateur « *virtual* » nécessaire en C++
 - dynamic binding vs static binding(A lire sur [Stackoverflow.com](https://stackoverflow.com))
 - Ligature dynamique
- Les champs ne sont pas virtuelles

XH

10



Les modificateurs d'accès: public, protected, "rien", private



	a (public)	b (protected)	c	d (private)
accessible par C2	Oui	Oui	Oui	-
accessible par C3	Oui	Oui	Oui	-
accessible par C4	Oui	Oui	-	-
accessible par C5	Oui	-	-	-

XH



11



Visibilité des membres d'une classe

- Visibilité des membres d'une classe avec les modificateurs d'accès:
 - public, protected, "rien", private
 - (du plus permissif au plus restrictif)
- Encapsulation des champs avec un modificateur restrictif
- « rien » veut dire « package access » (friendly)
- protected veut dire package access + accessible par des classes filles (d'un autre package)
- Pour plus d'informations sur la visibilité des méthodes, des classes, voir le pdf de Bougeault

XH

12