

# Cours 3

Héritage (début)

# Passage par valeur

---

```
public class ParamParVal {  
    public static void parVal(int i){  
        i=0;  
        System.out.println("dans parVal i="+i);  
    }  
}  
//...  
int i =100;  
System.out.println("Avant i="+i);  
ParamParVal.parVal(i);  
System.out.println("Avant i="+i);
```

-----  
Avant i=100  
dans parVal i=0  
Avant i=100

# Mais...

---

- Comme les variables sont de références (sauf les types primitifs)...

```
public static void bidon(Astre a){  
    a=new Astre("bidon", null);  
    System.out.println("bidon a="+a);  
}  
public static void bidonbis(Astre a){  
    a.setNom("bidon");  
    a.setOrbite(null);  
    System.out.println("bidonbis a="+a);  
}
```

# Méthodes...

---

## □ Contrôler l'accès:

//...

```
public void setNom(String n){
    nom=n;
}
public void setOrbite(Astre a){
    orbite=a;
}
public String getNom(){
    return nom;
}
public Astre getOrbite(){
    return orbite;
}
```

# Méthodes, remplacement...

---

```
public String toString(){  
    String st=idNum + "("+nom+")";  
    if (orbite != null)  
        st += "en orbite "+ orbite;  
    return st;  
}
```

Remplace la méthode toString de la classe Object

# Nombre variable d'arguments...

---

```
public static void affiche(String ... list){  
    for(int i=0;i<list.length;i++)  
        System.out.print(list[i]+" ");  
}
```

```
//...
```

```
affiche("un", "deux","trois");
```

# Méthodes main

---

```
public static void main(String[] args) {  
    for(int j =0; j<args.length;j++){  
        System.out.print(args[j] + " ");  
    }  
}
```

Le main est le point d'accès et peut avoir des arguments:

# VI) exemple: Les astres...

---

```
package exempleClasses;
```

```
/**
 *
 * @author sans
 */
public class Astre {
    private long idNum;
    private String nom = "<pasdenom>";
    private Astre orbite = null;
    private static long nextId = 0;
    /** Creates a new instance of Astre */
    private Astre() {
        idNum = nextId ++;
    }
}
```

□



# Suite

---

```
public Astre(String nom, Astre enOrbite){
    this();
    this.nom=nom;
    orbite=enOrbite;
}
public Astre(String nom){
    this(nom,null);
}
public Astre(Astre a){
    idNum = a.idNum;
    nom=a.nom;
    orbite=a.orbite;
}//...
```

---

```
public void setNom(String n){
    nom=n;
}
public void setOrbite(Astre a){
    orbite=a;
}
public String getNom(){
    return nom;
}
public Astre getOrbite(){
    return orbite;
}
public String toString(){
    String st=idNum + "("+nom+")";
    if (orbite != null)
        st += "en orbite "+ orbite;
    return st;
}
```

# Chapitre III

## Héritage

---

# Chapitre III: Héritage

---

- A) Extensions généralités
  - Affectation et transtypage
- B) Méthodes
  - Surcharge et signature
- C) Méthodes (suite)
  - Redéfinition et liaison dynamique
- D) Conséquences
  - Les variables
- E) Divers
  - Super, accès, final
- F) Constructeurs et héritage

# A) Extension: généralités

---

- Principe de la programmation objet:
  - un berger allemand est un chien
    - il a donc toutes les caractéristiques des chiens
    - il peut avoir des propriétés supplémentaires
    - un chien est lui-même un mammifère qui est lui-même un animal: hiérarchie des classes
  - On en déduit:
    - Hiérarchie des classes (Object à la racine)
    - et si B est une extension de A alors un objet de B est un objet de A avec des propriétés supplémentaires

# Extension: généralités

---

Quand B est une extension de la classe A:

- Tout objet de B a toutes les propriétés d'un objet de A (+ d'autres).
- Donc un objet B peut être considéré comme un objet A.
- Donc les variables définies pour un objet de A sont aussi présentes pour un objet de B (+ d'autres). (Mais elles peuvent être *occultées*)
- Idem pour les méthodes : Les méthodes de A sont présentes pour B et un objet B peut définir de nouvelles méthodes.
- Mais B peut *redéfinir* des méthodes de A.

# Extension de classe

---

Si B est une extension de A

□ pour les variables:

- B peut ajouter des variables (et si le nom est identique cela *occultera* la variable de même nom dans A)

(occulter = continuer à exister mais "caché")

- *Les variables de A sont toutes présentes pour un objet B, mais certaines peuvent être cachées*

□ pour les méthodes

- B peut ajouter de nouvelles méthodes
- B peut *redéfinir* des méthodes (même signature)

# Remarques:

---

- pour les variables
  - c'est le nom de la variable qui est pris en compte (pas le type).
  - dans un contexte donné, à chaque nom de variable ne correspond qu'une seule déclaration.
  - (l'association entre le nom de la variable et sa déclaration est faite à la compilation)
- pour les méthodes
  - c'est la signature (nom + type des paramètres) qui est prise en compte:
    - on peut avoir des méthodes de même nom et de signatures différentes (surcharger)
    - dans un contexte donné, à un nom de méthode et à une signature correspond une seule déclaration
    - (l'association entre le nom de la méthode et sa déclaration est faite à la compilation, mais l'association entre le nom de la méthode et sa définition sera faite à l'exécution)



# Extension (plus précisément)

---

- Si B est une extension de A  
(`class B extends A`)
  - Les variables et méthodes de A sont des méthodes de B (mais elles peuvent ne pas être accessibles: `private`)
  - B peut ajouter de nouvelles variables (si le *nom* est identique il y a occultation)
  - B peut ajouter de nouvelles méthodes si la *signature* est différente
  - B redéfinit des méthodes de A si la signature est identique

# Remarques:

---

- Java est un langage typé
  - en particulier chaque variable a un type: celui de sa déclaration
  - à la compilation, la vérification du typage ne peut se faire que d'après les déclarations (implicites ou explicites)
  - le compilateur doit vérifier la légalité des appels des méthodes et des accès aux variables:
    - `a.f()` est légal si pour le type de la variable `a` il existe une méthode `f()` qui peut s'appliquer à un objet de ce type
    - `a.m` est légal si pour le type de la variable `a` il existe une variable `m` qui peut s'appliquer à un objet de ce type

# En conséquence:

---

- Une variable *déclarée* comme étant de classe A peut référencer un objet de classe B ou plus généralement un objet d'une classe dérivée de A:
  - un tel objet contient tout ce qu'il faut pour être un objet de classe A
- Par contre une variable *déclarée* de classe B ne peut référencer un objet de classe A: il manque quelque chose!

# Affectation downcast/upcast

---

```
class A{
    public int i;
    //...
}
class B extends A{
    public int j;
    //...
}
public class Affecter{
    static void essai(){
        A a = new A();
        B b = new B();
        //b=a; impossible que signifierait b.j??
        a=b; // a référence un objet B
        // b=a;
        b=(B)a; // comme a est un objet B ok!!
    }
}
```

# Upcasting

---

- Si B est une extension de A, alors un objet de B peut être considéré comme un objet de A:
  - A a=new B();
- On pourrait aussi écrire:
  - A a=(A) new B();
- l'upcasting permet de considérer un objet d'une classe dérivée comme un objet d'une classe de base
- Upcasting: de spécifique vers moins spécifique (vers le haut dans la hiérarchie des classes)
- l'upcasting peut être implicite (il est sans risque!)
- attention
  - il ne s'agit pas réellement d'une conversion: l'objet n'est pas modifié

# Downcasting

- Si B est une extension de A, il est possible qu'un objet de A soit en fait un objet de B. Dans ce cas on peut vouloir le considérer un objet de B
  - A a=new B();
  - B b=(B)a;
- Il faut dans ce cas un cast (transtypage) *explicite* (la "conversion" n'est pas toujours possible -l'objet référencé peut ne pas être d'un type dérivé de B)
- A l'exécution, on vérifiera que le cast est possible et que l'objet considéré est bien d'un type dérivé de B
- downcasting: affirme que l'objet considéré est d'un type plus spécifique que le type correspondant à sa déclaration (vers le bas dans la hiérarchie des classes)
- le downcasting ne peut pas être implicite (il n'est pas toujours possible!)
- attention
  - il ne s'agit pas réellement d'une conversion: l'objet n'est pas modifié

# Casting

---

- On peut tester la classe avant un "downcast":

```
Base sref;  
Derive dref;  
if(sref instanceof Derive)  
    dref=(Derive) sref
```

# 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

---

- Une 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");
    }
}
```

# Example

---

```
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)

# Example

---

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

```
-----
this.name():Derive
mref.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)

# Example

---

```
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 Object

---

- 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, notifyall)

# Example

---

```
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