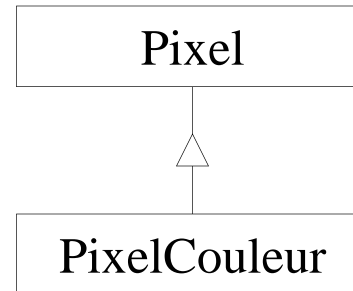
L’héritage

Principe

L'héritage : principe

- L'héritage permet de créer facilement des nouvelles classes à partir de classes existantes :

- Pixel est la classe de base.




```
class Pixel {
      private int x ;
      private int y ;

      public Pixel( int x, int y ) {
            // …
      }
      public void affiche() {
            // …
      }
}
```

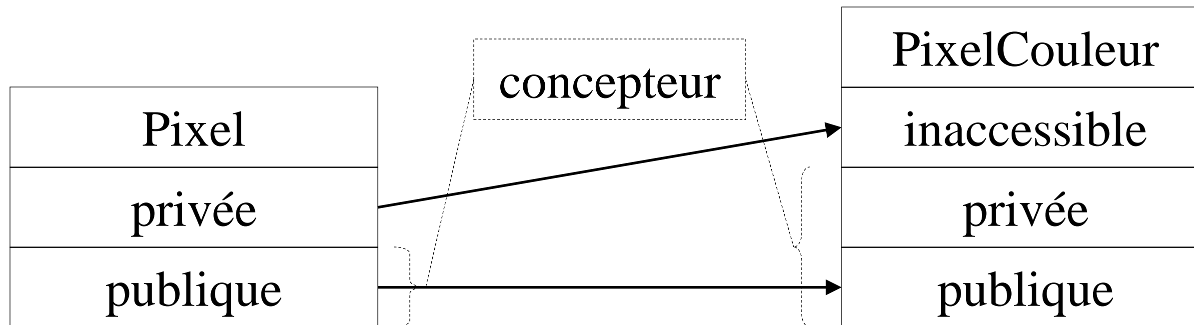
```
class PixelCouleur extends Pixel {
      private int couleur ;

      public PixelCouleur( int x, int y,  int couleur ) {
            super(x, y);
            this.couleur = couleur;
      }
      public static void main( String [] argv ) {
            PixelCouleur p = new PixelCouleur( 10, 20, 255 ) ;
            p.affiche() ;          // affiche de Pixel
      }
}
```

- PixelCouleur qui hérite de Pixel est la classe dérivée.

Accès aux données membres de la classe de base : mode protégé

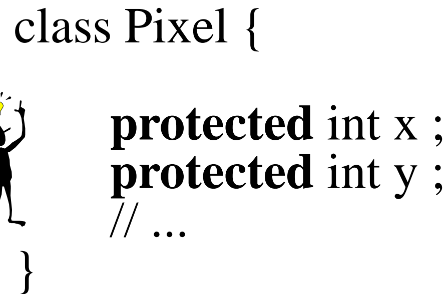
```
class Pixel {
	private int x ;
	private int y ;

	public Pixel( int x, int y ){
		// …
	}
	public void affiche() {
		// …
	}
}
```

```
class PixelCouleur extends Pixel {
	private int couleur ;

	public PixelCouleur( int couleur ) {
		// ...
	}
	public void deplacer( int deplacX, int deplacY ) {
		x = x + deplacX ;   // erreur à la compilation
		y = y + deplacY ;   // x et y inaccessibles
	}
}
```

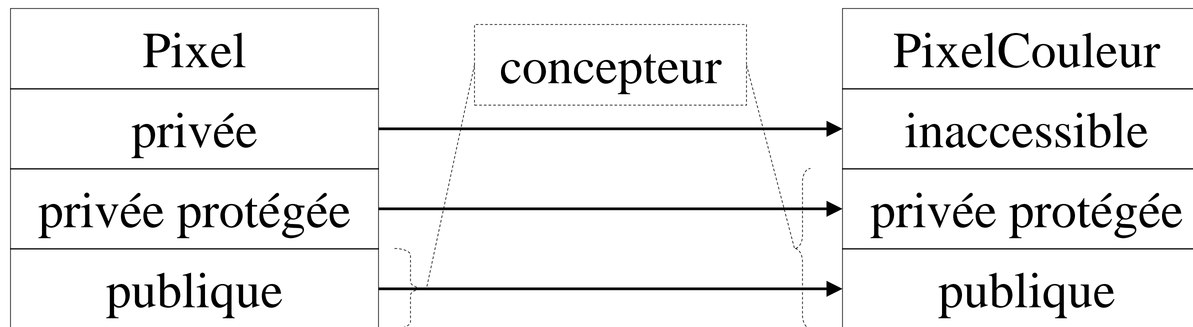


Le concepteur de PixelCouleur n’a pas accès aux données privées de Pixel

Accès aux données membres de la classe de base : mode protégé



```
class Pixel {
    protected int x ;
    protected int y ;
    // ...
}
```

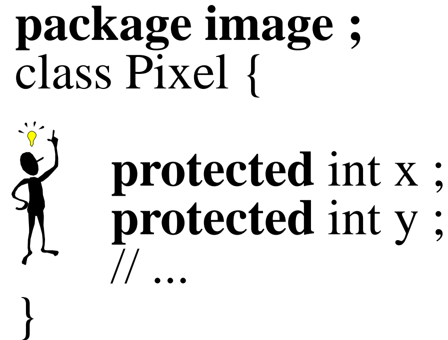



```
class PixelCouleur extends Pixel {
    // ...

    public void deplacer( int deplacX, int deplacY ){
        x = x + deplacX ;   //          OK
        y = y + deplacY ;   //          OK
    }
}
```

Pixel et PixelCouleur sont dans le même package ou dans des packages différents.

Accès aux données membres de la classe de base : mode protégé

```
package image ;
class Pixel {
        protected int x ;
        protected int y ;
        // ...
}
```



• Un membre protégé est accessible dans une classe fille du même package ou d'un package différent :

```
package image.couleur ;
class PixelCouleur extends Pixel {
      // ...

      public void deplacer( int deplacX, int deplacY ){
            x = x + deplacX ;   //           OK
            y = y + deplacY ;   //           OK
      }
}
```

• Un membre protégé est accessible dans toute les classes du même package :

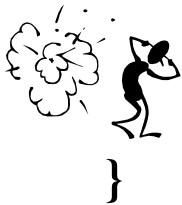
```
package image ;
class Television {

      public void allumer(){
            Pixel p( 0, 0 ) ;
            p.x = 1 ;  // OK
      }
}
```

Les limites du mode protégé

• Impossible d'accéder à un membre protégé dans une classe dérivée via une référence sur une classe de base si les deux classes sont dans des packages différents :

```
package image ;
class Pixel {

        protected int x ;
        protected int y ;
        // ...
}
```

```
package image.couleur ;
class PixelCouleur extends Pixel {

        public static void main( String argv[] ){
                Pixel p = new Pixel() ;
                p.x ;        // ______ erreur
        }
}
```



Le polymorphisme

Redéfinition des fonctions membres

- Quand une fonction est redéfinie dans une classe dérivée => la fonction de la classe de base est masquée :

```
class Pixel {
    // …

    public void affiche() {
        // …
    }

    public static void main( String argv[] ){
        Pixel p = new Pixel() ;
        p.affiche() ;
    }
}
```

Appel de affiche de Pixel.

→ redéfinition →

```
class PixelCouleur extends Pixel {
    // …

    public void affiche() {
        // …
    }

    public static void main( String argv[] ){
        PixelCouleur pc=newPixelCouleur();
        pc.affiche() ;
    }
}
```

Appel de affiche de PixelCouleur.

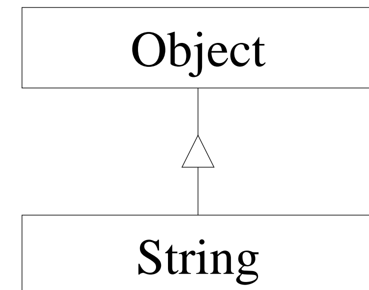
Compatibilité entre objets dérivés et objets de la classe de base

• En java, les transtypages (cast) sont vérifiés au moment de la compilation et au moment de l'exécution : l'exception ClassCastException est levée si le transtypage est illégal.

```
PixelCouleur pixelCouleur = new PixelCouleur() ;
Pixel pixel = pixelCouleur ;              // conversion type dérivé vers type de base
pixelCouleur = pixel ;                    // erreur à la compilation
pixelCouleur = (PixelCouleur)pixel;       // conversion type de base vers type dérivé
```

• Exemple avec la classe Vector (java.util) :

```
Vector phrases = new Vector() ;
String phrase = "Bonjour" ;
phrases.addElement( phrase ) ;

phrase = phrases.elementAt( 0 ) ;
// éviter les erreurs à l'exécution :
Object o = phrases.elementAt( 0 ) ;
if( o instanceof String ){
        String s = (String)o ;
}
```



Toutes les classes Java hérite de la classe Object

Le polymorphisme

• Le polymorphisme consiste à changer le comportement d'une classe grâce à un héritage tout en continuant à l'utiliser comme une classe de base.

```
class Pixel {
      // ...

      public Pixel(){
            // …
      }
      public void affiche() {
            // …
      }
}
```

```
class PixelCouleur extends Pixel {
      private int couleur ;

      public PixelCouleur() {
            // ...
      }
      public void affiche() {
            // …
      }
}
```

```
Pixel pixel = new Pixel() ;
pixel.affiche()                    // appel de affiche de Pixel

PixelCouleur pixelCouleur = new PixelCouleur() ;
pixel = pixelCouleur ;
pixel.affiche()  ;                 // appel de affiche de PixelCouleur
```

Intérêt du polymorphisme

FigureGéom
tracer()

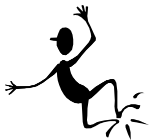
Polygone	Ellipse
tracer()	tracer()

Parallélogramme	Triangle
tracer()	tracer()

L'ajout d'une classe modifie

```
Parallelogramme para = new Parallelogramme() ;
Ihm ihm = new Ihm() ;
ihm.dessiner( para ) ;
```

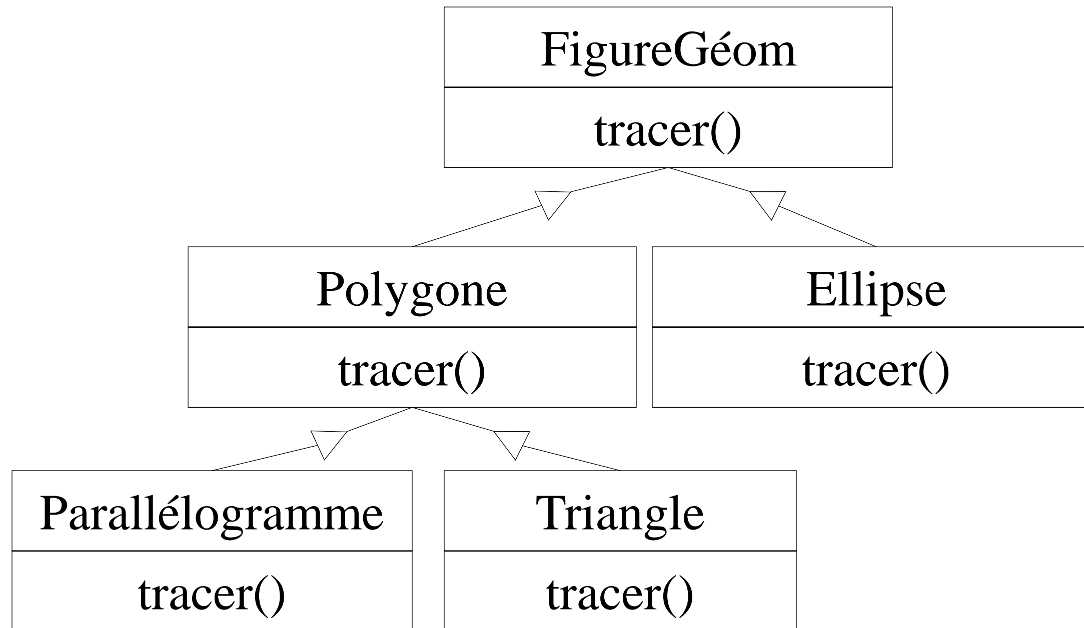
```
class Ihm {
	// …

	public void dessiner( FigureGeom fig ) {
		fig.tracer() ;
	}
}
```



On peut ajouter des classes sans modifier le code :

Les méthodes abstraites

Comment écrire le corps de tracer() de FigureGeom ?



Une fonction abstraite est une fonction qui n'est pas définie :

```
abstract class FigureGeom {
    // ...
    abstract public void tracer() ;
}
```

Une classe contenant une méthode abstraite doit être déclarée abstraite.

Les classes abstraites, les interfaces

Les classes abstraites

Si une classe contient au moins une fonction abstraite :

- c'est une classe abstraite ;
- elle ne peut pas être instanciée :

```
FigureGeom fig = new FigureGeom() ;        // erreur à la compilation
FigureGeom fig ;                           // OK
```

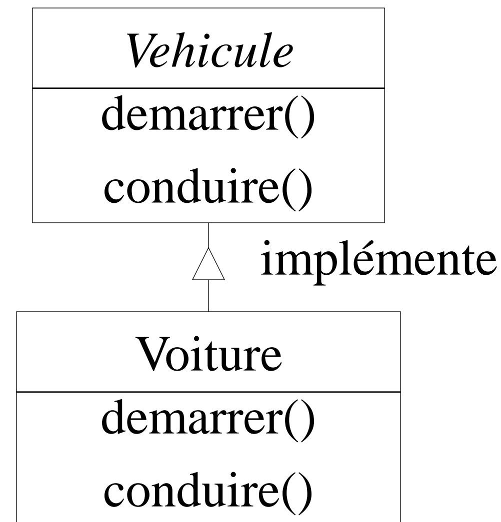
- elle ne peut être utilisée que par héritage.

Les interfaces

- Une interface est une classe qui n'a que des méthodes abstraites. On peut alors lui associer n'importe quelle implémentation.

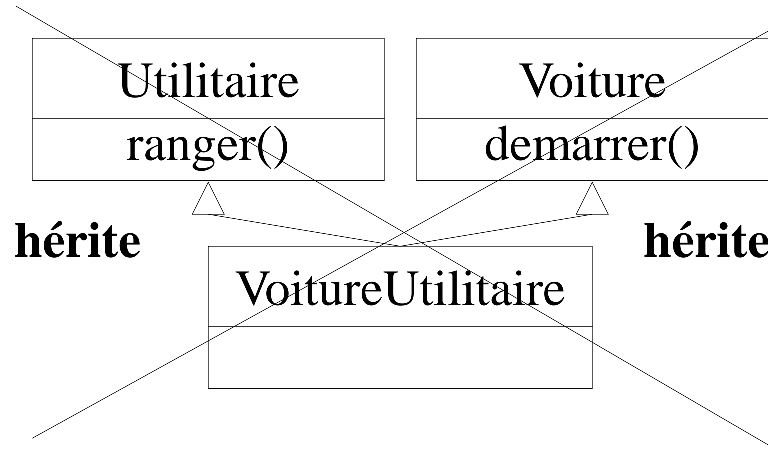
```
interface Vehicule {

        public boolean demarrer() ;
        public void conduire() ;
}

// implémentation possible
class Voiture implements Vehicule {
        public boolean demarrer() {
                // …
        }
        public void conduire() {
                // …
        }
}
```

- les champs d'une interface sont par défaut static et final ;
- les méthodes d'une interface sont publiques.



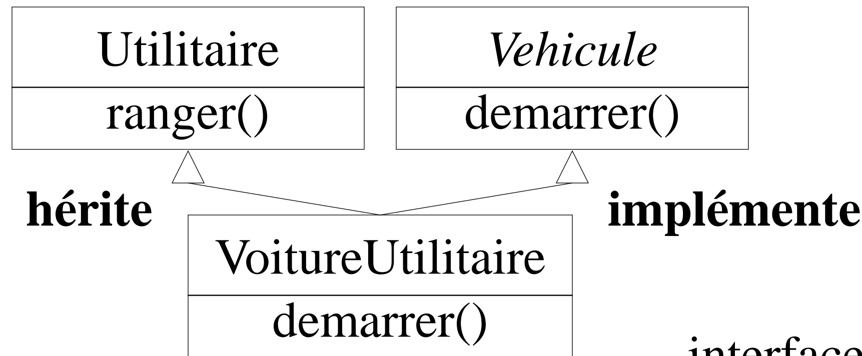
L'héritage multiple

- Une classe ne peut pas hériter de plusieurs classes :




```
class VoitureUtilitaire extends Utilitaire, Voiture {          // erreur à la compilation
    // ...
}
```

Les interfaces et l'héritage multiple : 1ére solution

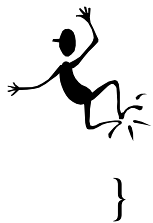



```
class Utilitaire {
    // ...
    public void ranger() ;
}
```

```
interface Vehicule {
    // ...
    public boolean demarrer() ;
}
```

- Une classe peut hériter d'une seule classe mais peut implémenter plusieurs interfaces :

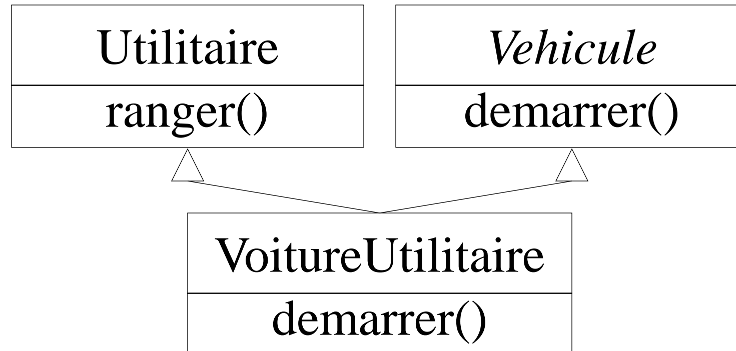
```
class VoitureUtilitaire extends Utilitaire implements Vehicule {

    public boolean demarrer() {
        // ...
    }
}
```



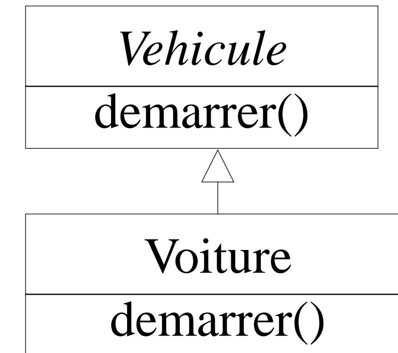
Les interfaces et l'héritage multiple : 2e solution

- la re-direction des méthodes permet de changer rapidement d'implémentation

```
interface Vehicule {
    // ...
    public boolean demarrer() ;
}
```




```
class Voiture implements Vehicule {
    public boolean demarrer() {
        // …
    }
}
```

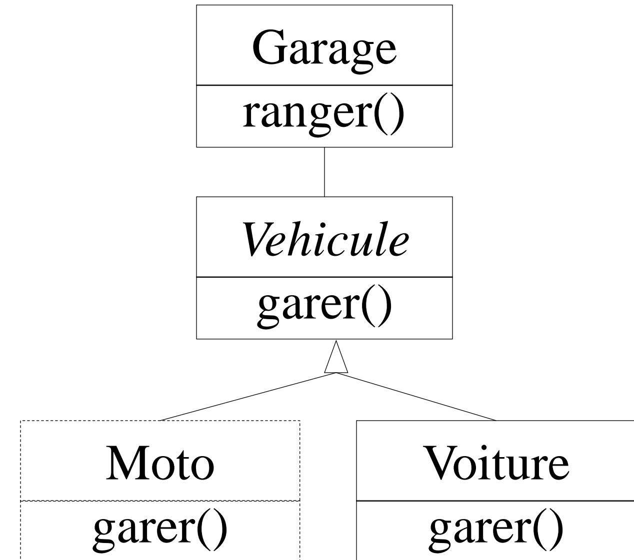



```
class VoitureUtilitaire extends Utilitaire implements Vehicule {
    Voiture voiture = new Voiture() ;

    public boolean demarrer() {
        voiture.demarrer() ;          // re-direction
    }
}
```

Les interfaces comme fonctions réflexes

```
class Garage {
        Vehicule vehicule ;
        public void setVehicule( Vehicule véhicule )
        {
                this.vehicule = vehicule ;
                vehicule.garage = this ;
        }
        public void ranger() {
                vehicule.garer() ;
        }
}
```

```
class Voiture implements Vehicule {
        // ...
}
```



Appels générique

```
Garage garage = new Garage() ;
Voiture voiture = new Voiture() ;
garage.setVehicule( voiture ) ;
garage.ranger() ;

Moto moto = new Moto() ;
garage.setVehicule( moto ) ;
garage.ranger() ;
```

Conclusion sur les interfaces

- Les interfaces permettent l'héritage multiple, mais elles ne disposent que de champs static et final ;
- Une interface doit être déclarée publique pour pouvoir être utilisée à l'extérieur du package où elle a été écrite ;
- Une interface peut étendre une autre interface :
- Une classe abstraite peut avoir une implémentation partielle.

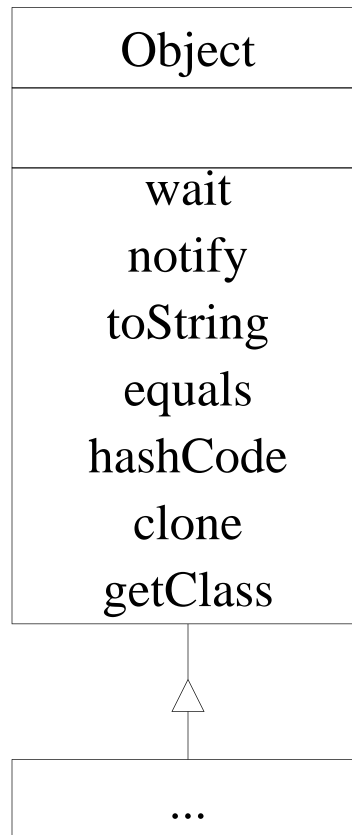
Conseil : toute classe, abstraite ou pas, prévue pour être étendue devrait implémenter une interface. Cela permet :

- de réaliser des héritage multiple basés sur cette classe, en utilisant éventuellement la re-direction ;
- d'implémenter la classe différemment si celle déjà écrite ne convient pas.

La classe Object :
mère de toute les classes

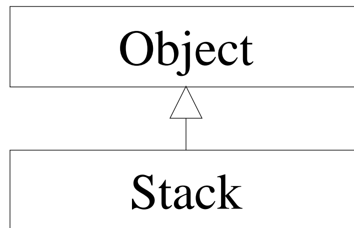
La classe Object

La classe Object est la classe mère de toutes les classes :



Intérêt de la classe Object

- Une référence de type Object peut référencer n'importe quel objet.



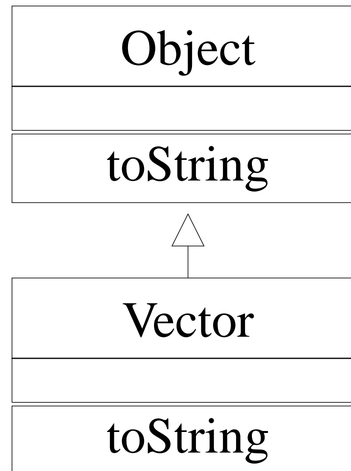

```
package java.util ;
public class Stack … {
        public Object push(Object item) { … }
        public synchronized Object pop() { … }
}
```

Exemple d'une pile de String :

```
Stack stack = new Stack() ;              // créé une pile
stack.push( "chaine" ) ;                 // pousse "chaine" sur la pile
stack.push( "autre chaine" ) ;
String s = (String)stack.pop() ;         // extrait "autre chaine" de la pile
```

La méthode toString de la classe Object

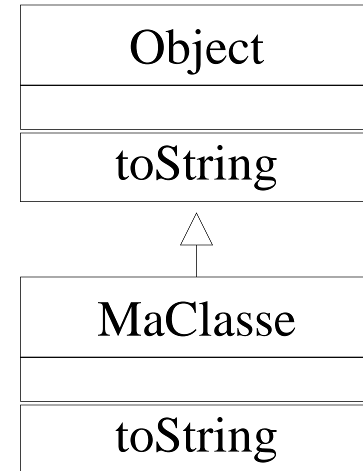
- La méthode toString peut être utilisée pour lire l'état d'un objet :




```
Vector v = new Vector() ;
String s = "chaîne" ;
v.addElement( s ) ;
Object o = new Object() ;
System.out.println( "v = " + v + ", o = " + o ) ;        // appel de toString pour v et pour o

// à l'écran : v = [chaine], o = java.lang.Object@1e19ac
```

La méthode toString de la classe Object

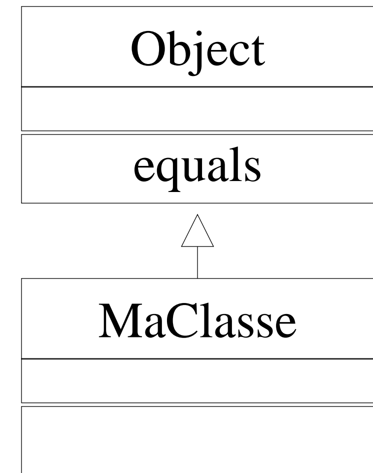
- La méthode toString peut être redéfinie :




```
class MaClasse {
    private int i ;
    private String s ;

    public MaClasse( int i, String s ) {
        this.i = i ;
        this.s = s ;
    }
    public String toString() {
        String desc = i + "     " + s ;
        return desc ;
    }
    public static void main( String [] argv ) {
        MaClasse monObjet = new MaClasse( 1, "azerty" ) ;
        System.out.println( "monObjet = " + monObjet ) ;  // appel de toString
    }
}
```

La méthode equals de la classe Object

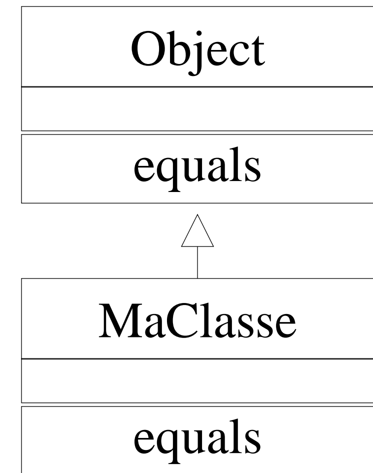
- Deux objets ayant le même état, ne sont pas identiques au sens de equals de Object.




```
class MaClasse{
    private String s ;

    public MaClasse( String s ) {
        this.s = s ;
    }
    public static void main( String [] argv ){
        MaClasse objet1 = new MaClasse( "chaine" ) ;
        MaClasse objet2 = new MaClasse( "chaine" ) ; // o1 et o2 ont le même état
        if( objet1.equals( objet2 ) )
            System.out.println( "equals" ) ;
        else
            System.out.println( "non equals" )          // affichage : non equals
    }
}
```

La méthode equals de la classe Object

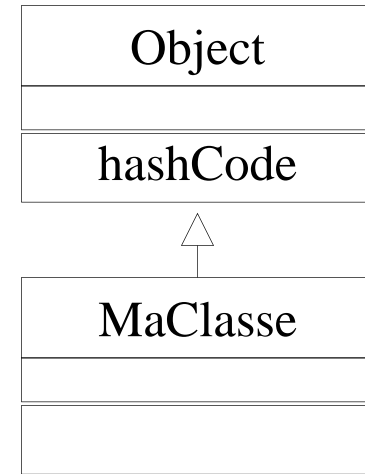
- Redéfinir la méthode equals pour que deux objets de même état soient égaux :




```
class MaClasse{
    private String s ;

    public MaClasse( String s ) {
        this.s = s ;
    }
    public boolean equals( Object obj ){
        if( (obj!=null) && (obj instanceof MaClasse) )
            return s.equals( ((MaClasse)obj).s ) ;
        else return false ;
    }
    public static void main( String [] argv ){
        MaClasse o1 = new MaClasse( "chaine" ) ;
        MaClasse o2 = new MaClasse( "chaine" ) ;      // o1 et o2 ont le même état
        if( objet1.equals( objet2 ) )
            System.out.println( "equals" ) ;          // affichage : equals
    }
}
```

La méthode hashCode de la classe Object

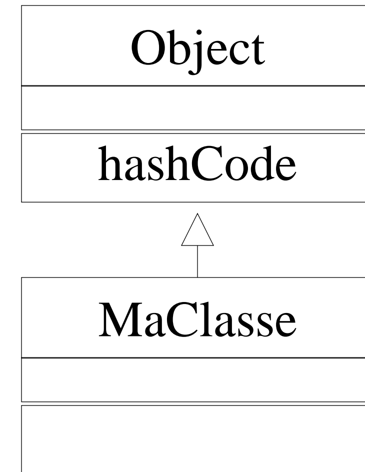
- La classe Hashtable utilise un code de hachage pour ranger des objets dans une table de hachage :




```
class MaClasse{
    private String s ;

    public MaClasse( String s ) {
        this.s = s ;
    }
    public static void main( String [] argv ){
        MaClasse o1 = new MaClasse( "chaine" ) ;
        Hashtable<MaClasse,Date> dates = new Hashtable<MaClasse,Date> () ;
        dates.put( o1, new Date( "27 Aug 1966" )) ;
        dates.put( o1, new Date( "3 Oct 178956" )) ;
        System.out.println( dates.size() ) ;      // 1 (une seule clef unique par table)
        Date d = dates.get( o1 ) ;
        System.out.println( d ) ;                 // Fri 3 Oct … 178956
    }
}
```

La méthode hashCode de la classe Object

- 2 objets ayant le même état sont considérés comme 2 clefs différentes pour une table de hachage :

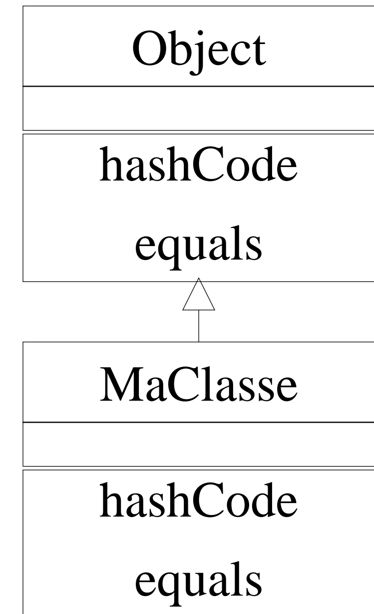



```
class MaClasse{
	private String s ;

	public MaClasse( String s ) {
		this.s = s ;
	}
	public static void main( String [] argv ){
		Hashtable<MaClasse,Date> dates = new Hashtable<MaClasse,Date> () ;
		MaClasse o1 = new MaClasse(  "mon anniversaire" ) ;
		dates.put( o1, new Date( "27 Aug 1966" )) ;

		MaClasse o2 = new MaClasse(  "mon anniversaire" ) ;    // o1 et o2 : même état
		dates.put( o2, new Date( "3 Oct 178956" )) ;
		System.out.println( dates.size() ) ;                    // 2 éléments dans la table
	}
}
```

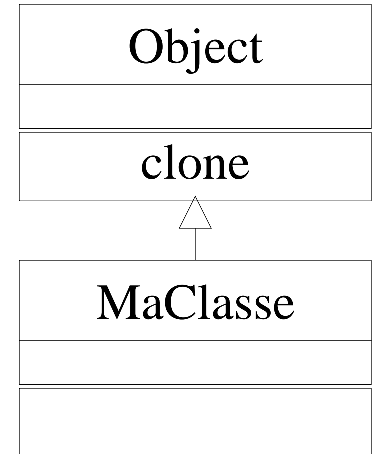
La méthode hashCode de la classe Object

- Redéfinir equals et hashCode pour que 2 objets ayant le même état soient considérés comme 1 seule clef pour une table de hachage :

```
class MaClasse{
    private String s ;

    public MaClasse( String s ) {
        this.s = s ;
    }
    public boolean equals( Object obj ){
        if( (obj!=null) && (obj instanceof MaClasse) )
            return s.equals( ((MaClasse)obj).s ) ;
        else return false ;
    }
    public int hashCode() {
        return s.hashCode() ;
    }
}
```



La méthode clone de la classe Object

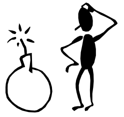


- La duplication d'un objet peut être obtenue avec la méthode clone :

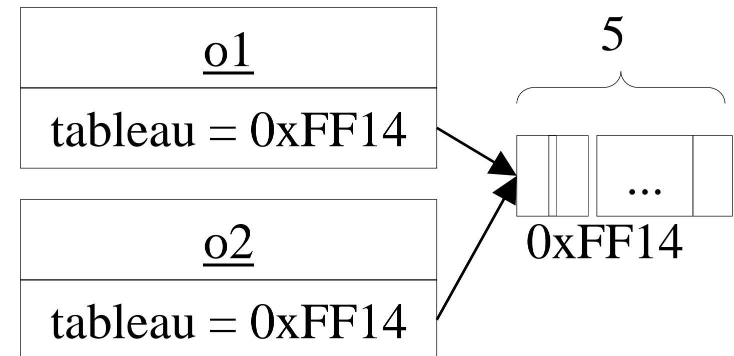
```
class MaClasse implements Cloneable {
        private int i = 0 ;

        public static void main( String [] argv ) {
        MaClasse o = new MaClasse() ;
        try{
                MaClasse o2 = (MaClasse)o1.clone() ;        // o2 est un clone de o1
                o2.i = 1 ;                                  // o1.1 n'est pas modifié
        } catch( CloneNotSupportedException e ) {
                System.out.println( e ) ;
        }
        System.out.println( o1.i ) ;                        // o1.i n'a pas changé
}
```

La méthode clone de Object lance systématiquement l'exception CloneNotSupportedException si la classe n'implémente pas l'interface Cloneable.

La méthode clone de la classe Object



- La méthode clone de objet ne peut pas être utilisée pour cloner des objets ayant des références :

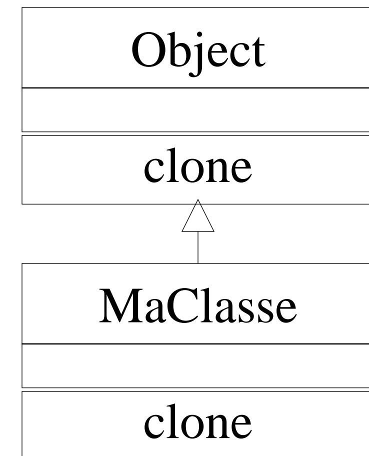



```
class MaClasse implements Cloneable{
    private int [] tableau = new int[ 5 ] ;

    public static void main( String [] argv ){
        MaClasse o1 = new MaClasse() ;
        o1.tableau[ 0 ] = -1 ;
        System.out.println( o1.tableau[ 0 ] ) ;          // -1
        try{
            MaClasse o2 = (MaClasse)o1.clone() ;     // o2 est une clone de o1
            o2.tableau[ 0 ] = 0 ;                    // le tableau est commun !
        } catch( CloneNotSupportedException e ) {
            System.out.println( e ) ;
        }
        System.out.println( o1.tableau[ 0 ] ) ;          // 0 changement !!
}
```

La méthode clone de la classe Object

- Pour cloner un objet ayant des références, il faut redéfinir la méthode clone :

```
class MaClasse implements Cloneable{
    private int [] tableau = new int[ 5 ] ;
    public Object clone() throws CloneNotSupportedException{
        MaClasse obj = (MaClasse)super.clone() ;
        obj.tableau = (int[])tableau.clone() ;
        return obj ;
    }
    public static void main( String [] argv ){
        MaClasse o1 = new MaClasse() ;
        o1.tableau[ 0 ] = -1 ;
        System.out.println( o1.tableau[ 0 ] ) ;
        try{
            MaClasse o2 = (MaClasse)o1.clone() ;    // clonage OK
            o2.tableau[ 0 ] = 0 ;                   // seul o2 est modifié
        } catch( CloneNotSupportedException e ) {
            System.out.println( e ) ;
        }
        System.out.println( o1.tableau[ 0 ] ) ;     // -1 : pas de changement
}
```



La méthode clone de la classe Object

- Pour cloner un objet ayant des références qui ne définissent pas clone :

```
class MaClasse implements Cloneable{
    private Date date = new Date( "27 Aug 1966" ) ;
    public Object clone() throws CloneNotSupportedException{
        MaClasse obj = (MaClasse)super.clone() ;
        obj.date = new Date( date.getYear(), date.getMonth(), date.getDate() ) ;
        return obj ;
    }
    public static void main( String [] argv ){
        MaClasse o1 = new MaClasse() ;
        System.out.println( o1.tableau[ 0 ] ) ;
        try{
            MaClasse o2 = (MaClasse)o1.clone() ;      // clonage OK
            o2.date.setYear( 48 ) ;                   // seul o2 est modifié
        } catch( CloneNotSupportedException e ) {
            System.out.println( e ) ;
        }
        System.out.println( o1.date ) ;         // 27/8/66 : pas de changement
}
```

La méthode clone de la classe Object

- Pour interdire le clonage :
  - ne pas implémenter cloneable ;
  - définir clone en lançant systématiquement une exception.

```
class MaClasse {
    // ...
    public Object clone() throws CloneNotSupportedException{
        throw new CloneNotSupportedException() ;
    }
    public static void main( String [] argv ){
        MaClasse o1 = new MaClasse() ;
        try{
            MaClasse o2 = (MaClasse)o1.clone() ;
        } catch( CloneNotSupportedException e ) {
            System.out.println( e ) ;
        }
}
```